Anna-Lena Johansson
Agneta Eriksson-Granskog
Anneli Edman

Prolog Versus You

An Introduction to Logic Programming

With 56 Figures

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo Hong Kong

Anna-Lena Johansson
Agneta Eriksson-Granskog
Anneli Edman
Uppsala University
Computer Science Department
P.O. Box 520
S-751 20 Uppsala
Sweden

Translation of the original Swedish edition
En match med Prolog, published by
Studentlitteratur AB, Lund, Sweden

Distribution rights by Springer-Verlag:
Europe, except UK and Ireland, and outside
Europe (exclusive)

Distribution rights by Studentlitteratur:
United Kingdom, Ireland (exclusive) and Europe

ISBN-13:978-3-540-17577-3 e-ISBN-13:978-3-642-71922-6
DOI: 10.1007/978-3-642-71922-6

Library of Congress Cataloging-in-Publication Data
Johansson, Anna-Lena, 1951- [Match med Prolog. English] Prolog versus you : an introduction to logic programming / Anna-Lena Johansson, Agneta Eriksson-Granskog, Anneli Edman. p. cm.
Translation of: En match med Prolog.
Includes index.
ISBN-13:978-3-540-17577-3
1. Prolog (Computer program language) 2. Logic programming. I. Eriksson-Granskog, Agneta, 1957- II. Edman, Anneli, 1952- . III. Title. QA76.73.P76J6413 1989 006.3-dc 20 89-10102 CIP

2145/3140-543210 – Printed on acid-free paper

Preface

The idea behind logic programming is to use a logic system as a basis for programming and reasoning about programs. The language of logic is used both for specifications of programs and for programs. The declarative semantics of a program is given by the set of atomic formulas that are logical consequences of the program. The procedural semantics is the set of atomic formulas deducible from the program. The uniform formalism makes it possible to relate a program to its specification by means of a deduction. Deductions are used for program execution, program synthesis, program verification and program transformation. So far the most used logic programming language is Prolog.

Prolog was developed in 1972 at the University of Marseille, France, by Alain Colmerauer and Philippe Roussel. The purpose was to use the language as an aid to natural language understanding. At the 1974 IFIP conference in Stockholm, Robert Kowalski presented a paper called "Predicate logic as a programming language", which inspired to the use of Prolog for programming. In 1977 David Warren, Fernando Pereira, and Luis Pereira implemented an interpreter and a compiler for Prolog on DECsystem-10. These implementations made efficient program execution possible. Since then many implementations have been made. The diffusion of the language was at first confined to Europe but in later years it has won ground all over the world. There are both annual and biennial conferences on the subject of logic programming. In Japan Prolog was chosen as the basic language for the Fifth Generation Computer project which started in 1982. Subgoals of the project are to build both sequential and parallel Prolog machines.

Prolog has been in use in Sweden since 1975 when it was introduced at Stockholm University by Sten-Åke Tärnlund. Since then Sten-Åke Tärnlund has played an important part in the research and development of logic programming and in 1980 he started the research laboratory UPMAIL (Uppsala Programming Methodology and Artificial Intelligence Laboratory) at Uppsala University. The focus of the research at UPMAIL is logic programming and the areas are knowledge-based systems, programming methodology and efficient inference machines.

The authors have given graduate courses in logic programming at the Computing Science Department at Uppsala University for several years, both in the

Sy tems Science and the Computer Science programs. The programming language used for these courses was Prolog. The present work was developed as course literature, but does not presuppose a previous knowledge of Prolog or of programming. The book emphasizes the declarative approach to logic programming and the connection with logic plays an important role. The programming language described in the book does not correspond exactly to the syntax of any particular Prolog implementation.

We thank colleagues and friends for valuable comments on the manuscript and for their support and encouragement.

This book was produced with the help of the TEX computer typesetting system created by Donald E. Knuth at Stanford University. Special thanks are due to Gunnar Blomberg and Jonas Barklund whose help has been invaluable in preparing the camera-ready manuscript.

The book was translated from Swedish by Marianne Ahrne. We are grateful to her for her patient work and imaginative adaption of examples with locality in Uppsala and Sweden to other parts of the world.

Uppsala, March 1989

Anna-Lena Johansson
Agneta Eriksson-Granskog
Anneli Edman

Contents

Round 4. Databases and Expert Systems

Round 5. Program Methodology

Round 6. Efficient Computation

Round 7. Input and Output

Round 1. Logic Programs

1.1 World Descriptions

A *logic program* is a description of an imaginary or real world. In every world there are various objects and the program describes properties of the objects and relations between them. The description, i.e. the program, makes it possible to draw conclusions about the objects in the described world. Objects in the world may be concrete *individuals* such as persons and numbers, abstract concepts or abstract *structures* such as lists or trees. Two descriptions of the same world may be entirely different, focusing on different objects and relations, depending on who is looking at the world and the purpose of the observation. Our outlook is mirrored in the description; we include only those objects, properties and relations which we think important.

A *logical statement* is a sentence about which we can say that it is either true or false. We use the symbol $\top$ to signify truth and the symbol $\bot$ to signify absurdity. A *Prolog program* is a type of logic program in which the statements about the world are expressed in a certain form, called a *Horn form.* A statement in Horn form is called *Horn clause*[1].

We will visit many different worlds in this book. Let us start with an ancient legendary world, that of Norse mythology. This world is peopled with gods, giants and dwarves. Individuals in this world are Thor, Balder, Freya, Odin, Loki, and many more. Let us study the gods. Thor is a god in Norse mythology. We may say that one of his properties is that he is a Norse god. In a logic program we can describe this fact by writing:

$$\mathsf{Norse\_god(Thor)} \leftarrow \tag{1}$$

We express that a certain object has a property by writing the name of the property, the *property name*, followed by the name of the object possessing the property, within brackets. Other Norse gods beside Thor are Balder and Freya.

$$\mathsf{Norse\_god(Balder)} \leftarrow \tag{2}$$
$$\mathsf{Norse\_god(Freya)} \leftarrow \tag{3}$$

[1] After the logician A. Horn who was the first to study clauses in this form.

When we describe a world we intend a certain interpretation of the symbols we use. In (1), (2), and (3) we have a description of a world with the intended interpretation that Thor, Balder, and Freya denote the individuals Thor, Balder, and Freya, and Norse_god denotes the property of being a god in Norse mythology. The clauses (1), (2), and (3) express what we hold to be true, i.e. facts in our world. These clauses together make up our program.

Let us take a closer look at the language we use and define some concepts.

- The property name may also be called the *predicate name*. Predicate names are a joint term for names of properties and names of relations and the name is distinguished by an upper-case letter at the beginning. A name followed by its argument is called an *atomic predicate logic formula*, also known as a *predicate*. The number of arguments to a certain predicate is called the *arity* of the predicate. Properties are expressed by predicates with arity 1. We have met with the property Norse_god. Relations have arity 2 or higher (see the relations Father and Mother further on in this section). Predicates without arguments, i.e. with arity 0, are called *propositional formulas*. The atomic formulas are the basic concepts in a predicate logic language.
- The arguments to a predicate are *terms*. A term denotes an object in the world and is a *constant*, a *variable*, or a *structure*. A constant names a specific individual in the world. A constant may be a name, like Thor, for example, or a number. A variable denotes an arbitrary individual or structure in the world. Constants begin with an upper-case and variable names begin with a lower-case letter. Text constants that do not have an initial upper-case letter are put within inverted commas. A structure, also called a composite term, is composed of a *constructor* and terms.
- Apart from the atomic formulas there may be connecting operators, *logical connectives*, in a logical statement.

Thor and Balder are Æsir[2], the chief gods of the Norse pantheon, while Freya belongs to the family of Vanir, another race of gods. All individuals who are Æsir are Norse gods and all individuals who are Vanir are Norse gods, too. Suppose that we want to describe the fact that all individuals are Norse gods if they are Æsir. The clause must be valid for all individuals who are Æsir, thus we cannot use a constant like Thor to denote the object which has the property Norse_god. We must express the clause using a *variable* which can be exemplified by specific names for individuals. Thus, if we want to express the fact that all individuals are Norse gods, then we can put the variable x in the place of the argument.

$$\mathsf{Norse\_god(x)} \leftarrow \tag{4}$$

In statements (1), (2), and (3) we have given specific individuals instead of the variable x but otherwise the expressions are identical. To substitute

[2] Æsir from Æs, the old Norse plural.

individual names for variables in a formula is called to *instantiate* the formula. The clauses (1), (2), and (3) are three different *instances* of the statement (4). In (1) x is instantiated to Thor and in (2) and (3) to Balder and Freya, respectively. The statement (4) holds for all values on the variable x. The statement is said to be *universally quantified*; if it is true, then it is valid for all individuals in our universe.

In logic the fact that a variable is quantified is expressed by means of a *quantifier* followed by the quantified variable. We use a quantifier when we want to express the quantity, i.e. the number of individuals. We shall use the universal quantifier and the existential quantifier, denoted by the symbols $\forall$ and $\exists$, respectively.

- The $\forall$-quantifier is used to denote all objects. $\forall x$ is read as "for all x".
- The $\exists$-quantifier is used to denote one or several objects, i.e. not necessarily all. $\exists x$ is read as "for some x" or "there exists x".

The *scope* of a quantifier is the expression it precedes. The variable that is quantified is said to be *bound* in the expression. In the expression $\forall x\ P(x, y)$, x is a bound variable, but y is a *free* variable.

The quantifiers are not explicitly written in a Prolog program; it is implicit that all variables in a Prolog program are universally quantified. The Horn clause (4) is equivalent to the following universally quantified statement:

$$\forall x\ Norse\_god(x)$$

We can call this form the *standard form* of predicate logic to distinguish it from the Horn form. The statement may be read "all are Norse gods" or "for all x holds that x is a Norse god".

Expression (4) does not correspond to the statement that all are Norse gods if they are Æsir. The expression allows us to instantiate x with Loki, and Loki belongs to the tribe of giants. A condition for someone to be a Norse god is that he also has the property Æs. In a Prolog program we write

$$\mathsf{Norse\_god(x)} \leftarrow \mathsf{Æs(x)} \tag{5}$$

which reads "x is a Norse god if x is an Æs". This is a conditional expression. If A and B are two clauses, the clause $A \leftarrow B$ expresses that A is true if B is true. The expression $A \leftarrow B$ is false only when B is true and A is false. A is the *consequence* of the expression and B is its *condition* or *antecedent*. A conditional clause is also called *implication*. The condition entails or implies the consequence. A clause without conditions as, for example, clause (1) should be interpreted as a valid clause, i.e. an unconditional clause.

Note that the same variable name x must be used for the arguments to the properties Norse_god and Æs in (5), since we are discussing the same object. When we create an instance all occurrences of the variable must be replaced by the same value. The first clause below is an instance of (5):

$$\text{Norse\_god(Thor)} \leftarrow \text{Æs(Thor)} \quad (6)$$
$$\text{Norse\_god(x)} \leftarrow \text{Æs(y)} \quad (7)$$
$$\text{Norse\_god(Loki)} \leftarrow \text{Æs(Thor)} \quad (8)$$

But from the expression (7) that contains two different variables we can obtain the instances (6) and (8), among others. Making the quantifiers explicit, the expressions (5) and (7) will look like this:

$$\forall x\,(Norse\_god(x) \leftarrow AEs(x))$$
$$\forall x \forall y\,(Norse\_god(x) \leftarrow AEs(y))$$

Since expression (7) contains two different variables, the variables may assume different values. Statement (7) is a false statement because not all instances are true. Thor is an Æs but Loki is not a Norse god.

In (5) we have arrived at a formulation of the expression "all Æsir are Norse gods". That all Vanir are Norse gods can be expressed correspondingly. The complete characterization of the property Norse_god is thus made up of the two Horn clauses (5) and (9) together.

$$\text{Norse\_god(x)} \leftarrow \text{Æs(x)} \quad (5)$$
$$\text{Norse\_god(x)} \leftarrow \text{Van(x)} \quad (9)$$

The variable x is universally quantified in each of the two expressions, but the variable x in expression (5) has no connection with variable x in expression (9). The Horn clauses are implicitly universally quantified and the scope of a quantifier is only the clause it precedes. The name we choose for the variable is irrelevant to the meaning of the expression. Instead of the variable name x in (5) we may just as well use an arbitrary variable name, e.g. person.

$$\text{Norse\_god(person)} \leftarrow \text{Æs(person)} \quad (10)$$

Clause (10) is a *variant* of clause (5) with the same meaning.

Characterizing Thor, Balder and Freya by the properties Æs and Van, our world description now has this appearance:

$$\text{Norse\_god(x)} \leftarrow \text{Æs(x)} \quad (5)$$
$$\text{Norse\_god(x)} \leftarrow \text{Van(x)} \quad (9)$$
$$\text{Æs(Thor)} \leftarrow \quad (11)$$
$$\text{Æs(Balder)} \leftarrow \quad (12)$$
$$\text{Van(Freya)} \leftarrow \quad (13)$$

Prolog is a deductive system and to *execute* a logic program means to draw conclusions from the statements describing the world, i.e. the world description in the program is the premise of a deduction. Clauses (1), (2), and (3) follow from the new world description because all Æsir and all Vanir are Norse gods and Thor and Balder are Æsir and Freya is of the family of Vanir.

Let us have a look at some relations or connections between individuals as well. A relation has two or more arguments and is therefore a statement about

more than one object. Relations that may be of interest are kinship relations. The basic relations for expressing kinship are fatherhood and motherhood. The relation mother holds for an individual and his mother. We may decide that the relation shall bear the name Mother and have two arguments. The intended meaning of the relation is to say that the motherhood relation exists between the first and the second arguments, i.e. the second argument denotes the mother of the individual who was given as the first argument. The formula Mother(x,y) may be read "mother of x is y". So now we can describe the fact that the mother of Thor is Earth and analogously that the father of Thor is Odin.

$$\mathsf{Mother(Thor,Earth)} \leftarrow \tag{14}$$
$$\mathsf{Father(Thor,Odin)} \leftarrow \tag{15}$$

From the relations Mother and Father we may proceed to describe the relation Parent. The parent relation exists between two individuals x and y if the mother of x is y, and also if the father of x is y.

$$\mathsf{Parent(x,y)} \leftarrow \mathsf{Mother(x,y)} \tag{16}$$
$$\mathsf{Parent(x,y)} \leftarrow \mathsf{Father(x,y)} \tag{17}$$

We may describe the Sibling relation using the Parent relation. A sibling is a person y that has at least one parent in common with x. Note that it is immaterial whether we write the clause on one or several lines.

$$\begin{aligned}\mathsf{Sibling(x,y)} \leftarrow\ & \\ & \mathsf{Parent(x,xparent),Parent(y,yparent),} \\ & \mathsf{xparent} = \mathsf{yparent}\end{aligned} \tag{18}$$

The condition in the clause consists of three parts listed with commas in between them. The Sibling relation is valid if x has one parent denoted by the variable xparent, y has one parent denoted by yparent and the parents are one and the same, i.e. xparent and yparent shall denote the same individual. If the persons have the same name we assume that they are the same individual. We can identify identity by the clause

$$\mathsf{x} = \mathsf{x} \leftarrow \tag{19}$$

The condition that both parents have to be the same person can also be expressed by using the same variable name to denote the mutual parent. Let us use the variable name parent as second argument to the two Parent conditions.

$$\mathsf{Sibling(x,y)} \leftarrow \mathsf{Parent(x,parent),Parent(y,parent)} \tag{20}$$

A logic program consists of clauses which we believe to be true statements about our world. The consequence is true if all the atomic formulas in the condition are true. In Horn form we use commas to connect several conditions. In a predicate logic language the symbol $\wedge$ is used when we want to express the fact that several clauses have to be true for the composite formula to be true. The symbol $\wedge$ reads "and". The expression $A \wedge B$ is called *conjunction* and is

true if both A and B are true. In standard form we would formulate the clause (20) like this:

$$\forall x \forall y \forall parent\,(Sibling(x, y) \leftarrow Parent(x, parent) \wedge Parent(y, parent))$$

In our description of the relation Sibling, we lay down as a condition that there be one mutual parent. We can make harder demands by saying that both the father and the mother have to be mutual. Thus we get a relation which we can call Full_Sibling:

```
Full_Sibling(x,y) ←                                    (21)
        Father(x,father),Father(y,father),
        Mother(x, mother),Mother(y,mother)
```

We can choose to describe the kinship relations in other ways. Our world descriptions depend on our purpose. We may choose to view the different kinship relations mother, father, sister, and brother as objects in the world instead. Let us use a relation Kin with three arguments. The first two arguments denote persons and the third denotes a family or kinship relation. The relation Kin(x,y,z) can be read as "The kinship relation between x and y is z". In the Kin relation we can describe all family relationships in a common description since we view the family relationship itself as an object in the world. Thus we can describe the fact that Earth is the mother of Thor and Odin is his father. We can also describe the relationships between parents and children using the relation Kin.

```
Kin(Thor,Earth,Mother) ←
Kin(Thor,Odin,Father) ←
Kin(x,y,Parent) ←
        Kin(x,y,Mother)
Kin(x,y,Parent) ←
        Kin(x,y,Father)
Kin(x,y,Sibling) ←
        Kin(x,parent,Parent), Kin(y,parent,Parent)
```

Let us take one more look at the property Æs to extend our description. Who are the Æsir? The great father of the Æsir is Odin; he is the progenitor of their race. Thus Odin is an Æs as described in clause (22). All who can trace their origin to Odin are Æsir. The children of Odin, above all, are Æsir as is described in clause (23). Also the grandchildren and great grandchildren, etc., of Odin are Æsir. Generally speaking, an individual is an Æs if one of his or her parents is an Æs. Horn clauses (22) and (24) together make up the description of the Æs property.

```
Æs(Odin) ←                                             (22)
Æs(x) ← Parent(x,y), y = Odin                          (23)
Æs(x) ← Parent(x,y), Æs(y)                             (24)
```

Odin was the father of Thor according to (15) and Thor in his turn is the father of Trud and Magni.

$$\begin{aligned} &\mathsf{Father(Thor,Odin)} \leftarrow && (15)\\ &\mathsf{Father(Trud,Thor)} \leftarrow && (25)\\ &\mathsf{Father(Magni,Thor)} \leftarrow && (26) \end{aligned}$$

It follows that Thor is an Æs according to our description because he has a father who is an Æs. Magni is also an Æs since Thor is one of his parents. Trud is an Æs according to our description because she has an Æs as father, although the correct Norse term for a female Æs, or goddess, is actually "asynja".

1.2 Clauses

1.2.1 Horn Clauses

Let us have a look at the Horn form in general. We have learnt that logic programs are expressed in Horn form. We have used clauses (5), (22), and (24) to describe the old Norse pantheon.

$$\begin{aligned} &\mathsf{Norse\_God(x)} \leftarrow \mathsf{Æs(x)} && (5)\\ &\mathsf{Æs(Odin)} \leftarrow && (22)\\ &\mathsf{Æs(x)} \leftarrow \mathsf{Parent(x,y)},\ \mathsf{Æs(y)} && (24) \end{aligned}$$

Let us take a look at the form these clauses have in common. Let us use the letters A and B to denote arbitrary atomic formulas. Furthermore, let us apply the convention of calling atomic formulas in the conditional part of a Horn clause B and atomic formulas in the consequence A. When more than one formula is included in a part we use subscripts to separate them, i.e. $\mathsf{B}_1, \ldots, \mathsf{B}_j$ and $\mathsf{A}_1, \ldots, \mathsf{A}_m$.

Clause (5) has exactly one atomic formula in the consequence and exactly one in the condition. The form can be characterized in the following schema:

$$\mathsf{A} \leftarrow \mathsf{B}$$

Clauses (22) and (24) both have one formula in the consequence but a varying number in the condition.

$$\begin{aligned} &\mathsf{A}_1 \leftarrow\\ &\mathsf{A}_2 \leftarrow \mathsf{B}_1,\ \mathsf{B}_2 \end{aligned}$$

In a Horn clause program we use clauses with an arbitrary number of atomic formulas in the conditional part but exactly *one* atomic formula in the consequence. We can summarize permitted program clauses in the following schema:

$$\mathsf{A} \leftarrow \mathsf{B}_1, \ldots, \mathsf{B}_j \qquad (27)$$

Let us study the parallels in standard form. Let $x_1, \ldots, x_n$ denote the variables in A and $B_1, \ldots, B_j$. Then the sentence corresponding to (27) is

$$\forall x_1 \ldots \forall x_n (A \leftarrow B_1 \wedge \cdots \wedge B_j)$$

The condition may be empty, as we noticed in clause (22), i.e. j = 0. Clauses with j = 0 mean that A holds unconditionally, i.e. clause A is true. The correspondence in standard form is

$$\forall x_1 \ldots \forall x_n \, (A \leftarrow \top) \text{ or } \forall x_1 \ldots \forall x_n \, A$$

The general Horn form also permits the consequence to be empty. We use Horn clauses with an empty consequence for formulating questions to the program. This will be discussed in Round 2. Clauses with an empty consequence mean that not all formulas B_1, ..., B_j are true.

$$\leftarrow \; B_1, \ldots, B_j$$

The equivalence in standard form is

$$\forall x_1 \ldots \forall x_n \, (\bot \leftarrow B_1 \wedge \cdots \wedge B_j) \text{ or } \forall x_1 \ldots \forall x_n \, \neg (B_1 \wedge \cdots \wedge B_j)$$

With this type of clause we express the fact that Loki is not a Norse god. We can also express the fact that no one can be both man and woman.

$$\leftarrow \; \text{Norse\_god(Loki)}$$
$$\leftarrow \; \text{Man(x), Woman(x)}$$

Thus we have a *negation*. Negation is expressed in standard form by the symbol $\neg$. The clause $\neg A$ is false if A is true and true if A is false. $\neg A$ consequently has the opposite truth value of A. If both the consequence and the condition are empty we have a contradiction. That is, we get $\bot \leftarrow \top$ which is a false statement.

We have seen that the standard form of predicate logic uses more logical symbols than the Horn form. In standard form the logical symbols are the quantifiers $\forall$, $\exists$ and the logical connectives $\wedge$, $\vee$, $\rightarrow$, $\leftarrow$, $\leftrightarrow$ and $\neg$ (which are read as "and", "or", "if ... then ...", "... if ...", "if and only if", and "not".

Most often the programs have more than one Horn clause. The program clauses are all statements about the world which we hold to be valid taken together. We have a conjunction of clauses and, consequently, the two Horn clauses (28) and (29) and the standard form conjunction below are equivalent expressions.

$$\text{A} \leftarrow \; \text{B}_1, \ldots, \text{B}_j \tag{28}$$
$$\text{C} \leftarrow \; \text{D}_1, \ldots, \text{D}_m \tag{29}$$

$$\forall x_1 \ldots \forall x_n (A \leftarrow B_1 \wedge \cdots \wedge B_j) \wedge \forall x_1 \ldots \forall x_k (C \leftarrow D_1 \wedge \cdots \wedge D_m)$$

A special case of this is when we have two Horn clauses with the same consequence and the same variables denoting corresponding arguments:

$$\mathsf{A} \leftarrow \mathsf{B}_1, \ldots, \mathsf{B}_j \tag{28}$$
$$\mathsf{A} \leftarrow \mathsf{D}_1, \ldots, \mathsf{D}_m \tag{30}$$

$$\forall x_1 \ldots \forall x_n((A \leftarrow B_1 \wedge \cdots \wedge B_j) \wedge (A \leftarrow D_1 \wedge \cdots \wedge D_m))$$

An expression in the form $(A \leftarrow B) \wedge (A \leftarrow D)$ may be transformed to the form $A \leftarrow B \vee D$. Clauses (28) and (30) may thus be written in standard form as:

$$\forall x_1 \ldots \forall x_n(A \leftarrow (B_1 \wedge \cdots \wedge B_j) \vee (D_1 \wedge \cdots \wedge D_m))$$

To summarize:
Program clauses and questions are formulated in Horn form. Clauses in a program are connected by conjunction.

- Program clauses: $\mathsf{A} \leftarrow \mathsf{B}_1, \ldots, \mathsf{B}_j$, where j can be greater than or equal to zero.
- Questions: $\leftarrow \mathsf{B}_1, \ldots, \mathsf{B}_j$

1.2.2 Clausal Form

When we generalize the Horn form so that the consequence too can contain more than one atomic formula, we have a *clausal form.* Every predicate logic formula in standard form may be transformed into clausal form. A clause has this general appearance:

$$\mathsf{A}_1, \ldots, \mathsf{A}_m \leftarrow \mathsf{B}_1, \ldots, \mathsf{B}_j \tag{31}$$

We have interpreted the commas between the formulas in the conditional part as the symbol $\wedge$. The commas between formulas in the consequential part, on the other hand, correspond to the logical connective $\vee$. $A \vee B$ is called a *disjunction* and is true if any of the formulas A or B is true. $A \vee B$ is thus false only if both A and B are false.

Let $x_1, \ldots, x_n$ denote the variables occurring in $A_1, \ldots, A_m$ and $B_1, \ldots, B_j$. The formula corresponding to (31) in standard form is

$$\forall x_1 \ldots \forall x_n(A_1 \vee \cdots \vee A_m \leftarrow B_1 \wedge \cdots \wedge B_j)$$

If $\mathsf{j} = 0$ in a clause then $\mathsf{A}_1, \ldots, \mathsf{A}_m$ holds unconditionally, i.e. any of the clauses $\mathsf{A}_1, \ldots, \mathsf{A}_m$ is true. The corresponding expression in standard form is

$$\forall x_1 \ldots \forall x_n\,(A_1 \vee \cdots \vee A_m \leftarrow \top)$$
$$\text{or } \forall x_1 \ldots \forall x_n\,(A_1 \vee \cdots \vee A_m)$$

For instance, propositions (5), (22) and (24) are clauses. Their form can be described by the schema (31). Program clauses are clauses with $\mathsf{m} = 1$. Not all expressions in standard form can be transformed to program clauses using the same predicates and the same logical symbols. Concomitantly, we deal with only a subpart of predicate logic when we program in Prolog.

1.3 Definitions

1.3.1 Introduction

A definition is an agreement on the meaning of an expression. The expression to be defined is called the *definiendum.* The definition relates the definiendum to a defining formula, the *definiens.* The expressions used in the definiens must already have been defined in their entirety before the definition is applied.

Let us begin to study definitions given in standard form and then proceed to study how they can be represented in Horn form. A definition has the following form, where A is the predicate to be defined and D is the definiens[3]:

$$\forall x_1 \cdots \forall x_n (A(x_1, \ldots, x_n) \leftrightarrow D)$$

The following restrictions shall hold:

i) $x_1, \ldots, x_n$ are distinct variable names

ii) D contains no free variables except $x_1, \ldots, x_n$ from the definiendum

iii) D is a logical expression which, apart from the logical constants, contains only concepts which are already defined in their entirety when the definition is used

The defined relation should be removable from clauses where it is used by replacing the definiendum by the definiens.

A definition of the sibling relation has the following appearance:

$$\forall x \forall y \, (Sibling(x, y) \leftrightarrow \exists \, parent \, (Parent(x, parent) \wedge Parent(y, parent)))$$

i.e. "for all x and y holds that a sibling of x is y if, and only if, there is someone who is parent both of x and y".

Let us discuss the restrictions on the form of definitions using the above example. The first restriction prevents us from writing the following definition

$$\forall x \, (Sibling(x, x) \leftrightarrow \exists \, parent \, (Parent(x, parent)))$$

The definition is certainly true in so far as everybody can be regarded as a sibling of themselves but it does not promote an understanding of the concept of sibling. It is unnecessary to name a variable more than once in the definiendum as the same variable name used for several arguments can be reduced to only one.

We try to define *Sibling* once more to find out what a violation of the second restriction will lead to.

$$\forall x \forall y \forall parent \, (Sibling(x, y) \leftrightarrow Parent(x, parent) \wedge Parent(y, parent))$$

---

[3] P. Suppes, *Introduction to Logic*, van Nostrand, New York, 1957.

The variables x and y are named in the definiendum. In addition to these the definiens also contains the free variable *parent*. Let us perform some transformations of the definition to investigate the meaning of the formula. An expression $A \leftrightarrow B$ can be transformed to $(A \rightarrow B) \wedge (A \leftarrow B)$; consequently for the definition we get the expression

$$\forall x \forall y \forall parent\,((Sibling(x,y) \rightarrow Parent(x,parent) \wedge Parent(y,parent)) \\ \wedge\,(Sibling(x,y) \leftarrow Parent(x,parent) \wedge Parent(y,parent)))$$

Furthermore, the expression $\forall x\,(A \wedge B)$ can be transformed to $\forall x\,A \wedge \forall x\,B$.

$$\forall x \forall y \forall parent\,(Sibling(x,y) \rightarrow Parent(x,parent) \wedge Parent(y,parent)) \\ \wedge\,\forall x \forall y \forall parent\,(Sibling(x,y) \leftarrow Parent(x,parent) \wedge Parent(y,parent))$$

A formula $\forall x\,(A \rightarrow B(x))$ can be transformed to $A \rightarrow \forall x\,B(x)$ on condition that x does not occur in A. $\forall x\,(A \leftarrow B(x))$ can be transformed to $A \leftarrow \exists x\,B(x)$ on the same condition.

$$\forall x \forall y(Sibling(x,y) \rightarrow \forall parent\,(Parent(x,parent) \wedge Parent(y,parent))) \\ \wedge\,\forall x \forall y(Sibling(x,y) \leftarrow \exists\,parent\,(Parent(x,parent) \wedge Parent(y,parent)))$$

The above formula is a contradiction. We can derive the invalid expression

$$\forall x \forall y\,(\exists parent\,(Parent(x,parent) \wedge Parent(y,parent)) \rightarrow \\ \forall parent\,(Parent(x,parent) \wedge Parent(y,parent)))$$

The formula may be read "for all x and y holds that if someone is a parent of both x and y then everyone is a parent of both x and y". We can easily find false instances of this expression.

The third restriction is necessary to avoid definitions of the following type:

$$\forall x \forall y\,(Sibling(x,y) \leftrightarrow Brother(x,y) \vee Sister(x,y)) \\ \forall x \forall y\,(Brother(x,y) \leftrightarrow Sibling(x,y) \wedge Man(x))$$

The concept *Sibling* as defined here cannot be removed from a logical formula by replacing the definiendum by the definiens using the above definitions.

1.3.2 Definitions - Programs

Let us return to the correct standard form definition of *Sibling* to see how the program relates to the definition. We can transform the definition into an expression consisting of two conditional clauses.

$$\forall x \forall y((Sibling(x,y) \rightarrow \exists parent(Parent(x,parent) \wedge Parent(y,parent))) \\ \wedge\,(Sibling(x,y) \leftarrow \exists\,parent(Parent(x,parent) \wedge Parent(y,parent))))$$

The expression $\forall x\,(A \wedge B)$ can be transformed into $\forall x\,A \wedge \forall x\,B$.

$$\forall x \forall y (Sibling(x,y) \rightarrow \exists\, parent\,(Parent(x,parent) \wedge Parent(y,parent)))$$
$$\wedge\, \forall x \forall y (Sibling(x,y) \leftarrow \exists\, parent\,(Parent(x,parent) \wedge Parent(y,parent)))$$

Furthermore, a formula $A \leftarrow \exists x\,B(x)$ can be transformed to $\forall x\,(A \leftarrow B(x))$ on condition that x does not occur in A, and $A \rightarrow \exists x\,B(x)$ can be transformed to $\exists x\,(A \rightarrow B(x))$ on the same condition.

$$\forall x \forall y\, \exists\, parent\,(Sibling(x,y) \rightarrow Parent(x,parent) \wedge Parent(y,parent))$$
$$\wedge\, \forall x \forall y\, \forall parent\,(Sibling(x,y) \leftarrow Parent(x,parent) \wedge Parent(y,parent))$$

We notice that the second part of the above conjunction is a clause that has a correspondence in Horn form, i.e. clause (20).

$$\mathsf{Sibling(x,y)} \leftarrow \mathsf{Parent(x,parent),Parent(y,parent)} \tag{20}$$

Thus the descriptions in the Prolog program uses only the "if" part of a complete definition. Concomitantly, the definiens can be exchanged for the definiendum. The second part, i.e. the "only if" part, adds that $Sibling(x,y)$ is true only if the given criteria are satisfied. We do not increase the ways of deriving $Sibling(x,y)$ if the descriptions are completed. Thus we cannot draw more conclusions about $Sibling(x,y)$ from the equivalence clause than from (20), therefore, the Horn form does not limit the conclusions about the definiendum that can be drawn from the definition[4].

The set of Horn clauses which characterizes a predicate is the description of the predicate in the program. We will call it a definition given in Horn form. Every property or relation in the programs is defined by the clauses where its name appears in the consequence. The relation Parent, for instance, may be defined by two Horn clauses (16) and (17).

$$\mathsf{Parent(x,y)} \leftarrow \mathsf{Mother(x,y)} \tag{16}$$
$$\mathsf{Parent(x,y)} \leftarrow \mathsf{Father(x,y)} \tag{17}$$

Expressed in standard form

$$\forall x \forall y\,(Parent(x,y) \leftarrow Mother(x,y) \vee Father(x,y))$$

Sometimes we are interested in producing a standard form definition of a program by simply adding the "only if" part. This definition is called the completed program. The procedure is to make all the consequences of the program identical and combine the conditions from the different clauses by disjunction. The program

$$\mathsf{P(A,B)} \leftarrow \mathsf{Q}$$

[4] See Sect. 4.1 for a discussion about ways of deriving negation, i.e. the negated definiendum.

$$\mathsf{P(x,C)} \leftarrow \mathsf{R}$$

is first transformed to

$$\begin{aligned}\mathsf{P(x,y)} &\leftarrow \mathsf{x} = \mathsf{A},\ \mathsf{y} = \mathsf{B},\ \mathsf{Q}\\ \mathsf{P(x,y)} &\leftarrow \mathsf{y} = \mathsf{C},\ \mathsf{R}\end{aligned}$$

The completed program is

$$\forall x \forall y\,(P(x,y) \leftrightarrow (x = A \wedge y = B \wedge Q) \vee (y = C \wedge R))$$

1.3.3 Construction of Definitions

We will leave the Norse mythology and family relations behind and look at some definitions in a world consisting of clauses and clause constructions. This time we will describe an imaginary world of abstract objects, their properties, and relations. We make the definitions in Horn form and accompany them with the corresponding completed definition in standard form, i.e. the completed program.

A predicate can be defined by enumerating all valid instances. Let us define a language in which the symbols A, B, C, False, and True stand for simple clauses. The simple clauses are objects in the world for which the property Unitclause holds. The symbols False and True represent a clause which is always false and a clause which is always true, respectively.

$$\begin{aligned}&\mathsf{Unitclause(A)} \leftarrow && (32)\\ &\mathsf{Unitclause(B)} \leftarrow && (33)\\ &\mathsf{Unitclause(C)} \leftarrow && (34)\\ &\mathsf{Unitclause(False)} \leftarrow && (35)\\ &\mathsf{Unitclause(True)} \leftarrow && (36)\end{aligned}$$

A clause (32) may be rewritten stating that Unitclause(symbol) holds if symbol = A, i.e. the condition for the argument, is brought to the conditional part of the clause. Clauses (32) – (36) will then have the following appearance:

$$\begin{aligned}\mathsf{Unitclause(symbol)} &\leftarrow \mathsf{symbol} = \mathsf{A}\\ \mathsf{Unitclause(symbol)} &\leftarrow \mathsf{symbol} = \mathsf{B}\\ \mathsf{Unitclause(symbol)} &\leftarrow \mathsf{symbol} = \mathsf{C}\\ \mathsf{Unitclause(symbol)} &\leftarrow \mathsf{symbol} = \mathsf{False}\\ \mathsf{Unitclause(symbol)} &\leftarrow \mathsf{symbol} = \mathsf{True}\end{aligned}$$

The completed program corresponding to both the programs for Unitclause is the following:

$$\begin{aligned}\forall symbol\,(&Unitclause(symbol) \leftrightarrow\\ &symbol = A \vee symbol = B \vee symbol = C\\ &\vee symbol = False \vee symbol = True)\end{aligned}$$

For the method of defining by enumeration to be feasible, the definition domain has to be very small. In other cases we make a definition through a description in which we use concepts which are already known or whose definition is being given. To build up a definition from fundamental concepts is called *bottom-up* construction. The use of the definition is independent of the order in which the hierarchy of definitions was created. The important thing is that all objects mentioned should have been defined when we use the definition in a program. In the Section 1.1 on World Descriptions we described the sibling relation bottom-up by first defining the relations Father and Mother. Building on these we constructed the relation Parent which we finally used to put together the definiens for the definition of Sibling.

Let us construct a new definition bottom-up. The definition will also illustrate the use of composite terms. A constant is a simple term that cannot be divided into parts. A structure, on the other hand, is a composite term where we use a constructor to compose several terms. If we wish to represent a point on a two-dimensional plane we have to construct a term consisting of two numbers, an x-coordinate and an y-coordinate. To join the two numbers we use a constructor; let us give it the name Dim2. Thus, one point is represented by the structure Dim2(x,y). We may continue to represent a point on a three-dimensional plane as the construction of a point on a two-dimensional plane and one additional number. The structure Dim3(Dim2(x,y),z) represents a point on a three-dimensional plane.

Let us return to the definition. Our basic concept is the property Unitclause as defined in clauses (32) – (36) and from this we can construct the property Conditional. The property Conditional holds for a structure constructed with the constructor If. The basic elements of the structure are unit clauses.

$$\begin{aligned}&\mathsf{Conditional(If(clause1,clause2))} \leftarrow \\ &\qquad \mathsf{Unitclause(clause1)},\ \mathsf{Unitclause(clause2)}\end{aligned} \tag{37}$$

Or equivalent:

$$\begin{aligned}&\mathsf{Conditional(clause)} \leftarrow \mathsf{clause} = \mathsf{If(clause1,clause2)}, \\ &\qquad \mathsf{Unitclause(clause1)},\ \mathsf{Unitclause(clause2)}\end{aligned}$$

In this definition the variable clause denotes a structure consisting of the constructor If and two terms named clause1 and clause2. The complete definition will eventually have the following appearance. The variables *clause*1 and *clause*2 do not appear in the definiendum, and consequently are quantified in the definiens.

$$\begin{aligned}&\forall \mathit{clause}\,(\mathit{Conditional}(\mathit{clause}) \leftrightarrow \\ &\qquad \exists \mathit{clause}1\,\exists \mathit{clause}2\,(\mathit{clause} = \mathit{If}(\mathit{clause}1, \mathit{clause}2) \\ &\qquad\qquad \wedge\ \mathit{Unitclause}(\mathit{clause}1) \\ &\qquad\qquad \wedge\ \mathit{Unitclause}(\mathit{clause}2))\end{aligned}$$

A third way of constructing a definition is *top-down*, beginning at an abstract level and gradually defining the concepts used until we are using fundamental concepts which are already familiar. In this case we use concepts during the construction which are not yet defined. Let us define a clause as consisting of a constructor If, a conjunction, and a disjunction.

$$\begin{aligned}\mathsf{Clause(clause)} \leftarrow\ & \mathsf{clause} = \mathsf{If(clause1,clause2)}, \\ & \mathsf{Disjunction(clause1)},\ \mathsf{Conjunction(clause2)}\end{aligned} \tag{38}$$

We continue the definition by defining Disjunction as either consisting of a simple clause or a structure built up by the constructor Or, a simple clause, and a disjunctive clause.

$$\mathsf{Disjunction(clause)} \leftarrow \mathsf{Unitclause(clause)} \tag{39}$$

$$\begin{aligned}\mathsf{Disjunction(clause)} \leftarrow\ & \mathsf{clause} = \mathsf{Or(clause1,clause2)}, \\ & \mathsf{Unitclause(clause1)},\ \mathsf{Disjunction(clause2)}\end{aligned} \tag{40}$$

The definition of Conjunction is analogous. Instead of the constructor Or, we use the constructor And.

$$\mathsf{Conjunction(clause)} \leftarrow \mathsf{Unitclause(clause)} \tag{41}$$

$$\begin{aligned}\mathsf{Conjunction(clause)} \leftarrow\ & \mathsf{clause} = \mathsf{And(clause1,clause2)}, \\ & \mathsf{Unitclause(clause1)},\ \mathsf{Conjunction(clause2)}\end{aligned} \tag{42}$$

Or equivalent:

$$\mathsf{Conjunction(clause)} \leftarrow \mathsf{Unitclause(clause)} \tag{43}$$

$$\begin{aligned}\mathsf{Conjunction(And(clause1,clause2))} \leftarrow\ & \\ & \mathsf{Unitclause(clause1)},\ \mathsf{Conjunction(clause2)}\end{aligned} \tag{44}$$

The definition in standard form is completed by "only if".

$$\begin{aligned}\forall \mathit{clause}\,(\,&\mathit{Conjunction}(\mathit{clause}) \leftrightarrow \\ &\mathit{Unitclause}(\mathit{clause}) \\ &\vee\ \exists \mathit{clause}1\ \exists \mathit{clause}2\,(\mathit{clause} = \mathit{And}(\mathit{clause}1, \mathit{clause}2) \\ &\qquad\qquad \wedge\ \mathit{Unitclause}(\mathit{clause}1) \\ &\qquad\qquad \wedge\ \mathit{Conjunction}(\mathit{clause}2)))\end{aligned}$$

The definientia of Disjunction and Conjunction both refer to the predicate to be defined, i.e. Disjunction and Conjunction, respectively. This type of definition is called a *recursive definition.* A recursive definition of a structure is given by first defining the basic elements in the structure; it constitutes the so-called *base case* of the definition, see clauses (39) and (41). A recursive definition always has at least one base case. We proceed by defining a structure in terms of its constituent parts, see (40) and (42). This is called the *recursive case* of the definition. The structure referred to in the definiens should always be less complex than the structure in the definiendum. Circular definitions are avoided this way. Eventually the structure will be reduced to basic elements. Every use

of the recursive definition will peel a constructor off; thus, the complexity of the structure is reduced. We say that we define the property through induction. When defining a relation instead of a property, we state what argument we reduce in the recursive definition by saying, for instance, that we define the relation by induction over the first argument.

Let us check whether the restriction that the defined concept should be removable from a formula is valid for a recursive definition. We test the restriction by trying to reduce Conjunction by exchanging the definiendum for the definiens. We use the program clauses (43) and (44).

Conjunction(And(A,And(B,C)))
is reduced to
Unitclause(A),Conjunction(And(B,C))
which is reduced to
Unitclause(A),Unitclause(B), Conjunction(C)
Finally we get
Unitclause(A), Unitclause(B), Unitclause(C)

In the last line the concept of Conjunction is expressed using only the concept Unitclause. These concepts are not defined inductively and it is obvious that the definition satisfies the restriction for definitions. Therefore, the restriction is satisfied for recursive definitions as well, provided that they are constructed so that the complexity of a structure is gradually reduced by the recursive cases in a way that in the end makes the base cases of the definition applicable. A recursive definition must contain at least one case without recursion. Several base cases may be necessary, but we can be certain that the definition is not correct if base cases are missing.

Let us take a look at one more example of a recursive definition. Definitions that are to hold for all natural numbers are given as recursive definitions. The basic element is the number 0. The subsequent numbers may be constructed with a constructor S. S(x) stands for the number following the number x in the succession of numbers. S(x) is called the successor of the number x.

S(0) denotes the number 1,
S(S(0)) the number 2, etc.

Let us define a relation Conjunctive_depth between two structures, the first constructed by the constructor And and the second by the constructor S. The construction with S represents a number that indicates *the depth* of the first structure. The depth is a measure of the complexity of the structure.

In the base case clause is a simple clause and the depth is 0. In the recursive case clause represents a construction with And. The depth of this construction is dependent on the depth of the constituent parts clause1 and clause2. clause1 has to be a simple clause and thus it does not contribute to the depth. The depth of clause2 is represented by the prec_depth. The depth of clause is a step higher than the depth of clause2 because clause contains one more constructor

And. The depth of clause is thus the successor of the depth of clause2.

$$\text{Conjunctive\_depth(clause, depth)} \leftarrow \text{Unitclause(clause)}, \quad \text{depth} = 0 \tag{45}$$

$$\text{Conjunctive\_depth(clause,depth)} \leftarrow \text{clause} = \text{And(clause1,clause2)}, \text{Unitclause(clause1)}, \text{Conjunctive\_depth(clause2, prec\_depth)}, \text{depth} = \text{S(prec\_depth)} \tag{46}$$

Valid instances of the relation are, for example,

Conjunctive_depth(And(A,And(B,C)),S(S(0)))
Conjunctive_depth(A,0)

Let us define a relation Sum_of_triads. This is a relation between a natural number num and a number sum representing a sum. The sum is a sum of numbers, each of which is divisible by three and less than or equal to the number num. Some valid instances of the relation Sum_of_triads follow below.

Sum_of_triads(S(S(S(0))),S(S(S(0)))), i.e. Sum_of_triads(3,3)
Sum_of_triads(S(S(S(S(S(0))))),S(S(S(0)))), i.e. Sum_of_triads(5,3)
Sum_of_triads(S(S(S(S(S(S(0)))))),S(S(S(S(S(S(S(S(S(0)))))))))),
i.e. Sum_of_triads(6,9)

The definition is given by induction over the natural number in the first argument. When num is equal to 0 there are no numbers less than zero; thus, the sum is 0.

$$\text{Sum\_of\_triads(num, sum)} \leftarrow \text{num=0}, \quad \text{sum} = 0 \tag{47}$$

The recursive case when num is a structure is divided into two cases depending on whether num is divisible by three or not. Thus we have two recursive clauses. We define a property Triad for a number that is divisible by three. If a number is not divisible by three the property NonTriad holds. The properties Triad and NonTriad partition the two natural numbers into two parts. We define Sum_of_triads top-down, i.e. we use the predicates Triad, NonTriad and Sum which are not yet defined.

If S(prec_num) is divisible by three, the sum is equal to the sum for prec_num and num. If on the other hand, S(prec_num) is not divisible by three, the sum is equal to the sum for prec_num.

$$\text{Sum\_of\_triads(num,sum)} \leftarrow \text{num} = \text{S(prec\_num)}, \text{Triad(num)}, \text{Sum\_of\_triads(prec\_num,prev\_sum)}, \text{Sum(num,prev\_sum,sum)} \tag{48}$$

$$\text{Sum\_of\_triads(num,sum)} \leftarrow \text{num} = \text{S(prec\_num)}, \text{NonTriad(num)}, \text{Sum\_of\_triads(prec\_num,prev\_sum)}, \text{prev\_sum} = \text{sum} \tag{49}$$

The complete definition in standard form is

$$\begin{aligned}
&\forall num\, \forall sum\, (Sum_of_triads(num, sum) \leftrightarrow \\
&\quad num = 0 \wedge sum = 0 \\
&\quad \vee \exists prec_num \exists prev_sum (num = S(prec_num) \\
&\qquad\qquad \wedge Sum_of_triads(prec_num, prev_sum) \\
&\qquad\qquad \wedge (Triad(num) \wedge Sum(num, prev_sum, sum) \\
&\qquad\qquad\quad \vee Non\ Triad(num) \wedge prev_sum = sum)))
\end{aligned}$$

A number is equally divisible by three if we can group the S constructors of the number in groups of three S's each, i.e. a number has the property Triad if we can reduce it to the number zero by reducing it successively by three S's.

$$\begin{aligned}
&\mathsf{Triad(num)} \leftarrow \mathsf{num = 0} && (50)\\
&\mathsf{Triad(num)} \leftarrow \mathsf{num = S(S(S(smaller_num))),} && (51)\\
&\qquad \mathsf{Triad(smaller_num)}
\end{aligned}$$

A number is not equally divisible by three if we can reduce the number to S(0) or S(S(0)) by successively removing three S's.

$$\begin{aligned}
&\mathsf{NonTriad(num)} \leftarrow \mathsf{num = S(0)} && (52)\\
&\mathsf{NonTriad(num)} \leftarrow \mathsf{num = S(S(0))} && (53)\\
&\mathsf{NonTriad(num)} \leftarrow \mathsf{num = S(S(S(smaller_num))),} && (54)\\
&\qquad \mathsf{NonTriad(smaller_num)}
\end{aligned}$$

For the summation we use the relation Sum which we also define recursively over the natural numbers. The sum of 0 and any other number is equal to this other number. Compare $0 + num2 = num2$. If the first number is not 0, i.e. it is constructed with the constructor S, e.g. S(prec_num), then the sum is the successor of the sum we get if we add S(prec_num) and num2. Compare $S(prec\_num) + num2 = S(prec\_num + num2)$.

$$\begin{aligned}
&\mathsf{Sum(num1,num2,sum)} \leftarrow \mathsf{num1 = 0,} && (55)\\
&\qquad \mathsf{sum = num2} \\
&\mathsf{Sum(num1,num2,sum)} \leftarrow \mathsf{num1 = S(prec_num),} && (56)\\
&\qquad \mathsf{sum = S(prev_sum),} \\
&\qquad \mathsf{Sum(prec_num,num2,prev_sum)}
\end{aligned}$$

Another example of a recursive definition, but not using a structure, is the definition of the property Æs given in clauses (21) and (22).

Top-down definitions are applied in *structured programming*. When we give a top-down definition we can postpone the details to lower levels, the auxiliary definitions, and concentrate on the overall view of the program. The definition becomes step-wise more and more detailed until we reach a level of concepts already known; the definition is stepwise refined.

"If-and-only-if" definitions given in standard form may be said to be the specification of a program. A specification is an abstract description of a relation, not necessarily computationally useful, where the important thing is

to characterize the relation without any thoughts about how easy it will be to draw conclusions from the specification. The definitions we have seen so far, have been on the same level of abstraction as the program clauses. The program clauses can thus be obtained by simple transformations of the specification. Very often, however, the specification of the program will be on a higher level of abstraction than the program and will in fact be a description of a class of programs that satisfies the specification. In such a case it is not trivial to show that a certain program is a consequence of the specification. But if we can show this formally we can be assured that the program agrees with the specification. Derivations of programs in Horn form from specifications given in standard form will be treated in greater detail in Round 5.

1.4 Exercises

Exercise 1:
Give the definition for a relation Playmate in standard form as well as in Horn form. The relation shall hold for all individuals living in the same neighborhood.

Exercise 2:
Try to define multiplication by using the constructor S to represent numbers.

Exercise 3:
Write down the relation Part which has two arguments, one is a simple clause and the other a disjunctive clause constructed with the constructor Or. The relation shall hold if the simple clause is an element of the disjunctive clause.

Exercise 4:
Game is divided into big and small game. Big game is moose, deer, wild boar, and lion among others. Small game is fox, rabbit, and birds. Define the property Game.

Exercise 5:
In judo one competes in different classes depending on one's body weight. Define a relation between a weight and a class. Suppose that the relations $<$, $\leq$, and $>$ are defined.

Lightweight	up to 63 kg
Welterweight	64-70 kg
Middleweight	71-80 kg
Light heavyweight	81-93 kg
Heavyweight	over 93 kg

Round 2. Execution of Logic Programs

2.1 Horn Clause Proof Procedure

2.1.1 Resolution

The time has come for us to find out how conclusions can be drawn from our world descriptions, i.e. how to execute Prolog programs. The conclusions are verified or refuted by a deduction. The clauses of the program are the premises of the deduction. A deduction consists of a sequence of deduction steps. To create a new step in the sequence we apply an *inference rule* either on premises or on formulas we have obtained from previous deduction steps. An inference rule states how a formula will follow from certain premises, i.e. an inference rule takes a set of premises and produces a conclusion:

$$\frac{P_1 \ \cdots P_n}{S}$$

S is a conclusion inferred from the premises $P_1, \ldots, P_n$.

There are several principles for performing a deduction:

- By the principle of direct proof we proceed from the premises $P_1, \ldots, P_n$ and try to derive a conclusion C.
- Using the principle of proof by contradiction $\neg C$ is assumed and we try to prove that this assumption together with the premises will lead to a contradiction; the principle is also called *reductio ad absurdum*. The premises and the assumption constitute an *inconsistent* set of clauses. If we can demonstrate a contradiction, we conclude that the assumption $\neg C$ is false and C consequently true.

Our premises (the world descriptions) are expressed in Horn form. The form is motivated by the ease with which deductions can be mechanized. We will construct the deductions using the principle of proof by contradiction. The inference rule we will use for the deductions is called *the resolution rule*. The resolution rule has two premises and one conclusion. The conclusion is called

the *resolvent* and the premises are called the *parents* of the resolvent. To apply the resolution rule is called to *resolve*.

The resolution rule for Horn clauses is

$$\frac{\begin{array}{rr} E \leftarrow B_1, \ldots, B_m & (57) \\ C \leftarrow D_1, \ldots, D_k, E, F_1, \ldots, F_n & (58) \end{array}}{C \leftarrow D_1, \ldots, D_k, B_1, \ldots, B_m, F_1, \ldots, F_n}$$

Clauses (57) and (58) are the parents and the clause below the line is the resolvent of the application of the resolution rule. We resolve between two clauses (57) and (58). We have a resolvent if there is an instance of an atomic formula E in the consequence of (57) and the same instance E in the condition of (58). The conditional part of the resolvent is formed by formulas from the conditional part of (57), i.e. $B_1, \ldots, B_k$ and formulas from the conditional part of (58) apart from E, i.e. $D_1, \ldots, D_k$ and $F_1, \ldots, F_n$. The consequence of the resolvent contains all consequences of (57) and (58) apart from E, i.e. C.

To apply the resolution rule as above, when both clauses have a consequential as well as a conditional part, is called to reason *middle-out*. To apply the resolution rule when one of the parent clauses lacks conditions means to reason *bottom-up*, i.e. we reason from a clause without conditions. Compare the construction of definitions from given basic concepts. We know that E is valid. The condition for C to be valid is that all of $D_1, \ldots, D_k, E, F_1, \ldots, F_n$ be true. Since E is true the condition for C can be reduced to $D_1, \ldots, D_k, F_1, \ldots, F_n$.

$$\frac{\begin{array}{rr} E \leftarrow & (59) \\ C \leftarrow D_1, \ldots, D_k, E, F_1, \ldots, F_n & (60) \end{array}}{C \leftarrow D_1, \ldots, D_k, F_1, \ldots, F_n}$$

The other way around is to apply the resolution rule *top-down*, i.e. to proceed from negated clauses. The clause $\leftarrow D_1, \ldots, D_k, E, F_1, \ldots, F_n$ means that not all the formulas $D_1, \ldots, D_k, E, F_1, \ldots, F_n$ are true.

$$\frac{\begin{array}{rr} E \leftarrow B_1, \ldots, B_m & (61) \\ \leftarrow D_1, \ldots, D_k, E, F_1, \ldots, F_n & (62) \end{array}}{\leftarrow D_1, \ldots, D_k, B_1, \ldots, B_m, F_1, \ldots, F_n}$$

We can reason like this:

E may be true or false.

- If E is false then
  all of $B_1, \ldots, B_m$ cannot be true;
  we can replace E by $B_1, \ldots, B_m$ in (62) without affecting the truth value.

- If E is true then, on the other hand,
  one of $D_1, \ldots, D_k, F_1, \ldots, F_n$ must be false;
  $B_1, \ldots, B_m$ can be either true or not;
  we can still replace E by $B_1, \ldots, B_m$ without changing the truth value.

The deduction is performed by negating the clause whose validity we wish to check. We then apply the resolution rule until we reach a contradiction, i.e. a Horn clause without conditions and without a consequent. When we obtain a contradiction we conclude that the assumption was correct according to the given premises since the deduction was performed under the principle of proof by contradiction.

Let us look at a few examples. First, as a very simple case, suppose that we want to derive Mother(Thor,Earth) from (14). The negation of the clause is ¬ Mother(Thor, Earth), i.e. in Horn form

$$\leftarrow \text{ Mother(Thor,Earth)} \tag{63}$$

The premise is Mother(Thor, Earth) ←. The assumption and the premise contradict each other immediately. We obtain the following deduction which contains only one step.

$$\begin{array}{rr} \text{Mother(Thor,Earth)} \leftarrow & (14) \\ \leftarrow \text{ Mother(Thor,Earth)} & (63) \\ \hline \leftarrow & \end{array}$$

We have resolved between two clauses (14) and (63). In the consequent of (14) and in the condition of (63) we have the atomic formula Mother(Thor, Earth). The conditional part of the resolvent is formed from the conditional part of (14) which is empty, and the conditional part of (63), apart from Mother(Thor, Earth). In this case the conditional part of the resolvent will be empty. The consequence part of the resolvent contains all the consequences in (14), except Mother(Thor, Earth), together with all the consequences of (63). Thus the consequence part is empty, too. We have performed a deduction of Mother(Thor,Earth) from the premise Mother(Thor,Earth).

2.1.2 Unification

We are already acquainted with the concepts instance and variant of a clause from Round 1. Let us turn to the use of these concepts in a deduction. Suppose that we want to derive the statement Norse_god(Odin) from the premises:

$$\begin{array}{ll} \text{Norse\_god(x)} \leftarrow \text{ Æs(x)} & (5) \\ \text{Æs(Odin)} \leftarrow & (22) \end{array}$$

The negation of Norse_god(Odin) is

$$\leftarrow \text{ Norse\_god(Odin)} \tag{64}$$

Let us apply the resolution rule bottom-up, i.e. starting from a clause with an empty conditional part.

$$\frac{\begin{array}{ll} \text{Æs(Odin)} \leftarrow & \\ & \text{Norse\_god(x)} \leftarrow \text{Æs(x)} \end{array}}{} \qquad \begin{array}{l} (22) \\ (5) \end{array}$$

We cannot apply the resolution rule directly, because Æs(x) and Æs(Odin) are not identical atomic formulas. The formulas have the same predicate names and the same arity. The difference is that in the first formula we have the constant Odin as argument whereas the other has the variable x. Norse_god(x) ← Æs(x) is a universally quantified formula, i.e. a formula which holds for all individuals, and we can create instances from it. When we instantiate x with Odin, we get Norse_god(Odin) ← Æs(Odin). We are now able to apply the resolution rule.

$$\frac{\begin{array}{ll} \text{Æs(Odin)} \leftarrow & \\ & \text{Norse\_god(Odin)} \leftarrow \text{Æs(Odin)} \end{array}}{\text{Norse\_god(Odin)} \leftarrow}$$

The resolvent contradicts the assumption (64) and the whole deduction will be as follows

$$\frac{\begin{array}{ll} \text{Æs(Odin)} \leftarrow & \\ & \text{Norse\_god(Odin)} \leftarrow \text{Æs(Odin)} \end{array}}{} \qquad \begin{array}{l} (22) \\ (5) \end{array}$$

$$\frac{\begin{array}{ll} \text{Norse\_god(Odin)} \leftarrow & \\ & \leftarrow \text{Norse\_god(Odin)} \end{array}}{\leftarrow} \qquad (64)$$

For reasons of space the resolvent is placed under the line, to the left, instead of exactly underneath, in derivations consisting of several steps. In the succeeding resolution step we then use the resolvent as a premise.

We have proven that Odin is a Norse god. In another example we have the following layout where we want to resolve the consequent in (16) and the condition Parent(x,parent) in (20).

$$\frac{\begin{array}{ll} \text{Parent(x,y)} \leftarrow & \text{Mother(x,y)} \\ & \text{Sibling(x,y)} \leftarrow \text{Parent(x,parent), Parent(y,parent)} \end{array}}{} \qquad \begin{array}{l} (16) \\ (20) \end{array}$$

Nor in this case can the resolution rule be applied directly. The difference between Parent(x,y) and Parent(x,parent) is the different variable names in the

second argument of the predicate. When we generate a variant of the clause (16), the problem is solved.

$$\frac{\begin{array}{l}\mathsf{Parent(x,parent)} \leftarrow \mathsf{Mother(x,parent)} \\ \qquad\qquad \mathsf{Sibling(x,y)} \leftarrow \mathsf{Parent(x,parent),\ Parent(y,parent)}\end{array}}{\mathsf{Sibling(x,y)} \leftarrow \mathsf{Mother(x,parent),\ Parent(y,parent)}}$$

Thus, we generate instances and variants to make two atomic formulas identical. The examples are special cases. Generally we may very well have to generate instances and variants of both formulas. Our goal is to make the formulas uniform which is called to *unify* the formulas. We have unified Æs(x) and Æs(Odin) by instantiating the variable x to Odin. We unified Parent(x,y) and Parent,(x,parent) by creating a variant. Both the formulas to be unified have the same predicate name and an equal number of arguments.

The general rules for unification are:
two predicates can be unified if

- they have the same predicate names and
- they have the same arity and
- the arguments can be unified

two terms can be unified if

- one of them is a variable or
- they are identical or
- both are structures and if
  - they have the same constructor and
  - they have the same arity and
  - the arguments can be unified

When unifying two terms a substitution is performed on both the expressions either by substituting a variable for individuals or structures (generate instances), or a variable name for another variable name (generate variants). In short, we exchange a variable name for a term. The variable and the term form a substitution pair. We call the substitution pairs we get in a deduction a *substitution.* A substitution $\{\langle x_1,t_1\rangle, \ldots,\langle x_m,t_m\rangle\}$ consists of substitution pairs in the form $\langle x_i,t \rangle$ where x_i denotes a variable and t_i a term. The substitution for unifying Æs(Odin) and Æs(x) will be $\{\langle x,Odin\rangle\}$.

The result of applying the substitution $\{\langle x,Odin\rangle\}$ to the expression Æs(x) is Æs(Odin), that is, we have substituted the variable x for the term Odin. Given an expression E and a substitution θ, we denote the result of applying the substitution to E as $E\theta$. $E\theta$ and E are identical with the exception that for every pair $\langle x_i, t_i\rangle$ which belongs to θ, if E contains x_i then each occurrence x_i is replaced by t_i in $E\theta$. A substitution unifies two expressions E_1 and E_2 if it makes them identical, i.e. $E_1\theta = E_2\theta$ (see Figures 2.1 and 2.2).

E_1	E_2	θ	$E_1\theta$
x	Odin	{⟨x,Odin⟩}	Odin
Thor	y	{⟨y,Thor⟩}	Thor
And(B,C)	And(B,C)	{}	And(B,C)
S(0)	x	{⟨x,S(0)⟩}	S(0)
And(x,A)	And(y,z)	{⟨x,y⟩,⟨z,A⟩}	And(y,A)
G(B,y)	G(x,G(A,B))	{⟨x,B⟩,⟨y,G(A,B)⟩}	G(B,G(A,B))

Figure 2.1: Examples of successful unifications.

Thor	Balder
Æs(Odin)	Norse_god(Odin)
And(A,x)	And(B,x)
G(x,y,z)	G(A,B)

Figure 2.2: Examples of expressions that cannot be unified.

To carry out the unification as neatly as possible, we make sure that the two parental clauses do not contain any variables with the same name so that the meaning of the variable name in the substitution becomes unambiguous. We exchange one variable name for another at each occurrence of the first variable name. These new variable names are used only during the deduction. We may call them *internal variable names* or *parameters* which have no meaning outside the deduction.

Suppose that we want to unify Parent(x,y) with Parent(y,parent). The variable y in the first expression has nothing whatever to do with the variable y in the second expression.

Thus, unification of these two expressions consists of two steps:

- to generate unique variable names and
- to find the substitution set for the two formulas.

We generate unique variable names by adding a number to the name. We may add the number 0 to every variable name in the first formula. Then we get Parent(x0, y0). Adding the figure 1 to every variable name in the second formula Parent(y1, parent1) makes the variable names in the two formulas distinct. We shall now look at the differences in the formulas to find the substitution set. Both formulas have the same predicate names and the same arity. One difference is that where x0 occurs in the first formula, y1 occurs in the second. The next difference appears in the second argument between y0 and parent1; otherwise the formulas are identical. The substitution {⟨y1,x0⟩,⟨parent1,y0⟩} unifies the formulas. When we unify two variables it does not matter which of the variables we put first in the substitution pair. We can apply the convention of always placing first the variable name with the highest figure added to it.

So, if the variable names are distinct in the formulas, we proceed in the following manner to find a substitution between two atomic formulas A and B[1]:

- Find the differences between the formulas. The difference may be expressed as a sequence of pairs {x,y} where x is the expression in the formula A and y the corresponding expression in B.
- Unification is possible if at least one of the expressions in each pair is a variable. Furthermore, none of the expressions may be a component of the other expression, i.e. x must not occur in y and vice versa. This is called "occur check". The occur check helps us to avoid generating infinite structures. Supposing we have the pair {x,S(x)}, for instance, we will be able to substitute x for S(x) an infinite number of times.
- Select one of the pairs in the difference set to be included in the substitution θ. The substitution pair we have chosen is applied to formula A and formula B and to the previous substitution pairs in the substitution.
- We repeat the procedure for the formulas $A\theta$ and $B\theta$ until there is no difference between the formulas, i.e. a unification is performed, or we discover through the difference set that the formulas cannot be unified.

Let us look at a slightly longer deduction. We would like to check whether Æs(Trud) is a correct conclusion from the following clauses.

$$\text{Æs(Odin)} \leftarrow \tag{22}$$
$$\text{Æs(x)} \leftarrow \text{Parent(x,y), Æs(y)} \tag{24}$$
$$\text{Parent(x,y)} \leftarrow \text{Father(x,y)} \tag{17}$$
$$\text{Father(Thor,Odin)} \leftarrow \tag{15}$$
$$\text{Father(Trud,Thor)} \leftarrow \tag{25}$$

The assumption is

$$\leftarrow \text{Æs(Trud)} \tag{65}$$

In the deduction below we show that clauses (15), (17), (22), (24), (25), and (65) are inconsistent. The resolution rule is applied bottom-up and middle-out. Instead of explicitly writing the unified forms of the parents of the resolvent, we mark every step with the substitution used.

$$\text{Æs(x0)} \leftarrow \text{Parent(x0,y0), Æs(y0)} \tag{24}$$
$$\text{Æs(Odin)} \leftarrow \tag{22}$$
$$\theta_1 = \{\langle \text{y0,Odin} \rangle\}$$
$$\text{Æs(x0)} \leftarrow \text{Parent(x0,Odin)}$$
$$\text{Parent(x1,y1)} \leftarrow \text{Father(x1,y1)} \tag{17}$$
$$\theta_2 = \{\langle \text{x1,x0} \rangle, \langle \text{y1,Odin} \rangle\}$$

[1] The algorithm was given by J.A. Robinson in *Logic: Form and Function*, Edinburgh University Press, 1979. A program that finds a substitution according to this algorithm is to be found in Round 6.

Æs(x0) ← Father(x0,Odin)
Father(Thor,Odin) ← (15)
$\theta_3 = \{\langle x0, Thor\rangle\}$

Æs(Thor) ←
Æs(x2) ← Parent(x2,y2), Æs(y2) (24)
$\theta_4 = \{\langle y2, Thor\rangle\}$

Æs(x2) ← Parent(x2,Thor)
Parent(x3,y3) ← Father(x3,y3) (17)
$\theta_5 = \{\langle x3, x2\rangle, \langle y3, Thor\rangle\}$

Æs(x2) ← Father(x2,Thor)
← Æs(Trud) (65)
$\theta_6 = \{\langle x2, Trud\rangle\}$

← Father(Trud,Thor)
Father(Trud,Thor) ← (25)

←

The desired conclusion can contain variables. The negation of the formula $\exists s\ \mathit{Sibling}(\mathit{Magni}, s)$ is $\neg\exists s\ \mathit{Sibling}(\mathit{Magni}, s)$ which can be reformulated to $\forall s\ \neg\mathit{Sibling}(\mathit{Magni}, s)$. In Horn form we get this assumption for a proof by contradiction:

← Sibling(Magni,s) (66)

The deduction is carried out top-down, i.e. starting from (66)

← Sibling(Magni,s0) (66)
Sibling(x1,y1) ← Parent(x1,parent1), Parent(y1,parent1) (20)
$\theta_1 = \{\langle x1, Magni\rangle, \langle y1, s0\rangle\}$

← Parent(Magni,parent1),Parent(s0,parent1)
Parent(x2,y2) ← Father(x2,y2) (17)
$\theta_2 = \{\langle x2, Magni\rangle, \langle y2, parent1\rangle\}$

← Father(Magni,parent1), Parent(s0,parent1)
Father(Magni,Thor)← (26)
$\theta_3 = \{\langle parent1, Thor\rangle\}$

← Parent(s0,Thor)
Parent(x3,y3) ← Father(x3,y3) (17)
$\theta_4 = \{\langle x3, s0\rangle, \langle y3, Thor\rangle\}$

$$\frac{\leftarrow \text{Father(s0,Thor)} \qquad \text{Father(Trud,Thor)} \leftarrow \quad (25)}{\theta_5 = \{\langle \text{s0,Trud} \rangle\}}$$

$$\leftarrow$$

The fact that we have demonstrated a contradiction implies that we have found an individual s for whom $Sibling(Magni, s)$ holds. We have found a counter-example of the clause $\forall s \neg Sibling(Magni, s)$, i.e. $\exists s\ Sibling(Magni, s)$ is valid. By applying the different substitutions in the deduction on the assumption, we can discover the value of the variables making the clause true.

Following the chain of substitutions in the deduction we can see that s0 in θ_5 has been unified with Trud. The combination of these substitutions gives us s = Trud. We have succeeded in refuting ← Sibling(Magni,s) on condition that the value of s is Trud. We can draw the conclusion that Sibling(Magni,Trud) is a true statement.

Let us study further examples of unification by looking at a deduction from the definition of Sum in the clauses (55) and (56). To carry out the deduction we also need the definition of the relation =.

$$\text{Sum(x,y,s)} \leftarrow \text{x} = 0,\ \text{s} = \text{y} \quad (55)$$

$$\text{Sum(x,y,s)} \leftarrow \text{x} = \text{S(nx)},\ \text{s} = \text{S(ns)},\ \text{Sum(nx,y,ns)} \quad (56)$$

$$\text{x} = \text{x} \leftarrow \quad (19)$$

Suppose that we want know if there exists a number x so that the relation Sum(x,S(S(0)),S(S(S(0)))) is valid. The assumption is

$$\leftarrow \text{Sum(x,S(S(0)),S(S(S(0))))} \quad (67)$$

and a deduction constructed bottom-up will look as follows

$$\leftarrow \text{Sum(x0,S(S(0)),S(S(S(0))))} \quad (67)$$

$$\text{Sum(x1,y1,s1)} \leftarrow \text{x1} = \text{S(nx1)},\ \text{s1} = \text{S(ns1)},\ \text{Sum(nx1,y1,ns1)} \quad (56)$$

$$\theta_1 = \{\langle \text{x1,x0} \rangle, \langle \text{y1,S(S(0))} \rangle, \langle \text{s1,S(S(S(0)))} \rangle\}$$

$$\leftarrow \text{x0} = \text{S(nx1)},\ \text{S(S(S(0)))} = \text{S(ns1)},\text{Sum(nx1,S(S(0)),ns1)}$$

$$\text{x2} = \text{x2} \leftarrow \quad (19)$$

$$\theta_2 = \{\langle \text{x2,S(nx1)} \rangle, \langle \text{x0,S(nx1)} \rangle\}$$

$$\leftarrow \text{S(S(S(0)))} = \text{S(ns1)},\ \text{Sum(nx1,S(S(0)),ns1)}$$

$$\text{x3} = \text{x3} \leftarrow \quad (19)$$

$$\theta_3 = \{\langle \text{x3,S(S(S(0)))} \rangle, \langle \text{ns1,S(S(0))} \rangle\}$$

$$\leftarrow \text{Sum(nx1,S(S(0)),S(S(0)))}$$

$$\text{Sum(x4,y4,s4)} \leftarrow \text{x4} = 0,\ \text{s4} = \text{y4} \quad (55)$$

$$\theta_4 = \{\langle \text{x4,nx1} \rangle, \langle \text{y4,S(S(0))} \rangle, \langle \text{s4,S(S(0))} \rangle\}$$

$$\frac{\leftarrow \mathsf{nx1} = 0,\ \mathsf{S(S(0))} = \mathsf{S(S(0))} \qquad \mathsf{x5} = \mathsf{x5} \leftarrow}{\theta_5 = \{\langle \mathsf{x5,0}\rangle, \langle \mathsf{nx1,0}\rangle\}} \tag{19}$$

$$\frac{\leftarrow \mathsf{S(S(0))} = \mathsf{S(S(0))} \qquad \mathsf{x6} = \mathsf{x6} \leftarrow}{\theta_6 = \{\langle \mathsf{x6,S(S(0))}\rangle\}} \tag{19}$$

$\leftarrow$

Let us find the substitution for the second resolution step, i.e. when we shall unify $\mathsf{x0} = \mathsf{S(nx1)}$ and $\mathsf{x2} = \mathsf{x2}$. The difference between the two formulas is the set $\{\{\mathsf{x2,x0}\},\{\mathsf{x2,S(nx1)}\}\}$. Each pair contains a variable so unification is not impossible. As the first pair in the substitution we may choose $\langle \mathsf{x2,x0}\rangle$. This substitution yields the new formulas $\mathsf{x0} = \mathsf{S(nx1)}$ and $\mathsf{x0}=\mathsf{x0}$. The difference between them is $\{\{\mathsf{x0,S(nx1)}\}\}$. Thus the next substitution pair will be $\langle \mathsf{x0,S(nx1)}\rangle$. This substitution gives two identical formulas $\mathsf{S(nx1)}= \mathsf{S(nx1)}$ and $\mathsf{S(nx1)}= \mathsf{S(nx1)}$. We apply the substitution pair to the substitution $\{\langle \mathsf{x2,x0}\rangle\}$ which yields $\{\langle \mathsf{x2,S(nx1)}\rangle\}$. The final substitution is $\{\langle \mathsf{x2,S(nx1)}\rangle, \langle \mathsf{x0,S(nx1)}\rangle\}$.

We may rewrite the program by moving the identities to the arguments of the predicate. The condition for (55) is that $\mathsf{x} = 0$ and that $\mathsf{s} = \mathsf{y}$. Instead of the argument x in (55) we may write 0 and instead of s we write y. Similar moves can be made for (56).

$$\mathsf{Sum(0,y,y)} \leftarrow \tag{68}$$
$$\mathsf{Sum(S(px),y,S(ps))} \leftarrow \mathsf{Sum(px,y,ps)} \tag{69}$$

The corresponding resolution proof for clauses (68) and (69) with (67) is:

$$\frac{\leftarrow \mathsf{Sum(x0,S(S(0)),S(S(S(0))))} \qquad \mathsf{Sum(S(nx1),y1,S(ns1))} \leftarrow \mathsf{Sum(nx1,y1,ns1)}}{\theta_1 = \{\langle \mathsf{x0,S(nx1)}\rangle, \langle \mathsf{y1,S(S(0))}\rangle, \langle \mathsf{ns1,S(S(0))}\rangle\}} \tag{69}$$

$$\frac{\leftarrow \mathsf{Sum(nx1,S(S(0)),S(S(0)))} \qquad \mathsf{Sum(0,y2,y2)} \leftarrow}{\theta_2 = \{\langle \mathsf{nx1,0}\rangle, \langle \mathsf{y2,S(S(0))}\rangle\}} \tag{68}$$

$\leftarrow$

θ_1 applied to the question Sum(x0, S(S(0)), S(S(S(0)))) yields Sum(S(nx1), S(S(0)), S(S(S(0)))). We have obtained part of the result. We notice that the element which satisfies the relation Sum shall be a constructed number. We have achieved a *partial result.* By applying θ_2 to Sum(S(nx1),S(S(0)),S(S(S(0)))) we arrive at the final result Sum(S(0),S(S(0)),S(S(S(0)))).

2.1.3 Search Space

A *search space* illustrates all possible ways of applying the inference rules to a set of clauses. The inference rule we shall use is the resolution rule. The

search space consists of *nodes* and, between the nodes, *branches*. The nodes correspond to clauses and the branches to an inference step. Every node in the space branches off to the resolvents we get on resolution between the clause and the program clauses. Let us look at the search space for the deduction of a sibling of Magni. We use the clauses

Parent(x,y) ← Father(x,y) (17)
Sibling(x,y) ← Parent(x,p), Parent(y,p) (20)
Father(Trud,Tor) ← (25)
Father(Magni,Tor) ← (26)
← Sibling(Magni,s) (66)

We start with the assumption (66). The only clause which has Sibling in the consequence is (20). The resolvent is

← Parent(Magni,p1), Parent(s0,p1)

Using this resolvent we can resolve in two different ways with clause (17), which results in a branching in the search space. The nodes of the tree always correspond to an application of the resolution rule. Thus we do not need to mark every node with the rule used since it is always the resolution rule.

A branch may be concluded in two ways.

i) We have discovered a contradiction and cannot perform further resolution steps. We mark the node with o ←. A branch that is ended by a contradiction is a completed deduction.

ii) We have derived a clause where we cannot proceed to resolve, this time because there is no clause in the program that can be used together with this clause for a resolution step. We mark the node with •. We have failed to find a contradiction and hence have not found a deduction in this branch.

The whole search space for our example, with the resolution rule applied top-down, is displayed in Figure 2.3. After each resolution step we write only the resolvent. The formula at each node in a search space constructed top-down is a Horn clause without consequence. On the uppermost level, level 0, in the space we have the assumption for proof by contradiction; the assumption is thus the root of the tree. The rest of the nodes in the tree are successive resolvents.

The number of nodes in the search space is finite, i.e. we have a *finite search space*. The total search space in this case consists of 12 branches that are all valid deductions, i.e. are ended by a contradiction. There are only two different answers, however, for which the statement is true but every answer can be deduced in six different ways.

To traverse the search space we can use different *search strategies*.

- *Breadth-first* strategy traverses the trees level by level. First all nodes on level 0, i.e. the top level of the search space are explored. On level 1 we have the roots of the subtrees of level 0. They are then investigated before

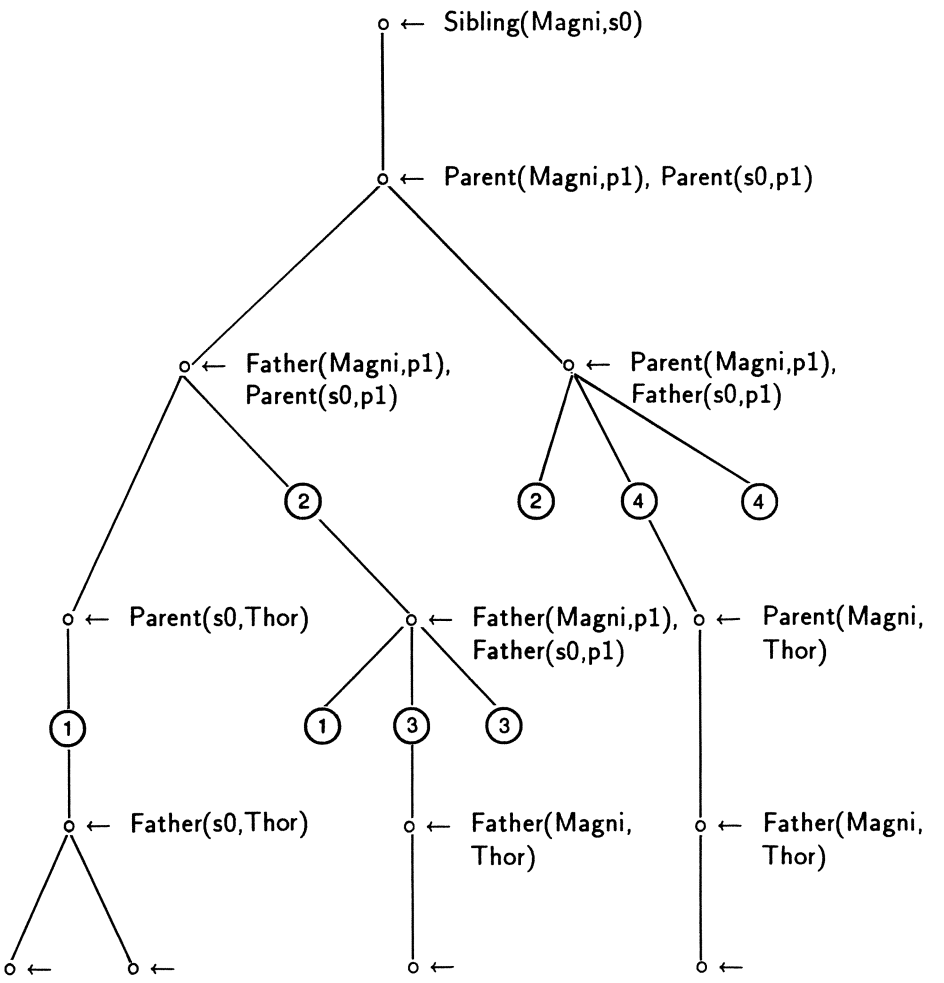

Figure 2.3: Total top-down search space for Sibling(Magni,s). The marks in the search space names a subtree that will be repeated once more in the search space.

we continue to the nodes of their subtrees, etc. On level 0 we find the node ← Sibling(Magni,s0), on level 1 we have the node ← Parent(Magni,p1), Parent(s0,p1).

- *Depth-first* is the strategy where we follow a branch of the search space until its end and then *backtrack*, i.e. we go back along the branch until it branches

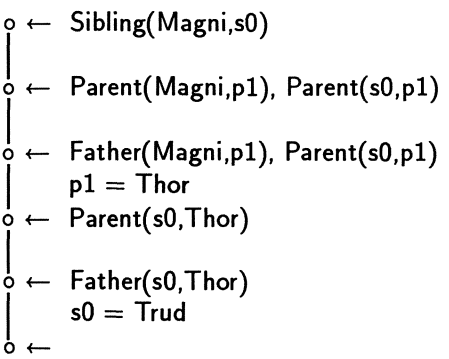

Figure 2.4: The first branch in the search space for Sibling(Magni,s).

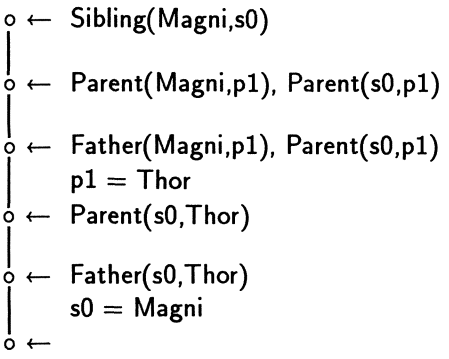

Figure 2.5: The second branch in the search space for Sibling(Magni,s).

off next and continue down the nearest branch to the right. A branch point is a place where the resolution rule can be applied in several different ways and when searching depth-first we go through the alternatives completely one after the other.

The first branch, see Figure 2.4, found by depth-first search yields s = Trud. The second branch in Figure 2.5 differs only on the second lowest step. We have backtracked one step because ← Father(s0,Thor) can be resolved both with (25) as in the first branch and (26) which gives the second deduction. Thus we have one more deduction, the solution being s = Magni.

All alternatives at the node ← Father(y0,Thor) are now exhausted. To find more solutions we have to backtrack further. The nearest choice point is the clause ← Father(Magni,p1), Parent(s0,p1). We explored this case when we

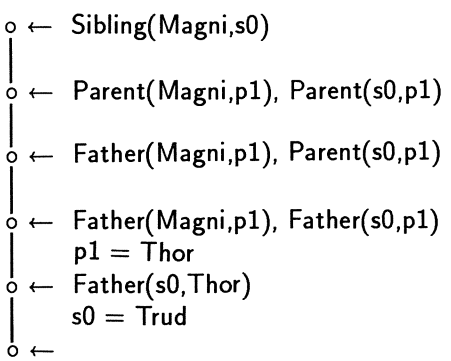

Figure 2.6: The third branch in the search space for Sibling(Magni,s).

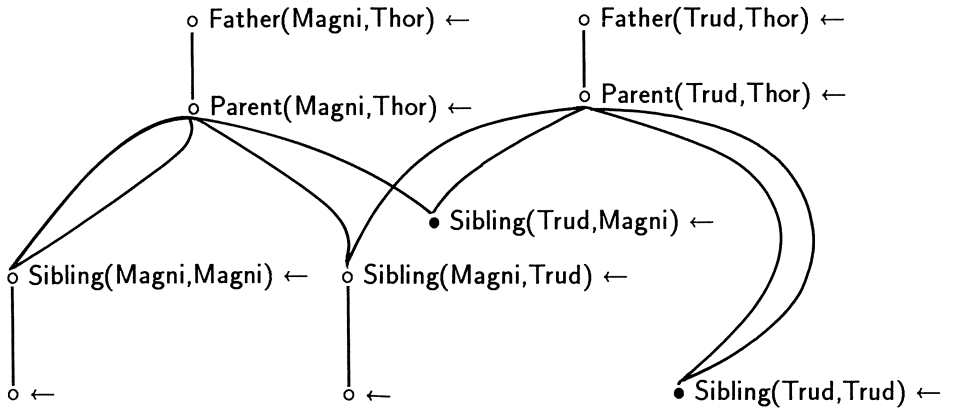

Figure 2.7: Total bottom-up search space for Sibling(Magni,s).

resolved with (26). The alternative is to use (17). The third branch appears in Figure 2.6. The remaining branches can be constructed similarly.

We have looked at the search space constructed top-down. We can also apply the resolution rule bottom-up and construct the search space bottom-up, see Figure 2.7. In this case every node in the search space is an unconditional Horn clause. We start from (26) and (25). Both these statements may be resolved with (17). The resolvent is Parent(Magni,Thor) ← and Parent(Trud,Thor) ← respectively. The two resolvents, in their turn, can be resolved against (20) in four different ways. Two of the resulting resolvents render a contradiction with (66). The other two cannot be resolved further. The search space in Figure 2.7 also contains both of the solutions, Sibling(Magni,Magni) and Sibling(Magni,Trud).

Let us also look at the search space for the second of the two deductions given in the previous section, that is, to derive Æs(Trud) from the premises

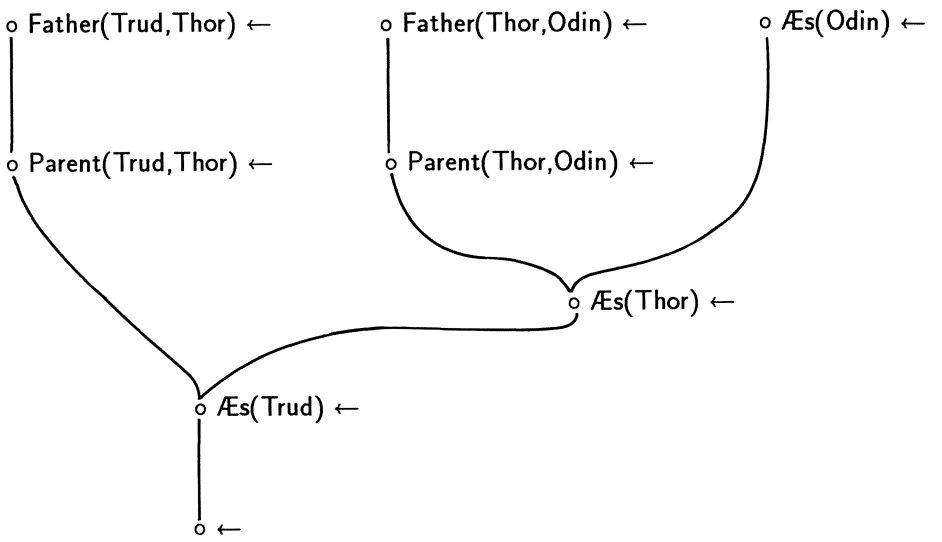

Figure 2.8: Total bottom-up search space for Æs(Trud).

Parent(x,y) ← Father(x,y)	(17)
Æs(Odin) ←	(22)
Æs(x) ← Parent(x,y), Æs(y)	(24)
Father(Thor,Odin) ←	(15)
Father(Trud,Thor) ←	(25)

The search space constructed bottom-up is finite, see Figure 2.8. The search space constructed top-down in Figure 2.9, on the other hand, is not finite. Hence we have an *infinite search space.* A search space is infinite when at least one branch in the space is infinite. The branch on the far right in the search space in Figure 2.9 is not finite.

By depth-first traversal of the search space we first arrive at the deduction in the branch at the far left in Figure 2.9. Backtracking to find another solution, i.e. resolving between ← Father(Thor,y1), Æs(y1), and (24), we obtain the resolvent ← Father(Thor,y1), Parent(y1,y2), Æs(y2), which will not lead to a contradiction. By a depth-first traversal of the tree we can select an infinite branch and therefore not find the solution. To guarantee that we do find a deduction in a search space the search strategy must avoid the infinite branches in the search space.

We will take one more example. Let us construct search spaces top-down and bottom-up for these clauses

Triad(0) ←	(50)
Triad(S(S(S(prec_num)))) ← Triad(prec_num)	(51)

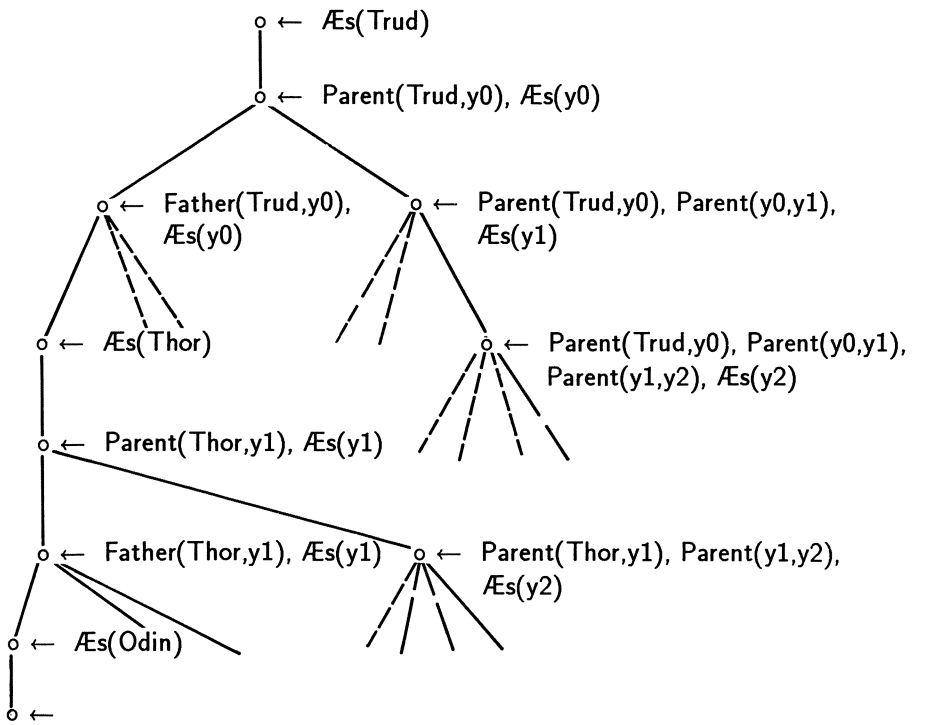

Figure 2.9: Total top-down search space for Æs(Trud).

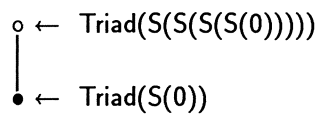

Figure 2.10: Top-down search space for Triad(S(S(S(S(0))))).

$$\leftarrow \text{Triad}(S(S(S(S(0)))))$$

The search space constructed top-down in Figure 2.10 is small. Constructed bottom-up, on the other hand, the search space is not finite, see Figure 2.11.

The search spaces can be of great volume. We can reduce the displayed top-down search space if we let the nodes branch only according to the alternative clauses that can be used to resolve against a selected literal in the question. In Figure 2.12 the selected atoms are underlined.

2.1.4 Proof Trees

Another way of illustrating the possibilities of deducing a clause is the *AND-OR-tree*. In an AND-OR-tree the root of the tree is the conclusion to be proven.

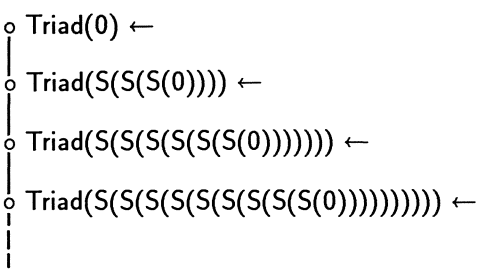

Figure 2.11: Bottom-up search space for Triad(S(S(S(S(0))))).

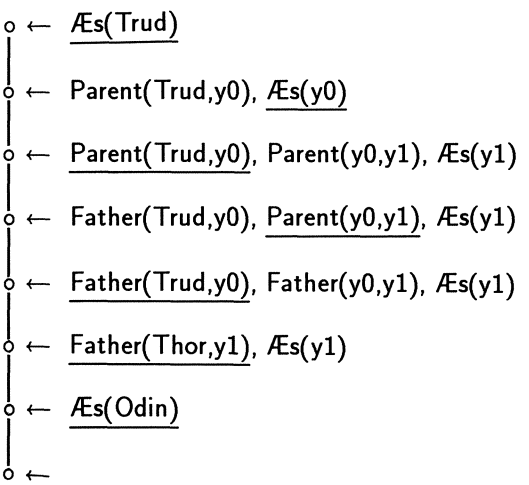

Figure 2.12: Partial top-down search space for Æs(Trud).

The nodes can be either AND-nodes or OR-nodes.

We have an AND-node when the root of a tree is a formula A and A is described using $B_1, \ldots, B_n$, that is,

$$A \leftarrow B_1, \ldots, B_n$$

For A to be valid all of B_1 to B_n have to be valid. The AND-node is marked by an arc connecting the branches, see Figure 2.13.

If there are alternative ways of showing that a formula is valid we draw an OR-node in the tree. This situation arises when we have several clauses defining A. For instance (see Figure 2.14)

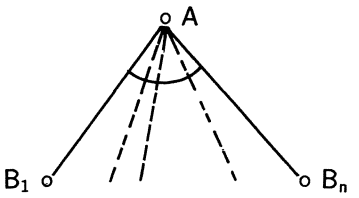

Figure 2.13: AND-node.

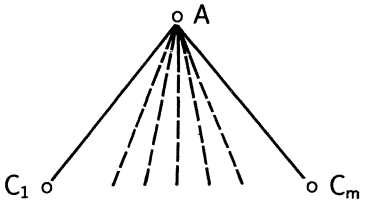

Figure 2.14: OR-node.

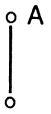

Figure 2.15: A leaf in an AND-OR-tree.

$$\begin{aligned} A &\leftarrow C_1 \\ &\ldots \\ A &\leftarrow C_m \end{aligned}$$

If any of $C_1, \ldots, C_m$ contains several conditions we have an OR-node in which an OR-branch consists of several AND-branches. A leaf in the tree corresponds to a statement without conditions, $A \leftarrow$, see Figure 2.15.

In Figure 2.16 we have our now familiar example of trying to find a sibling of Magni.

2.2 Prolog Proof Procedure

2.2.1 Derivations in Prolog

So far, we have performed deductions for clauses in Horn form generally, and used bottom-up, middle-out, as well as top-down construction. But in a Prolog system we naturally want to construct the deductions mechanically. The mechanization is simplified if we always use a predefined strategy when facing alternatives.

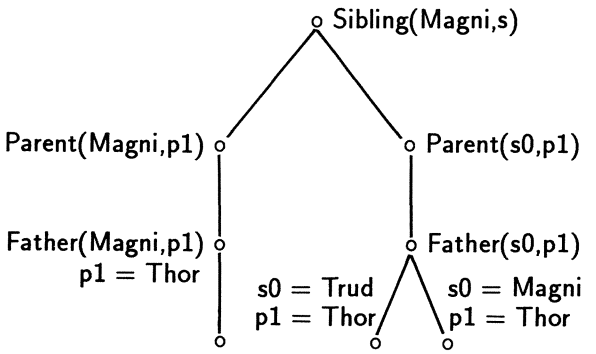

Figure 2.16: AND-OR-tree for the solution to Sibling(Magni,s).

When we shall perform a deduction we have several choices to make:

- the deduction can be constructed top-down or bottom-up
- the search space can be traversed depth-first or breadth-first

In Prolog systems the following choices have generally been made:

⋆ the deduction is constructed top-down

⋆ the search space is traversed through depth-first search

The decision to execute the deduction top-down means that we start the deduction from a Horn clause in the form

$$\leftarrow B_1, \ldots, B_j$$

When we apply the resolution rule we have two more choices to make:

- we have to select one of the atomic formulas $B_1, \ldots, B_j$
- we have to select a clause in the program

In Prolog systems the following choices have generally been made:

⋆ the atomic formulas are selected from left to right

⋆ clauses are tried in the order they are given in the program, i.e. textual order.

We have seen before that for a proof by contradiction we assume the negation of the conclusion. We may regard the assumption as a *query* whether the conclusion is valid. By performing a deduction from the assumption we receive an answer to the question. This can also be called an evaluation of the question. The answer may be *no* and thus the deduction has failed. When a deduction has succeeded according to Prolog's strategy, we receive the answer *yes* if the query was ground, i.e. without variables. We simply test if the question holds for the

individuals given. If the question contained variables, i.e. we have asked for individuals or structures that satisfy the relations of the question, the answer of a successful deduction will be the values which the variables have been unified with during the deduction. In most Prolog implementations unification works in the same manner as described in previous sections, with the exception that there is no occur check. So, if we are not careful, we may generate infinite structures.

The resolution proof for ← Siblings(Magni,s) (66) in the previous section was executed with the same search order as our Prolog system uses. An interaction with a Prolog system may look like this:

← Sibling(Magni,s)	A question to the Prolog system
s = Trud	s has been unified with Trud
← Sibling(Magni,Trud)	A question to the Prolog system
yes	The deduction was successful
← Sibling(Magni,Thor)	A question to the Prolog system
no	The deduction failed

2.2.2 Proof Trace

A branch in a search space corresponds to a *trace* of the different steps in the deduction.

Let us illustrate the working of Prolog by further examples. The various gods were worshipped for different reasons. Odin was worshipped as the god of wisdom and Thor as the god of thunder. Suppose that we want to define a relation between a mythical god and the sphere of which that god was held to be the protector. We may define the relation Protector with two arguments: the name of the god and his sphere. But if we extend the world to reach outside the Norse mythology, this relation is not very suitable. Let us include deities from Greek and Roman mythology. We can expand the relation Protector to be a relation between three arguments, where the first is the name of the pantheon of the god in question, the second is the name of the god, while the third is the god's sphere of protection.

Protector(Norse,Hel,Death) ← (70)
Protector(Norse,Freya,Love) ← (71)
Protector(Norse,Ægir,Sea) ← (72)
Protector(Norse,Odin,Wisdom) ← (73)
Protector(Norse,Tiw,Victory) ← (74)

Protector(Greek,Aphrodite,Love) ← (75)
Protector(Greek,Ares,War) ← (76)
Protector(Greek,Athena,Wisdom) ← (77)
Protector(Greek,Eirene,Peace) ← (78)

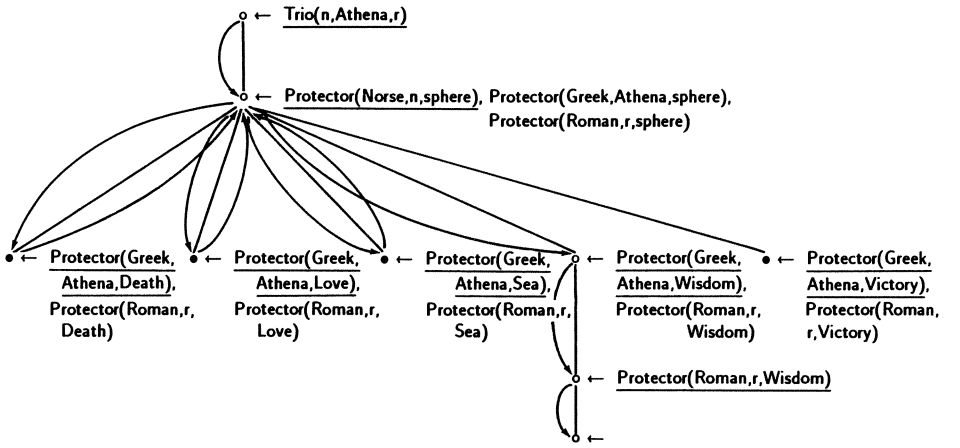

Figure 2.17: Prolog search-space for Trio(n,Athena,r).

$$\text{Protector(Greek,Poseidon,Sea)} \leftarrow \tag{79}$$

$$\begin{aligned}
&\text{Protector(Roman,Venus,Love)} \leftarrow && (80)\\
&\text{Protector(Roman,Diana,Hunting)} \leftarrow && (81)\\
&\text{Protector(Roman,Minerva,Wisdom)} \leftarrow && (82)\\
&\text{Protector(Roman,Pax,Peace)} \leftarrow && (83)\\
&\text{Protector(Roman,Vulcanus,Smithery)} \leftarrow && (84)
\end{aligned}$$

We can define a relation Trio to hold for three deities, one from Norse, one from Greek, and one from Roman mythology, in which the deities correspond to each other if they protect the same sphere.

$$\begin{aligned}
\text{Trio(n,g,r)} \leftarrow\ & && (85)\\
&\text{Protector(Norse,n,sphere)},\\
&\text{Protector(Greek,g,sphere)},\\
&\text{Protector(Roman,r,sphere)}
\end{aligned}$$

Suppose that we want to find out what deities in the Norse and the Roman pantheons correspond to Athena in the Greek pantheon. The intended interpretation of the relation Trio places the name of the Greek deity as the second argument. We want to check if there are values on n and r to satisfy the statement Trio(n,Athena,r), i.e. if there is an answer to the question. Let us first construct a Prolog search space for the question, see Figure 2.17. The arrows indicate the search strategy.

We will take a look at a trace of the deduction. Let us use a letter in addition to the numbering to mark alternatives, i.e. first we use "a", then "b", etc. We may be forced to backtrack if we arrive at a dead-end in the search space where we cannot continue the deduction. When we then create a new resolvent we

give the step the same number but assign it the next letter of the alphabet.

(1a) ← Trio(n,Athena,r)

From the definition of Trio we get a new question which contains three subquestions.

(2a) ← Protector(Norse,n,sphere), Protector(Greek,Athena,sphere), Protector(Roman,r,sphere)

The clauses for the relation Protector are tried in order of appearance. The first clause for which the consequence can be unified with the subquestion Protector(Norse,n,sphere) is selected. Two relations can be unified when they have the same relational name, the same arity, and the arguments can be unified. The relations can be unified if they can be made uniform, i.e. when their form or pattern can be matched. Protector(Norse,n,sphere) and clause (70) have the same pattern. Norse can be unified with Norse and the variables n and r can be unified with Hel and Death, respectively. We say that Protector(Norse,n,sphere) and (70) *match.*

n = Hel, sphere = Death

(3a) ← Protector(Greek,Athena,Death), Protector(Roman,r,Death)

But we cannot find a clause to match Protector(Greek,Athena,Death). We find ourselves in an unsuccessful branch of the search space. We must backtrack to the nearest choice point. There are several ways to resolve from Protector(Norse,n,sphere). But of these alternatives we have already used (70) which did not give a contradiction for the whole of the question. The remaining alternatives are (71) – (74) since the first argument has to be Norse to match. Matching with (71) yields:

n = Freya, sphere = Love

(3b) ← Protector(Greek,Athena,Love), Protector(Roman,r,Love)

We have to backtrack several steps and get new values for the variables n and sphere.

n = Ægir, sphere = Sea

(3c) ← Protector(Greek,Athena,Sea), Protector(Roman,r,Sea)

n = Odin, sphere = Wisdom

(3d) ← Protector(Greek,Athena,Wisdom), Protector(Roman,r,Wisdom)

We finally manage to find a match to the first part of the question which leads to an answer to the entire question, as we see below.

(4d) ← Protector(Roman,r,Wisdom)

r = Minerva

(5d) ←

Hence, the deities corresponding to Athena are the unifications we have had for n and r, i.e. Odin and Minerva.

We can examine one more deduction in which the third argument to the relation Trio is given instead. Thus we will find the Norse and Greek correspondences to the Roman goddess Minerva.

(1a) ← Trio(n,g,Minerva)

(2a) ← Protector(Norse,n,sphere), Protector(Greek,g,sphere), Protector(Roman,Minerva,sphere)

n = Hel, sphere = Death

(3a) ← Protector(Greek,g,Death), Protector(Roman,Minerva,Death)

We are not able to find any Greek god protecting death among our clauses and have to backtrack.

n = Freya, sphere = Love

(3b) ← Protector(Greek,g,Love), Protector(Roman,Minerva,Love)

g = Aphrodite

(4b) ← Protector(Roman,Minerva,Love)

Minerva is not the goddess of love, so we shall have to backtrack. There is no goddess other than Aphrodite who is the goddess of love so we shall have to go back for a new Norse goddess. When backtracking we lose all information on what happened between the spot where we failed to carry on the deduction and the nearest node, in this case nodes (4b) and (2a). The unifications made in between are thus no longer there and we unify n and sphere with new values after backtracking.

n = Ægir, sphere = Sea

(3c) ← Protector(Greek,g,Sea), Protector(Roman,Minerva,Sea)

g = Poseidon

(4c) ← Protector(Roman,Minerva,Sea)

This question fails so we go back again and get a new unification for n and sphere.

n = Odin, sphere = Wisdom

(3d) ← Protector(Greek,g,Wisdom), Protector(Roman,Minerva,Wisdom)

g = Athena

(4d) ← Protector(Roman,Minerva,Wisdom)

(5d) ←

The answer to the question is n=Odin and g=Athena.

2.2.3 Alternative Answers

There may be several ways to refute an assumption, i.e. there may be more than one branch in the search space that will lead to a contradiction. The question ← Sibling(Magni,s) received the answer s = Trud but if we want further answers we are able to force one more deduction. We can also get the answer s = Magni as we saw in the search spaces we constructed previously.

The relation Sum as defined in (68) and (69) can produce several alternative answers to the question Sum(n1,n2,S(S(S(0))))

$$\mathsf{Sum(0,y,y)} \leftarrow \tag{68}$$
$$\mathsf{Sum(S(nx),y,S(ns))} \leftarrow \mathsf{Sum(nx,y,ns)} \tag{69}$$

Depending on the arguments we leave uninstantiated, we can view the relation Sum in different ways. If we give values to the first and second arguments, i.e. Sum(S(0),S(S(0)),sum), the answer will be the sum of the two. If we give the value of the third argument and one of the first two arguments, i.e. Sum(S(0),difference,S(S(0))), the answer will be the value that is the difference between the two given values, i.e. the result of the subtraction between the third argument and the second. If we give a value only for the third argument the answer will be the different pairs of numbers into which the number can be divided. Let us investigate the deductions if we give the number S(S(S(0))) , i.e. 3, as third argument to Sum.

(1a) ← Sum(n1,n2,S(S(S(0))))
n1 = 0, n2 = S(S(S(0)))
(2a) ←

The answer is n1=0 and n2=S(S(S(0))). Let us search for alternative answers by backtracking. We can force the Prolog system to go back when a deduction has succeeded by writing ";" to indicate that we want more solutions.

```
← Sum(n1,n2,S(S(S(0))))
n1 = 0
n2 = S(S(S(0)));        ";" to force Prolog
                        to backtrack
```

We can also unify (1a) with (69):

n1 = S(nx1), y1 = n2
(2b) ← Sum(nx1,n2,S(S(0)))
nx1 = 0, n2 = S(S(0))
(3b) ←

We gained a new answer n1 = S(0) and n2 = S(S(0)), but we backtrack again for more answers.

nx1 = S(nx2), y2 = n2

n1	n2	Sum(n1,n2,S(S(S(0))))
0	S(S(S(0)))	Sum(0,S(S(S(0))),S(S(S(0))))
S(0)	S(S(0))	Sum(S(0),S(S(0)),S(S(S(0))))
S(S(0))	S(0)	Sum(S(S(0)),S(0),S(S(S(0))))
S(S(S(0)))	0	Sum(S(S(S(0))),0,S(S(S(0))))

Figure 2.18: Alternative answers to Sum(n1,n2,S(S(S(0)))).

(3c) $\leftarrow$ Sum(nx2,n2,S(0))
nx2 = 0, n2 = S(0)
(4c) $\leftarrow$

This time the answer is n1 = S(S(0)) and n2 = S(0). We will try again:

nx2 = S(nx3), y3 = n2
(4d) $\leftarrow$ Sum(nx3,n2,0)
nx3 = 0, n2 = 0

The last answer is n1 = S(S(S(0))) and n2 = 0. We have now exhausted the whole search space. In Figure 2.18 are all the possible values of n1 and n2 to make the Sum(n1,n2,S(S(S(0)))) valid.

2.2.4 Incomplete Proof Strategy

When Prolog selects a clause against which to resolve an assumption the clauses in the program are tried in the order in which they are given. The first clause whose consequence can be unified with the chosen subquestion is selected. This selection makes the deduction dependent on the order in which we state the clauses in the program. The deduction is also dependent on the order in which the conditions in the conditional part of a clause are written. Let us look at the definition of the Æs property. The definition as it appeared in Section 1.1 on World Descriptions is:

Æs(Odin) $\leftarrow$ (22)
Æs(x) $\leftarrow$ Parent(x,y), Æs(y) (24)

The clause Æs(Thor) can be derived in the following manner:

(1a) $\leftarrow$ Æs(Thor)
(2a) $\leftarrow$ Parent(Thor,y), Æs(y)
(3a) $\leftarrow$ Father(Thor,y), Æs(y)
y = Odin
(4a) $\leftarrow$ Æs(Odin)
(5a) $\leftarrow$

If we change the order of the clauses in the definition of Æs and the order of the conditions in clause (24) we cannot derive Æs(Thor) although this is a valid conclusion. Hence, Prolog's proof strategy is not *complete*, i.e. we are not guaranteed to be able to derive a clause although it is a valid conclusion.

Æs(x) ← Æs(y), Parent(x,y) (24)
Æs(Odin) ← (22)

The deduction of Æs(Thor) will be infinite.

(1a) ← Æs(Thor)
(2a) ← Æs(y0), Parent(Thor,y0)
(3a) ← Æs(y1), Parent(y0,y1), Parent(Thor,y0)
(4a) ← Æs(y2), Parent(y1,y2), Parent(y0,y1), Parent(Thor,y0)
(5a) ← Æs(y3), Parent(y3,y2), Parent(y1,y2), Parent(y0,y1), ..., etc

But if we simply change the order of the clauses in the definition of Æs, Prolog will succeed in finding a deduction of Æs(Thor) after backtracking.

Æs(x) ← Parent(x,y), Æs(y) (24)
Æs(Odin) ← (22)

The clause Æs(Thor) can be derived in the following manner:

(1a) ← Æs(Thor)
(2a) ← Parent(Thor,y), Æs(y)
(3a) ← Father(Thor,y), Æs(y)
y = Odin
(4a) ← Æs(Odin)
(5a) ← Parent(Odin,y1), Æs(y1)
(6a) ← Father(Odin,y1), Æs(y1)

As we have not given any clause telling who is the father of Odin, we fail. But on backtracking, on the other hand, we get an immediate match with (22).

(5b) ←

2.3 Predefined Relations

We will say more about language forms common in Prolog systems and relations that are usually *predefined.*

A definition of a relation may consist of several Horn clauses. If any of these clauses have the same consequence, Prolog permits us to use an abbreviated way of writing. The following two expressions are equivalent

$$(A \leftarrow B) \wedge (A \leftarrow C)$$
$$A \leftarrow B \vee C$$

In Prolog we use ";" to signify a disjunction. Thus we write

$$\mathsf{A} \leftarrow \mathsf{B} \,;\, \mathsf{C}$$

We do not have to repeat the consequence. If the clauses have the conditions in common, as well as the consequence, they, too, have only to be written once.

We can illustrate this by an example of a program which defines the relation paternal grandparent. A person y is paternal grandparent of the person x if x has a father z and y is mother or father of z.

First we define the relation in standard form.

$$\begin{aligned}\forall x \forall y \,(&\mathit{Paternal\_grandparent}(x, y) \leftrightarrow \\ &\exists z \,(\mathit{Father}(x, z) \\ &\quad \wedge (\mathit{Mother}(z, y) \vee \mathit{Father}(z, y))))\end{aligned}$$

Now we can define the relation in a Prolog program consisting of Horn clauses.

$$\mathsf{Paternal\_grandparent(x,y)} \leftarrow \mathsf{Father(x,z),Mother(z,y)} \qquad (86)$$
$$\mathsf{Paternal\_grandparent(x,y)} \leftarrow \mathsf{Father(x,z),Father(z,y)} \qquad (87)$$

Prolog permits us to join (86) and (87) in one clause. The Prolog program will then be

$$\mathsf{Paternal\_grandparent(x,y)} \leftarrow \mathsf{Father(x,z),(Mother(z,y);Father(z,y))} \qquad (88)$$

The execution of the question $\leftarrow$ Paternal_grandparent(x,Odin) will be as follows if we use clauses (86) and (87) and clauses (15), (25), and (26) for the relation Father.

(1a) $\leftarrow$ Paternal_grandparent(x,Odin)

(2a) $\leftarrow$ Father(x,z), Mother(z,Odin)

x = Thor, z = Odin

(3a) $\leftarrow$ Mother(Odin,Odin)

x = Trud, z = Thor

(3b) $\leftarrow$ Mother(Thor,Odin)

x = Magni, z = Thor

(3c) $\leftarrow$ Mother(Magni,Odin)

We have to backtrack and use the second clause (87) in the definition of Paternal_grandparent.

(2b) $\leftarrow$ Father(x,z), Father(z,Odin)

x = Thor, z = Odin

(3b) $\leftarrow$ Father(Odin,Odin)

x = Trud, z = Thor

(3c) ← Father(Thor,Odin)

(4c) ←

If we use definition (88) the execution will have the following appearance instead:

(1a) ← Paternal_grandparent(x,Odin)

(2a) ← Father(x,z), Mother(z,Odin)

x = Thor, z = Odin

(3a) ← Mother(Odin,Odin)

We will now backtrack to the latest alternative. We gave Father(z,y) as an alternative to Mother(z,y) with the construction ";". Thus the execution continues with the question Father(Odin,Odin).

(3b) ← Father(Odin,Odin)

This question fails and we will have to search for alternative solutions to Father(x,z).

x = Trud, z = Thor

(3c) ← Mother(Thor,Odin)

(3d) ← Father(Thor,Odin)

(4d) ←

Primarily we look for alternative evaluations within the disjunction, both Mother(z,y) and Father(z,y) of the person z are checked before backtracking to find a new instantiation of z in Father(x,z).

"," binds stronger than ";". To change the order of priority we can use parentheses. If we leave out the parentheses in (88), Prolog will interpret this clause in the same way as the system will interpret the following Horn clauses.

Paternal_grandparent(x,y) ← Father(x,z), Mother(z,y) (89)

Paternal_grandparent(x,y) ← Father(z,y) (90)

Clause (90) is not correct since the condition Father(x,z) is missing. Neither will the variable x be instantiated in (90) unless it is already instantiated in the question, since it appears only in the consequent of the clause. Clause (89), on the other hand, corresponds to (86) but the entire definition of Paternal_grandparent as given in (89) and (90) is not a correct definition of the concept paternal grandparent.

The conditions in a Prolog program can thus be joined by ";" and ",". A question can also consist of several parts connected by ",", ";", or both.

We will introduce relations for *comparison* and relations for *arithmetic*. Arithmetical relations and relations for comparison are often already predefined in a Prolog system. Predefined predicates may also be called *built-in predicates*.

We have a built-in relation x = y that decides whether the two arguments can be unified or not. When at least one of the arguments is uninstantiated, it is unified with the second argument, if the patterns are similar, and the relation will be true. The built-in predicates for comparisons take numbers as arguments and to make the comparisons feasible both the arguments in the relations have to be instantiated to non-variables. These relations can be used only for testing whether the relation is satisfied or not, between two given values, and so cannot give values as the answer.

Relations for comparison

$x < y$	x is less than y
$x > y$	x is greater than y
$x \leq y$	x is less than or equal to y
$x \geq y$	x is greater than or equal to y
$x \neq y$	x cannot be unified with y

In our Prolog system there is a built-in predicate Value which has an arithmetic expression as its first argument and the numerical value of this expression as its second argument, i.e. result is the value we receive on evaluation. The relation Value can be used to test whether the relation is true when both arguments are instantiated, on one hand, and if only the first argument is instantiated, on the other, to receive the computed value of the expression as the answer.

```
Value(expression,result)
```

An arithmetic expression is built up by arithmetic relations, variables, and numbers. Arithmetic expressions have to be instantiated before evaluation.

Arithmetic operators

x + y	the sum of x and y
x − y	the difference between x and y
x * y	the product of x and y
x / y	the quotient of x and y
x Mod y	the remainder in an integer division of x by y

Our Prolog program for Sum_of_triads, (47) and (48), which is a relation between a number and the sum of all numbers divisible by three that are less than the first number, will be shorter if we use the built-in predicates for arithmetic and represent our numbers by our Arabic numerals instead of by the constructor S. Since some of the predefined relations can only be used when some of their arguments are instantiated, the order between the conditions is important in a Prolog clause. It is not sufficient that all conditions are there — we also have to consider Prolog's order of execution. For instance, the condition Value(x+ys,s) must be placed so that we can be certain that both x and ys are instantiated before the relation is evaluated.

```
Sum_of_triads(0,0) ←                                          (91)
Sum_of_triads(x,s) ← x > 0, Triad(x),                         (92)
        Value(x-1,y), Sum_of_triads(y,ys),
        Value(x+ys,s)
Sum_of_triads(x,s) ← x > 0, NonTriad(x),                      (93)
        Value(x-1,y), Sum_of_triads(y,s)
```

In clauses (47) and (48) the base case and the recursive case were separated by the structure of the first argument. Clauses (91) - (93) all match a question about Sum_of_triads where the first argument is zero. The relation is defined only for positive integers, hence we must complement the recursive case by the condition $x = 0$ to avoid getting negative numbers when backtracking.

```
Triad(n) ←                                                    (94)
        Value(n Mod 3,rem), rem = 0
NonTriad(n) ←                                                 (95)
        Value(n Mod 3,rem), rem ≠ 0
```

We can thus replace the relation Sum by the built-in relation Value. A trace of the question ← Sum_of_triads(6,s) is given in Appendix B.1.

Let us finally look at a few more predefined relations which may occur in the Prolog system apart from the arithmetic relations. There are two predefined relations True and False, of which the former is always true and the latter is always false. The relation True can be used together with the abbreviated way of writing clauses using ";". Several of the clauses have mutual conditions, i.e.

```
A ← B,C,D
A ← B,C,E,F
A ← B,C
```

Conditions B and C are common to the three clauses. The last clause has no further conditions apart from the mutual conditions. Using the abbreviation we can write

```
A ← B,C, (D; E,F; True)
```

Appendix D contains a compilation of the most common predefined relations and control constructions.

2.4 An Example

We shall look at an example where backtracking is used to make the solution very simple. We have a chess board and one chess piece, a knight. The knight shall go round the board and each square must be visited only once.

In the figure we can see what moves a knight is permitted to make if placed in the square marked K.

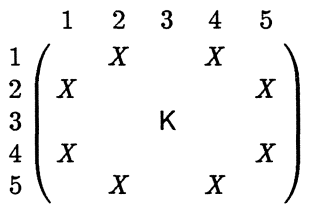

Figure 2.19: The knight's permitted moves

Let us define the relation Knights_tour. One argument shall be the itinerary and another a number stating the length of the itinerary. We also need to know the size of the board n to be able to make the condition that the knight must not go outside the board. A square may be represented as a pair S(row,col) where row denotes the number of the row and col the number of the column on the board. The route may be represented as a structure consisting of the squares the knight has visited. If a knight has visited a square S(row,col) his itinerary consists of that square. If the knight has visited several squares his itinerary is a construction of the square last visited and the previous route Route(Square,itinerary).

The knight starts on square (1,1) and has then advanced one step.

```
Knights_tour(1,S(1,1),n) ←
```

After two advances the itinerary consists of the previous square, i.e. the starting square and the next possible square.

```
Knights_tour(2,Route(square,previous_square),n) ←
        Knights_tour(1,previous_square,n),
        Possible_step(previous_square,square,n)
```

There is an itinerary consisting of more than two steps, i.e. Route(square, Route(previous_square,itinerary)) if there exists an itinerary that is shorter by one step, i.e. Route(previous_square, itinerary) and we can advance from the square previous_square to square, i.e. square is a possible advance and not yet visited. We have to divide the structure of the one step shorter itinerary into the previous square and the previous itinerary, since the square we are advancing to is dependent on where the knight was placed previously.

```
Knights_tour(visited,Route(square,Route(previous_square,itinerary)),n) ←
        Value(visited - 1,previously_visited),
        Knights_tour(previously_visited,Route(previous_square,itinerary),n),
        Possible_step(previous_square,square,n),
        NotVisited(square,itinerary)
```

The remaining definitions in the program are presented by giving a description in a *comment* included in special parentheses /* and */.

```
/* It is possible to advance to a square
        if it is a permitted advance for the knight and
        the square is within the limits of the board. */
Possible_step(previous_square,square,n) ←
        Step(previous_square,square),
        Within(square,n)

/* The valid advances of the knight */
Step(S(row,col),S(row1,col1)) ←
        Value(row-2,row1), Value(col-1,col1);
        Value(row-2,row1), Value(col+1,col1);
        Value(row-1,row1), Value(col+2,col1);
        Value(row+1,row1), Value(col+2,col1);
        Value(row+2,row1), Value(col+1,col1);
        Value(row+2,row1), Value(col-1,col1);
        Value(row+1,row1), Value(col-2,col1);
        Value(row-1,row1), Value(col-2,col1)

/* The board has squares from 1 upto n */
Within(S(row,col),n) ←
        row > 0, col > 0, row ≤ n, col ≤ n

/* A square S(row,col) is unvisited
        if every previously visited square of the itinerary
        differs from S(row,col) */
Unvisited(S(row,col),S(row1,col1)) ←
        S(row,col) ≠ S(row1,col1)
Unvisited(square,Route(square1,itinerary)) ←
        square ≠ square1,
        Unvisited(square,itinerary)
```

For a trace of how the knight can make a tour of five advances on a board size of 4 x 4 squares, see Appendix B.2.

2.5 Exercises

Exercise 1:
Unify the following expressions and construct the substitution:

a) Father(x,y) and Father(z,Frey)
b) And(x,Or(y,z)) and And(u,w)
c) Sum(x,y,S(z)) and Sum(S(x),S(S(0)),S(x))

Exercise 2:
Define a relation Fibonacci between two numbers. Every number in a Fibonacci series, except the first two, is the sum of the two previous numbers

in the series. The series starts with the numbers 0 and 1, i.e. the beginning of an expansion of the series will be as follows: 0, 1, 1, 2, 3, 5, 8, 13, ... The relation Fibonacci shall be a relation between a number from the Fibonacci series and the position of the number in the series.

```
← Fibonacci(7,8)
yes
```

Construct the search space for Fibonacci(4,2) bottom-up and top-down.

Exercise 3:

Demonstrate in a resolution proof that Halley's comet is a celestial body with the help of the following statements:

(1) Celestial_body(x) ← Comet(x)
(2) Celestial_body(x) ← Star(x)
(3) Comet(x) ← Tail(x), Close_to_the_sun(x)
(4) Close_to_the_sun('Halleys Comet') ←
(5) Tail('Halleys Comet') ←

Exercise 4:

Given the clauses

(1) ← Equiangular(x), Has_obtuse_angle(x)
(2) Equiangular(x) ← Equilateral(x)

prove that the following clause holds:
The triangle T is not equilateral if T has an obtuse angle, i.e.

$$\neg Equilateral(T) \leftarrow Has\_obtuse\_angle(T)$$

The solution shall consist of a proof in which the resolution steps are shown. Remember that resolution is used in a proof by contradiction.

Round 3. Data Structures

3.1 Constructed Terms

In our examples we have met with different types of terms. We have used, for instance, the constants Thor, Odin, and 0 and the variables x, parent, and itinerary. Apart from variables and constants we have used constructed terms or structures. A structure is a collection of terms and corresponds to a structural relation between the terms. We express a structure by a constructor and one or several arguments that are terms. We may denote a pair, for instance, by the constructor Pair and the two constants A and 5 as the two arguments: Pair(A,5).

In Section 1.3 on Definitions we used structures to express natural numbers and logical expressions. We used the constructor S with an argument that was a natural number. The constructed term S(0) denotes the number 1. When we use the constructor S we express the fact that we are referring to the successor of the argument. The term S(S(0)) then denotes the number 2 and S(n) denotes the number n+1. In S(S(0)) the argument is a structure as well, and in this case we say that S(S(0)) is a *nested term* with *depth* 2.

We use structures when we want to refer to a collection of objects. It may be convenient at one time to be able to view the composite term as an object, and at another time to consider its parts. We prefer to be just as detailed as necessary when we express a relation.

Let us define the relation Born with two arguments. Its first argument is the name of a person and its second argument is a structure containing the date of birth. In this case we express day, month, and year by numbers, not by the structure S which is inconvenient for the represention of large numbers.

```
Born(John,Date(05,01,64)) ←                    (96)
Born(Mary,Date(19,08,64)) ←                    (97)
```

We can define one person as being older than another in a relation Older. We compare the birth dates of two persons and express that one date is earlier than another using the relation Earlier.

```
Older(person1,person2) ←                                    (98)
        Born(person1,date1), Born(person2,date2),
        Earlier(date1,date2)
```

In clause (98) we are satisfied with expressing that the relation Earlier holds for the birth dates of the persons in question; we need not enter into how we have represented birth date. In the definition of the relation Earlier, however, in which we compare two dates, we must have an exact knowledge about the representation of a date. We have represented date as a structure by the constructor Date and three arguments. We use this form in the relation Earlier and can directly reach the variables denoting year, month, and day which we use for our comparison.

```
Earlier(Date(day1,month1,year1),Date(day2,month2,year2)) ←   (99)
        year1 < year2
Earlier(Date(day1,month1,year),Date(day2,month2,year)) ←    (100)
        month1 < month2
Earlier(Date(day1,month,year),Date(day2,month,year)) ←      (101)
        day1 < day2
```

An alternative is to represent the dates in (96) and (97) as an integer and in (98) only compare which of the integers is the smallest. A representation of dates in a structure like (96) and (97), however, makes it easier to express, for example, that two persons were born in the same year.

```
Born_same_year(person1,person2) ←
        Born(person1,Date(day1,month1,year)),
        Born(person2,Date(day2,month2,year))
```

The choice of representation is thus dependent on how we want to use the term. We use the structure Date to group three objects that we want to be able to reach separately and still be able to refer to as a unity.

Another use of structures is when we have objects that are recursively defined, like natural numbers and the S structure above. A definition of a relation that has a recursive structure as argument can be expressed recursively. We may, for instance, express a relation Product recursively over the structure S. In the base case we state that the product of 0 and an arbitrary number is 0. In the recursive case we declare that the product of S(x) and a number y is the sum of y and the product of x and y. In the definition of Product we use the relation Sum from Section 1.3 on Definitions in Round 1.

```
Product(0,y,0) ←                                            (102)
Product(S(x),y,z) ←                                         (103)
        Product(x,y,w),
        Sum(y,w,z)
```

In Section 1.3 we learned that recursive definitions consist of base cases and recursive cases. In clauses (102) and (103) we express the clauses recursively

over the first argument in Product. The base case consists of a consequence where the first argument in the relation is 0. The recursive case has a consequence where the first argument is S(x) and a conditional part where the first argument is x. In order to be evaluated the recursive cases and the base cases in a definition have to be coordinated so that the structure only can obtain forms that can be captured by the base cases. At an evaluation of Product the depth of the structure will be successively reduced by one and the structure will thus be reduced to the base case form.

In our definition of product the relation is true when the third argument is the product of the first two. By instantiating the arguments in Product in different ways, we can use the definition to multiply two numbers, divide two numbers, or divide a number into factors.

We will formulate a few questions about the relations Earlier and Product which we defined above. To make sure that one date is further back in time than another, we can ask a question. We use Earlier as defined in (99) – (101) and denote the date by the structure Date.

```
← Earlier(Date(31,05,84),Date(01,06,84))
yes
```

The value for month in the first argument is less than the value for month in the second argument and according to (100) the expression is true.

When we use Product for multiplying two numbers we give the values of the first two arguments and let the third remain uninstantiated. To multiply the number 3 by the number 2 we express the question

```
← Product(S(S(S(0))),S(S(0)),x)
x = S(S(S(S(S(0)))))
```

When we want to divide the number 6 by the number 3 we give the expression for the number 6 as third argument and the expression for the number 3 as the first or second argument.

```
← Product(S(S(S(0))),x,S(S(S(S(S(0))))))
x = S(S(0))
```

We can also use Product to get the factors of a number. We give the third argument and get the different factors a number can be divided into. We ask into what factors the number 4 can be divided.

```
← Product(x,y,S(S(S(S(0)))))
x = S(0)
y = S(S(S(S(0))))
```

A number often consists of several pairs of factors and so we may receive several answers.

3.2 List Structures

The list structure[1] is a representation of sequences of terms[2]. We call the terms in the list *elements*. The number of elements in a list is not fixed; we can always add or remove an element, thus the list is a *dynamic* structure. The structure Date that we have used above always contains three elements and is an example of a *static* structure.

When we build a structure we can place the constructor before, between or after the arguments. A structure that has two arguments is naturally expressed with the constructor between the arguments. We say that a constructor is *infix* if placed between the arguments. A constructor that is placed before the arguments is called *prefix* and if placed after the arguments it is called *postfix*.

We use the constant $\emptyset$ for the empty list and the structure x.y for the constructed list. The punctuation mark, ".", is an infix constructor and the terms x and y are the arguments. The term x denotes the first element in the list and y denotes that list which is the rest of the list. If we want to express the constructed list in the same way as those structures we have seen so far, i.e. in prefix form, it will be .(x,y). With our notation, a list with the elements 1, 2, and 3, will be written as 1.2.3.$\emptyset$. It is composed of the element 1 and the list 2.3.$\emptyset$. List 2.3.$\emptyset$, in its turn, is composed of the element 2 and the list 3.$\emptyset$. List 1.2.3.$\emptyset$ is thus a nested term with the depth 3.

The elements in a list are terms and thus can also be structures. A list of dates in which we use the structure Date can be written as

Date(02,04,87).Date(03,04,87).Date(04,04,87).$\emptyset$

When we express a list of lists, we use parentheses to separate the levels.

(Su.Mo.Tu.$\emptyset$).(We.Th.$\emptyset$).(Fr.Sa.$\emptyset$).$\emptyset$

A definition of the list structure says that a list is either empty or constructed by an element and a list. The definition will look as follows in standard form[3]

$$\forall l\,(List(l) \leftrightarrow l = \emptyset \vee \exists x \exists y\ (l = x.y \wedge Element(x) \wedge List(y)))$$

The structural property of the list is that the position of an element in a list is related to the positions of the other elements in the list; an element is either before or after another element in the list.

Depending on what properties structures have, we can express different fundamental relations about them. We will study some fundamental relations of the list structure.

---

[1] We also use the names linear list and simple list.

[2] See D. E. Knuth, *The Art of Computer Programming, Vol 1, Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1975.

[3] Å. Hansson and S-Å. Tärnlund have formulated definitions of list structures in "A Natural Programming Calculus", Proc. IJCAI-6, Tokyo, 1979.

We want to be able to reach elements in a list. When a list contains some element it is constructed and has the form x.y. In a constructed term we can reach the arguments directly and thus we can reach the first element in the list directly. We say that we have *access* to the beginning of the list. When we want to reach elements at specific positions in lists or specific elements at unknown positions we define general relations expressing this.

We can, for example, be interested in the kth element in a list and define a relation between a list and the element in position k in the list. Let us call the relation Position and let the position be represented by the constructor S. We assume that position one corresponds to the first element in the list and get a definition with two clauses. We say that the element in the first position in list is elem if list = elem.restlist and that the successor of the nth element in list is elem if list = x.restlist and the nth element in restlist is elem.

```
Position(S(0),elem,list) ← list = elem.restlist                (104)
Position(S(n),elem,list) ← list = x.restlist,                  (105)
      Position(n,elem,restlist)
```

We can use a constructor to *construct* structures or to *decompose* structures. In clause (104) we are looking for the first element in list and decompose the list into the first element and the rest so that we can access the first element. In (105) we are interested in the list after the first element and decompose list so we can express the clause about this restlist.

As we have learned in Section 1.3, the definition of Position can be abbreviated by putting the decomposition of the list in the consequence part of the clauses, instead.

Often, when using structures in a program, we want to know if a certain term is part of a structure. For lists this corresponds to deciding whether a specific element is a member of the list. We will say that an element is a member of a list if it is equal to the first element in the list. Moreover, we define the fact that an element elem is a member of a list if the list is constructed by an element x and a list restlist and elem belongs to restlist.

```
Member(elem,elem.restlist) ←                                   (106)
Member(elem,x.restlist) ←                                      (107)
      Member(elem,restlist)
```

We may also want to change an element in a list. Since we have access to the beginning of the list, we can change the first element, directly, using only the constructor. Again, when we want to change an element at a specific position or a specific element at an unknown position, we have to define a recursive relation over the list structure.

To enlarge or decompose lists we can add or remove, as the case may be, a sublist from the beginning of the list. However, we may also want to manipulate the rear of a list. We may want to add one or a few elements to the end of a list, or more generally, add a list to another, or divide a list into two smaller lists.

A relation Append between three lists, in which the first two lists put together make up the third list, expresses a relation we can use for such manipulation.

We say that when we append a list list to an empty list, the composite list is list. When we append a list list2 to a list elem.list1, the composite list is a list elem.list3, if list2 appended to list1 is list3.

$$\text{Append}(\emptyset,\text{list},\text{list}) \leftarrow \tag{108}$$

$$\text{Append}(\text{elem.list1},\text{list2},\text{elem.list3}) \leftarrow \tag{109}$$
$$\quad \text{Append}(\text{list1},\text{list2},\text{list3})$$

Both Member and Append are recursively defined over the list structure. The Append relation is general; we can append a list to any list. Concomitantly, it has a base case for the case in which the list in the first argument is empty, and a recursive case where the relation holds for elem.list1 if the relation holds for list1 which is one element smaller. But the relation Member, (106) and (107), is not valid for all lists. It is true only when the element in the first argument is in the list. In the definition of Member clause (106) is a base case. If we expressed Member for the empty list we would get the clause

$$\leftarrow \text{Member}(\text{elem},\emptyset) \tag{110}$$

Clause (110) expresses the fact that no element belongs to the empty list. It expresses something that *does not hold* for Member. But we are concerned only with what *holds* for the relation and thus we will leave out such a clause.

We say that a list y is a *permutation* of list x if both contain the same elements. For example, the list B.A.C.$\emptyset$ is a permutation of the list A.B.C.$\emptyset$. For a list y to be a permutation of x, the elements in y do not have to be in the same order as the elements in x. If two lists contain the same elements and the elements are given in the same order in the two lists, they are equal or *identical.*

We may be interested in having the elements in a list sorted according to some order relation, for example, to have a list of numbers placed in ascending order. We shall express a relation Sort[4] in which the first argument is an arbitrary list of numbers and the second argument is a list which is a permutation of the list in the first argument with the elements in ascending order.

The sorted permutation of an empty list is an empty list. The sorted permutation of a constructed list elem.restlist consists of the list we get when elem has been put in the correct place in the sorted permutation of restlist. To put an element in a sorted list, we also have to define a relation Insert.

$$\text{Sort}(\emptyset,\emptyset) \leftarrow \tag{111}$$

$$\text{Sort}(\text{elem.restlist},\text{slist}) \leftarrow \tag{112}$$
$$\quad \text{Sort}(\text{restlist},\text{srestlist}),$$
$$\quad \text{Insert}(\text{elem},\text{srestlist},\text{slist})$$

[4] The algorithm is called List Insertion Sort in D. E. Knuth, *The Art of Computer Programming, vol 3, Sorting and Searching*, Addison-Wesley, Reading, MA, 1973

Insert(elem,list1,list2) is a relation between an element elem, a sorted list list1, and a sorted list list2 which in addition to all the elements of list1, also contains elem. The base case for the program is the case when the first list is empty. When the first list is empty, the second will contain only the element elem.

$$\text{Insert(elem,}\emptyset\text{,elem.}\emptyset\text{)} \leftarrow$$

When the first list is constructed, x.list1, and elem is less than or equal to the first element x, the second list is constructed from elem, x, and from the restlist list1. Finally, Insert holds for elem and two constructed lists with the same first element x, if elem is greater than x and if Insert holds for elem and the restlists.

$$\begin{aligned}&\text{Insert(elem,x.list1,elem.x.list1)} \leftarrow \text{elem} \leq \text{x}\\ &\text{Insert(elem,x.list1,x.list2)} \leftarrow \text{elem} > \text{x},\\ &\qquad\text{Insert(elem,list1,list2)}\end{aligned}$$

We will exemplify the definitions above with a few questions. We can investigate, for example, whether an element belongs to a list.

```
← Member(3,0.1.1.2.3.5.8.∅)
yes
```

Apart from finding out if an element belongs to a list, we get to know what elements belong to a given list.

```
← Member(elem,Blue.Yellow.Red.∅)
```

The answer is `elem = Blue`, `elem = Yellow` or `elem = Red`.

If we want to join two lists we can use Append.

```
← Append(A.B.C.∅,D.E.∅,list)
list = A.B.C.D.E.∅
```

We can find out if a list constitutes the first part of another list

```
← Append(A.B.∅,list,A.B.C.D.E.∅)
```

Here we learn that A.B.∅ constitutes a part of A.B.C.D.E.∅ and that the remaining part is C.D.E.∅. If we had given no value either for the first or the second argument in Append, we would have had alternative divisions of A.B.C.D.E.∅ as the answer, for example, A.B.C.∅ and D.E.∅.

We may add an element to a sorted list by using Insert

```
← Insert(7,1.2.3.5.∅,list)
list = 1.2.3.5.7.∅
```

If we want to sort a list we can use the relation Sort above. The list is given as the first argument.

```
← Sort(2.0.3.1.8.1.5.∅,list)
list = 0.1.1.2.3.5.8.∅
```

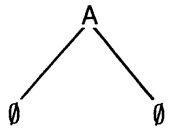

Figure 3.1: A binary tree with only one node.

3.3 Tree Structures

We will proceed to study another data structure, viz. *trees*[5]. A tree is built up by *nodes* and *branches* between nodes. The structural property of a tree is the ramifying relation between the nodes. A node A may have branches to nodes B, C and D, which may, in their turn, have branches to several nodes.

A tree is a collection of nodes where one node is the *root* of the tree and the other nodes belong to the *subtrees*. A tree in which every node has two subtrees is called a *binary tree*. Every tree with an arbitrary number of subtrees in the nodes can be changed into a binary tree and we will restrict our study of trees to binary trees.

A binary tree is either empty or composed of a constructor and three arguments. These consist of *the left subtree*, *the root of the tree*, and *the right subtree*, and the subtrees are binary trees. Let us use T as constructor for binary trees. We express the definition in standard form.

$$\forall x\,(\mathit{Binary\_tree}(x) \leftrightarrow$$
$$x = \emptyset \vee \exists ls \exists r \exists rs\,(x = T(ls, r, rs)$$
$$\wedge\ \mathit{Binary\_tree}(ls) \wedge \mathit{Root}(r) \wedge \mathit{Binary\_tree}(rs)))$$

The binary tree T(∅,A,∅) denotes a binary tree with the node A as root and with empty subtrees only, see Figure 3.1. The binary tree in Figure 3.2 is expressed in our notation as

$$\mathsf{T}(\mathsf{T}(\emptyset,\mathsf{A},\emptyset),\mathsf{B},\mathsf{T}(\emptyset,\mathsf{C},\mathsf{T}(\emptyset,\mathsf{D},\emptyset)))$$

This is a tree that has three other nodes apart from the root. In the left subtree, node A is the root and in the right subtree the root is C. Nodes with empty subtrees only are called *leaves*. In Figure 3.1 root A is a leaf and in Figure 3.2 nodes A and D are leaves.

We use the tree structure when our object has a tree-like structure. A typical example is the family tree. The term family tree denotes the fact that the relations between the objects form branches. We can also say that the objects stand in a one-to-many relation to each other[6]. We can express the family tree

---

[5] See D. E. Knuth, *The Art of Computer Programming, Vol 1, Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1975.

[6] See Round 4, Section 4.2 on Databases.

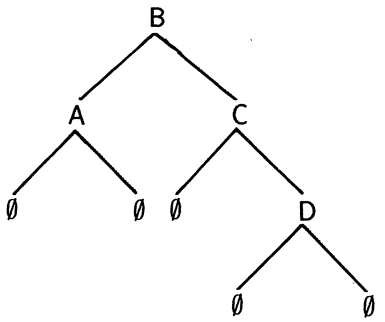

Figure 3.2: A binary tree.

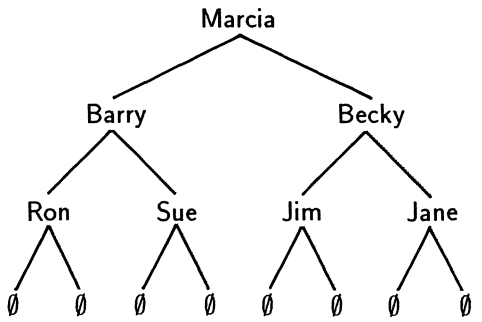

Figure 3.3: A family tree.

in such a way that every node denotes a child and either root of its two subtrees stands for the parents of the child. Thus the family tree forms a binary tree structure, see Figure 3.3. Expressed as a structure with the constructor T we have

```
T(T(T(∅,Ron,∅),Barry,T(∅,Sue,∅)),
                Marcia,
T(T(∅,Jim,∅),Becky,T(∅,Jane,∅)))
```

The knowledge of who is the father and the mother of somebody is implicit in the structure. We have constructed the tree so that if a child is the root of the tree, the father is to be found in the root of the left subtree and the mother in the root of the right subtree. We can express the relation Mother_of(child,mother,family_tree) by a first argument denoting a child and a second denoting its mother, and a third argument which shall be a binary tree constructed as our example above.

The first clause expressed the fact that the mother of the child who is the root of the tree is in the root of the right subtree. By using T for decomposition

we can express this directly in the consequence.

$$\mathsf{Mother\_of(child,mother,T(left,child,T(ls,mother,rs)))} \tag{113}$$

The other clauses express the fact that someone is mother of a child if the relation holds for the child, the mother, and any of the subtrees.

$$\mathsf{Mother\_of(child,mother,T(left,root,right))} \leftarrow \mathsf{Mother\_of(child,mother,left)} \tag{114}$$

$$\mathsf{Mother\_of(child,mother,T(left,root,right))} \leftarrow \mathsf{Mother\_of(child,mother,right)} \tag{115}$$

The relation is recursively defined over the third argument which is a binary tree. We have a base case, (113), for the case when the first argument and the root of the tree are equal. In the recursive cases in a definition, the relation holds for a structure if it holds for its components. A relation with a constructed binary tree holds if it holds for the two subtrees that are connected to the root. However, a relation with a constructed binary tree sometimes holds if it holds for some of the two subtrees also. The latter case occurs when a definition describes a property of one of the nodes or when the problem is to establish that there exists a certain node in the tree. The former case occurs when a definition expresses the fact that all nodes in the tree have a property or when a definition addresses all nodes in the tree with a certain property.

A relation that is recursively defined over a binary tree can thus be defined to hold either if the relation holds for both of the subtrees, or if the relation holds for one of the subtrees. In the relation Mother_of above, the definition expresses the existence of a mother of a child. It is enough that the relation holds for one of the subtrees, see clauses (114) and (115).

Another use of the tree structure is when the objects in the structure are ordered and we want access to arbitrary objects. If we use an *ordered binary tree* and *binary search*[7] we will find an arbitrary node with fewer searches than if we search for it sequentially in a *ordered* list, for instance.

That a tree is ordered means that a node is placed in a certain position in the tree according to its value. To define an ordered tree we need to decide on the order relation of the objects and to decide in what order to visit the nodes. We can traverse a binary tree so that we first visit the left subtree, then the root, and finally, the right subtree. The root is visited between the subtrees and this order is called *inorder*. Alternative ways of traversing trees are *preorder* and *postorder*. Preorder means first visiting the root of the tree, then the left subtree, and then the right. Postorder, finally, means that we first visit the left subtree, then we visit the right subtree, and last, the root.

Let us suppose that we have an ordered binary tree where the nodes are numbers and ordered in ascending order in inorder, for example see Figure 3.4.

---

[7] See D. E. Knuth, *The Art of Computer Programming, Vol 3, Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

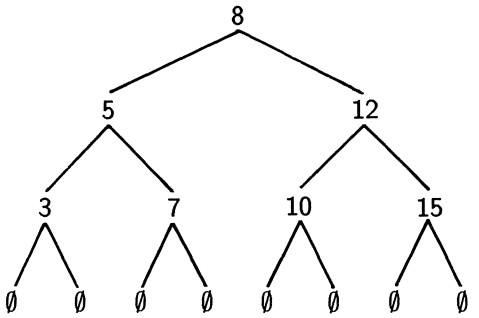

Figure 3.4: An ordered binary tree.

We will express the property ordered binary tree for a tree ordered in inorder. We say that a structure is an ordered binary tree if, and only if, either the tree is the empty tree or it is constructed of a root with a left and a right subtree such that all nodes in the left subtree are less than the root and all nodes in the right subtree are greater than the root. Furthermore, the left and the right subtree both have to be ordered binary trees.

$$\begin{aligned}\forall x\,(\mathit{Ord\_binary\_tree}(x) \leftrightarrow \\ x = \emptyset \vee \exists ls \exists r \exists rs\,(x = T(ls, r, rs) \\ \wedge \forall u\,(\mathit{Member}(u, ls) \rightarrow\ u < r) \\ \wedge\ \forall u\,(\mathit{Member}(u, rs) \rightarrow\ u > r) \\ \wedge\ \mathit{Ord\_binary\_tree}(ls) \wedge \mathit{Ord\_binary\_tree}(rs)))\end{aligned}$$

In the definition of ordered binary tree the relation Member is used for binary trees. We will express this and other fundamental relations for binary trees.

We say that a node belongs to a binary tree if it is equal to the root of the tree and that a node belongs to a binary tree if it belongs to any of the subtrees.

```
Member(node,T(ls,node,rs)) ←                    (116)
Member(node,T(ls,root,rs)) ← Member(node,ls)    (117)
Member(node,T(ls,root,rs)) ← Member(node,rs)    (118)
```

As we have seen earlier, it is trivial to add or subtract the first element of a list. It is not quite as simple in the case of trees. We have direct access to the root of the tree, but to take it away or to add a new root to an ordered tree will cause problems as to what shall constitute the new subtrees. To insert or remove[8] an element from an ordered binary tree, again in the general case, means fewer searches generally than if we used a sorted list.

[8] In Round 4, Section 4.2, we give a definition of a relation expressing removal of a node from an ordered binary tree.

We will define a relation between a node node, an ordered binary tree t and another ordered binary tree which, together with the nodes from t, contains node. When the first tree is empty the second is a tree with the node as root and two empty subtrees.

```
Insert(node,∅,T(∅,node,∅)) ←
```

When the first tree already has the node as root, the second tree is identical to the first.

```
Insert(node,T(ls,node,rs),T(ls,node,rs)) ←
```

If, on the other hand, node is less than root, node shall be inserted into the left subtree. The root and the right-hand subtree are not influenced in this case. Further, if node is greater than root we get a case expressed analogously.

```
Insert(node,T(ls,root,rs),T(ls1,root,rs)) ← node < root,
        Insert(node,ls,ls1)
Insert(node,T(ls,root,rs),T(ls,root,rs1)) ← node > root,
        Insert(node,rs,rs1)
```

Further, we will express a relation that describes a mapping between a binary tree and a list. For every node in the tree there is an element in the list and the order of the elements in the list is the order we get when we traverse the tree in inorder. Let us call the relation Inorder. When the tree is empty, the list is empty, too. When the tree is constructed the list is the concatenation of the list of the nodes in the left subtree and the list of the element root and the nodes of the right-hand subtree.

```
Inorder(∅,∅) ←
Inorder(T(ls,root,rs),list) ←
        Inorder(ls,lslist), Inorder(rs,rslist),
        Append(lslist,root.rslist,list)
```

The relations with binary trees we met before Inorder were all examples of definitions in which the expression with the subtrees consisted of a disjunction. In Inorder the relation must hold for both the subtrees for it to hold for the binary tree, and for us to get only one recursive clause.

We may say that the root of a tree is the mother of the roots in the subtrees and that nodes in a tree are sisters if they have the same mother. We shall express the relation sisters for binary trees. In contrast to the previous programs for binary trees we want to reach nodes on depth two in the tree, in the base case. This means that the relation Sister is only defined for trees with a depth deeper than one. We express Sister with the structure in two levels and get direct access to the nodes wanted.

```
Sisters(x,y,T(T(ls1,x,rs1),root,T(ls2,y,rs2))) ←
Sisters(x,y,T(ls,root,rs)) ← Sisters(x,y,ls)
Sisters(x,y,T(ls,root,rs)) ← Sisters(x,y,rs)
```

We will finish this section by formulating consequences from some of the relations we have defined for trees.

```
← Member(node,T(T(∅,A,∅),B,T(∅,C,∅)))
node = B

← Inorder(T(T(∅,A,∅),B,T(∅,C,∅)),x)
x = A.B.C.∅

← Mother_of(Becky,mother,T(T(∅,Barry,∅),Marcia,
                           T(T(∅,Jim,∅),Becky,T(∅,Jane,∅))))
mother = Jane
```

3.4 Difference Lists

To reach the lowest level of a nested structure, we have so far traversed the entire structure recursively. An alternative is to have access from the rear as well.

Working with the list structure we have use for a means of reaching the rear of the list, not only the front. By using Prolog's matching we can achieve such a result. When we manipulate the rear of a list without traversing the list, we get a more efficient program.

We will study the relation Reverse which for two lists expresses the fact that the elements of one list are in reverse order compared to the other list. We say that the inversion of an empty list consists of an empty list. The inversion of a list with a first element elem and a restlist restlist is composed by the list we get when elem is added to the end of the inversion of restlist. We can use Append which was defined in Section 3.2 on List Structures to add elem to the end of revrestlist.

```
Reverse(∅,∅) ←                                          (119)
Reverse(elem.restlist,revlist) ←                        (120)
        Reverse(restlist,revrestlist),
        Append(revrestlist,elem.∅,revlist)
```

We will consider an example of a question using the Reverse relation.

```
← Reverse(Do.Re.Mi.∅,reverselist)
reverselist = Mi.Re.Do.∅
```

To get the elements placed in reversed order we use the Append relation. This solution is not efficient, however, because for every element we add to the rear of a list, we have to traverse the entire list, element by element.

By using an auxiliary argument, we can get a program that achieves the same result without the relation Append. When the list is constructed the

relation holds if it holds when the list is reduced by the first element and the auxiliary list is increased by that element.

$$\mathsf{Reverse}_{aux}(\emptyset,\mathsf{list},\mathsf{list}) \leftarrow \tag{121}$$

$$\mathsf{Reverse}_{aux}(\mathsf{elem.restlist},\mathsf{auxlist},\mathsf{revlist}) \leftarrow \tag{122}$$
$$\mathsf{Reverse}_{aux}(\mathsf{restlist},\mathsf{elem.auxlist},\mathsf{revlist})$$

When we formulate questions to Reverse we give an empty list as the value for the auxiliary list. We can avoid this by defining a relation between Reverse and $\mathsf{Reverse}_{aux}$.

$$\mathsf{Reverse}(\mathsf{x},\mathsf{y}) \leftarrow \mathsf{Reverse}_{aux}(\mathsf{x},\emptyset,\mathsf{y})$$

It is fairly easy to relate $\mathsf{Reverse}_{aux}$ to Reverse but to give an explanation of what the program $\mathsf{Reverse}_{aux}$ says is not quite so easy. The problem is to explain the relation between the three lists. If we study the relation $\mathsf{Reverse}_{aux}$, we see that the auxiliary list and the list in the third argument together express the reversed list. The inversion of the first list is the difference between the other two.

A way of surmounting the problem of explanation is to use a structure with a suitable interpretation. We can use the data structure *d-list*[9] that allows us to reach the rear of a list using the matching in Prolog. The list data structure *d-list* is a list structure which constitutes the difference between two lists. This pair of lists is so constructed that the second is the tail of the first in the pair.

We use the constructor D and two lists as arguments to express the data structure d-list. In D(x,y), y is the end of x. For example, from the lists A.B.C.x and the list x we can construct D(A.B.C.x,x) which corresponds to the simple list A.B.C.$\emptyset$. The structure D(Do.Re.Mi.$\emptyset$,Re.Mi.$\emptyset$) denotes a list with Do as its only element. An empty list is thus expressed by two equal lists as argument to D. D(x,x) and D(1.2.$\emptyset$,1.2.$\emptyset$), for example, both denote the empty list. Note that unifiers that allow unification of a variable and a structure which contains the variable will generate an infinite or cyclic structure if we try to unify, for instance, D(x,x) and D(A.B.C.y,y).

If we have a d-list where the second list and the corresponding part of the first list are uninstantiated, i.e. D(A.B.C.y,y), we can append a list to the rear by unification. If we express an Append relation for such a list, we get a program that consists of only one clause without conditional part.

$$\mathsf{Append}(\mathsf{D}(\mathsf{x},\mathsf{y}),\mathsf{D}(\mathsf{y},\mathsf{z}),\mathsf{D}(\mathsf{x},\mathsf{z})) \leftarrow \tag{123}$$

We can append the list D(E.F.z,z) to D(A.B.C.D.y,y) using the clause (123) and the question below. By giving also the third argument as a structure we get the answer in the form of a simple list.

---

[9] The data structure d-list, which is an abbreviation of difference list, was specified by Å. Hansson and S-Å. Tärnlund in the paper "A Natural Programming Calculus", Proceedings of IJCAI-6, Tokyo, 1979.

```
← Append(D(A.B.C.D.y,y),D(E.F.z,z),D(list,∅))
list = A.B.C.D.E.F.∅
```

Since the program Append above consists only of unification of terms, we can very often express the condition that certain terms shall be equal without using Append and still preserve clarity.

We express Inorder from the previous section without using Append by reformulating the definition with a d-list as second argument.

```
Inorder(∅,D(x,x)) ←
Inorder(T(x,y,z),D(u,w)) ←
        Inorder(x,D(u,y.v)), Inorder(z,D(v,w))
```

Furthermore, we can give an alternative definition for the Reverse relation, this time by using a d-list as second argument. The inversion of an empty list is an empty list. The relation Reverse holds for a constructed list and the corresponding reversed list if Reverse holds when the first element is removed from the list and the last element is removed from the reversed list.

```
Reverse(∅,D(x,x)) ←                                    (124)
Reverse(elem.restlist,D(x,y)) ←                        (125)
        Reverse(restlist,D(x,elem.y))
```

Let us use the program to obtain the reversed list of Do.Re.Mi.∅. We then ask following question:

```
← Reverse(Do.Re.Mi.∅,D(list,∅))
list = Mi.Re.Do.∅
```

3.5 Array Structures

So far, we have used terms for all our data structures. We have studied lists, trees, and lists represented as a difference between two lists (d-lists). Terms are so general that no other data structures are needed. Let us now increase the set by a term representation of the array data structure.

An array consists of elements, all of the same type, which are grouped together. Each element is accompanied by a so-called *index* denoting the position of the element in the structure. Consequently, an array is a set of pairs: index and value. In order to denote an individual element, the name of the entire structure is augmented by the index selecting the element. An array can be of several *dimensions*, e.g. one-dimensional vectors (see Figure 3.5), two-dimensional tables (see Figure 3.6), and so on. The dimensions of an array are fixed at the creation and, consequently, the array data structure is static. The number of elements in the structure is a function of the arrays dimensions.

A vector can be used to hold the values of a function with one argument, the domain being the index type and the array elements the function values. See for example the function Fibonnaci for the arguments 0 to 4 in Figure 3.5.

$$\begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 2 \\ 3 \end{pmatrix}$$

Figure 3.5: Vector with the values for the Fibonnaci series.

$$\begin{matrix} & 1 & 2 & 3 \\ 1 & & & \\ 2 & & & \\ 3 & & & \end{matrix} \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

Figure 3.6: A 3x3 array.

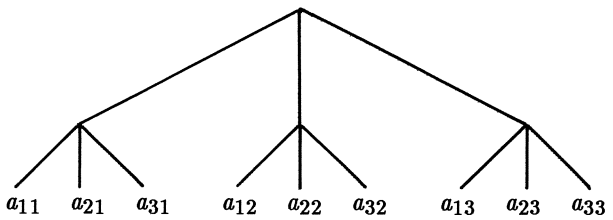

Figure 3.7: An array as a tree with column priority.

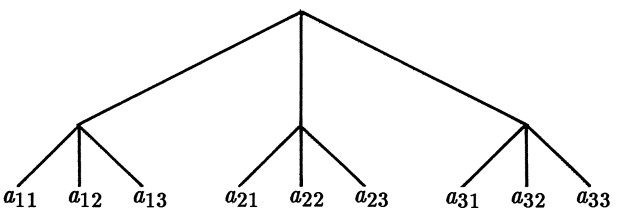

Figure 3.8: An array as a tree with row priority.

The access to the elements in the data structure is called random-access since all components can be selected at random and are equally accessible.

An array can be thought of as a specialization of the linear list data structure. In a two-dimensional array each element belongs to two linear lists: one *row* list and one *column* list. An array can be represented as a special case of the tree structure. In such a representation of a two-dimensional array, priority is given to one of the two relationships: the column or the row relationship, see Figures 3.7 and 3.8, respectively.

Let us start with a definition of the array structure in standard form using a tree-structured representation. The dimension of an array is indicated by a list. An array without a dimension, i.e. with dimension equal to an empty list is an element. An array with a dimension of at least one, i.e. the dimension is equal to n.mdim, consists of n arrays of dimension mdim. The index is given starting from 0 up to n-1; each position in the array valid according to the index holds an array. The representation favors rows.

$$\begin{aligned}
&\forall dim \forall w\,(Array(dim, w) \leftrightarrow \\
&\quad dim = \emptyset \wedge Element(w) \\
&\quad \vee \exists n \exists mdim \exists list\,(dim = n.mdim \\
&\qquad \wedge w = Array(list) \\
&\qquad \wedge \forall i\,(0 \leq i \wedge i < n \rightarrow \\
&\qquad\quad \exists subw\,(Array(mdim, subw) \wedge Pos(i, subw, list)))))
\end{aligned}$$

Each dimension in an array is represented as a list (a subtree). Each element in the list is an array with a dimension that is one less than the dimension of the array represented by the list.

The relation Pos(w,i,v) where v is the element (subarray) in ith place in the list w is defined below. The first place in the list has the position 0. w is a list u.recw.

$$\begin{aligned}
&\forall i \forall v \forall w\,(Pos(i, v, w) \leftrightarrow \\
&\quad \exists u \exists recw\,(w = u.recw \\
&\qquad \wedge (i = 0 \wedge v = u \\
&\qquad\quad \vee i > 0 \wedge j = i - 1 \wedge Pos(j, v, recw))))
\end{aligned}$$

Typical array relations concern a specific place in the array. We may want to retrieve the value of an element at a specific place and insert a new value also at a specific place. Before operations can be performed we actually need to create the array. Remember that the array is a static object. The number of possible values in the array is always the same irrespective of whether the value accompanying an index is instantiated or not.

The predicate Array can be used to create an array and for type checking a term. In Array(dim, array), the variable array represents an array with dimension dim. A Horn clause program defining the Array relation can be derived from the specification. An array with empty dimensions is an element and an array with the dimensions n.mdim consists of n subarrays with the dimension mdim.

$$\text{Array}(\emptyset,\text{w}) \leftarrow \text{Element(w)} \tag{126}$$

$$\text{Array(n.mdim,Array(w))} \leftarrow \text{ArrayList(n,w,mdim)} \tag{127}$$

The auxiliary relation ArrayList has three arguments: an integer, a list, and a dimension. The relation ArrayList(n,list,dim) is satisfied if the list contains n elements that are all arrays of the dimension dim.

```
ArrayList(0,∅,dim) ←                                    (128)
ArrayList(n,u.w,dim) ← n > 0,                           (129)
        Array(dim,u), Value(m,n-1),
        ArrayList(m,w,dim)
```

For example, the relation Array(2.2.2.∅,x) is satisfied if x is instantiated to

```
Array(Array(Array(x1.x2.∅).Array(x3.x4.∅).∅).
    Array(Array(x5.x6.∅).Array(x7.x8.∅).∅).∅)
```

Let us continue by defining the relation Array_Value(w,index,v). The relation is satisfied when the element in the place referenced by index in the array w has the value v. The relation has the function of an array retrieval if the variable v is uninstantiated.

```
Array_Value(Array(w),n.∅,v) ←
        Pos(n,v,w)
Array_Value(Array(w),n.nn.ref,v) ←
        Pos(n,sw,w),
        Array_Value(sw,nn.ref,v)

Pos(0,u,u.w) ←
Pos(i,v,u.w) ← i > 0, Value(j,i-1),
        Pos(j,v,w)
```

Changes in an array can be described by the relation New_Array_Value(array, index,v,new_array), where new_array is the array array but with the value v in the place indicated by index.

```
New_Array_Value(Array(w),n.∅,v,Array(new_w)) ←
        Ex(n,v,w,new_w)
New_Array_Value(Array(w),n.nn.ref,v,Array(new_w)) ←
        Ex(n,sw,w,new_w),
        New_Array_Value(sw,nn.ref,v,new_w)

Ex(0,v,u.w,v.w) ←
Ex(i,v,u.w,u.new_w) ← i > 0, Value(j,i-1),
        Ex(j,v,w,new_w)
```

As we have seen, the terms of the logic language can be used to construct advanced data structures such as arrays. The computational overhead, however, of using a very general data structure can be very high. The construction of new terms which differ slightly from the old can be very costly. The term representation of the array structure is not a random-access representation. Predicates for creating and manipulating arrays can be built into Prolog. For some uses of these predicates, it is possible for a compiler to produce code performing array references and updates that are as good as those produced by compilers for traditional programming languages. These predicates can be

written completely as Horn clauses without the use of any primitives, as shown above. The goal is to significantly improve the efficiency of some logic programs without sacrificing their logical purity. Examples of predefined relations are:

Array(dim,array)
Array_Value(array,index,value)
New_Array_Value(array,index,value,new_array)

The use of the built-in array facility can enhance the performance of programs using arrays. A special case of an array is the *hash table*[10]. The values are placed in the array according to an indexing function. A hash table is initially set up with a given size, but expands and contracts according to the number of elements to be stored. The hash table permits a wide range of index values but with a size that is relevant to the number of values in the table.

3.6 Exercises

Structures

Exercise 1:
Define the relation Even(number) such that number is expressed as a structure by the constructor S.

Exercise 2:
Define Less_than(lnum,gnum) such that the numbers lnum and gnum are expressed as structures by S. The relation is true when lnum is less than gnum.

List Structures

Exercise 1:
Define a relation Remove(elem,list1,list2) where the two lists list1 and list2 are equal except that list1 also contains the element elem.
Express questions to remove an element from, and to add an element to, a list.

Exercise 2:
Define a relation where the first argument is the number of elements in the list that forms the second argument.

Exercise 3:
Define a relation Insert(k,u,x,y) in which two lists x and y are equal except that y is to have one more element which shall be u and which shall be in the kth position in y.

---

[10] See also Sect. 8.2.

Express questions to insert an element into the kth position in a list and to remove the kth position from a list.

Exercise 4:

Express a relation Quicksort(list,slist) in which slist is the sorted permutation of list. Quicksort differs from Sort in (111) and (112) in that parts of the list are sorted and then put together, instead of the elements being inserted one by one into a sorted list.

First define a predicate Partition(elem,list,list1,list2) that separates list into two lists list1 and list2 so that all elements which are less than, or equal to, elem are inserted into list1 and all that are greater are inserted into list2. Then express the relation Quicksort using Partition and Append. Partition separates restlist into two shorter lists with elem as the discriminating element, and, finally, the sorted smaller lists and elem are put together to form the sorted list slist.

Tree Structures

Exercise 1:

Define the relation Subtree(node,tree,ltree,rtree) in which tree is a binary tree and node is a node in the tree. The variables ltree and rtree represent the left and the right subtrees of node.

Exercise 2:

Define a relation Motherhood(root,node,tree) in which root is the mother-node of node in the binary tree tree.

Exercise 3:

Define a relation Append for binary trees in analogy with the Append relation for lists.

Exercise 4:

Define the relation Remove(tree1,node,tree2). The variables tree1 and tree2 shall both be ordered binary trees with the same nodes, except that node is not to be found in tree2.

Difference lists

Exercise 1:

In the program Quicksort an Append relation is used to form a sorted list from two sorted sublists. To append lists, the program Append has to traverse the first list element by element. This is very inefficient for a program like Quicksort. Rewrite the program with a d-list as the second argument of Quicksort so the Append operation can be simplified.

Express Quicksort without Append by using a d-list as the second argument.

Exercise 2:

Express the program Pre_order(tree,dlist) that, given a tree, will put the nodes in a d-list. The tree shall be traversed in pre-order.

Exercise 3:

Express the substitution set for the two expressions below:

Append(D(A.B.C.v,v),D(D.E.F.w,w),D(I,ls))
Append(D(x,y),D(y,z),D(x,z))

Array Structures

Exercise 1:

Transform a solution of the knight's tour problem (for example the one given in Round 2) to a solution representing the chess board with an array.

Round 4. Databases and Expert Systems

4.1 Metalevels

4.1.1 Open and Closed World

Let us make a visit to the feline world where we can make the following world description:

$$\begin{aligned}
&\mathsf{Feline(x)} \leftarrow \mathsf{Cat(x)} && (130)\\
&\mathsf{Feline(x)} \leftarrow \mathsf{Big_cat(x)} && (131)\\
&\mathsf{Feline(x)} \leftarrow \mathsf{Cheetah(x)} && (132)\\
&\mathsf{Cat(x)} \leftarrow \mathsf{Domestic_cat(x)} && (133)\\
&\mathsf{Cat(x)} \leftarrow \mathsf{Wildcat(x)} && (134)\\
&\mathsf{Big_cat(x)} \leftarrow \mathsf{Lion(x)} && (135)\\
&\mathsf{Big_cat(x)} \leftarrow \mathsf{Tiger(x)} && (136)\\
&\mathsf{Cheetah(x)} \leftarrow \mathsf{Hunting_leopard(x)} && (137)\\
&\mathsf{Hunting_leopard(Juba)} \leftarrow && (138)\\
&\mathsf{Tiger(Shere\text{-}Khan)} \leftarrow && (139)\\
&\mathsf{Tiger(Amur)} \leftarrow && (140)\\
&\mathsf{Lion(Leona)} \leftarrow && (141)\\
&\mathsf{Domestic_cat(Tom)} \leftarrow && (142)
\end{aligned}$$

The statements (130) – (142) form an *open world.* It is possible that not all information is included in our description. We do not know, for instance, if only the relations Cat, Big_cat, and Cheetah together make up Feline. There might be more groups belonging to the felines but not included in our world.

Let us formulate (130), (131), and (132) in standard form.

$$\forall x\,(\mathit{Feline}(x) \leftarrow \mathit{Cat}(x) \vee \mathit{Big\_cat}(x) \vee \mathit{Cheetah}(x))$$

When we complete the above expression in standard form with "if-and-only-if" instead of "if", we get

$$\forall x\,(\mathit{Feline}(x) \leftrightarrow \mathit{Cat}(x) \vee \mathit{Big\_cat}(x) \vee \mathit{Cheetah}(x))$$

We now have a *closed world* for the predicate Feline. All information is included in a closed world. Now we know that a feline has to be either a cat or a big cat, or a cheetah. There are no other possibilities.

Another way of defining a closed world is to study the statements and then declare that the only instances of Feline are those expressed by (130), (131), and (132). When we study objects and describe objects in any language other than the *object language*, i.e. the language in which the objects are described, we say that we study the objects at a *metalevel* and express ourselves in *metalanguage*. In this case our objects are Horn clauses and our metalanguage is natural language. Hence we can express, in the object language together with the metalanguage, that felines can be only cats, big cats, or cheetahs.

(130), (131), and (132) define all instances of Feline(x) (143)

$\forall x\,(Feline(x) \leftrightarrow Cat(x) \vee Big\_cat(x) \vee Cheetah(x))$, and (130), (131), (132), and (143) now all express the same thing, but in the latter case we have to use two levels for the description. The expression (143) cannot be expressed in the object language.

We want all information to be included in our Prolog world, i.e. it shall be a closed world. One way, which we saw above, is to specify our facts as equivalences. If we have the equivalence $A \leftrightarrow B \vee C$ we can transform it into $(A \rightarrow B \vee C) \wedge (A \leftarrow B \vee C)$. We represent the expressions in clausal form[1]. The first expression corresponds to one clause and the latter to two clauses.

B, C ← A
A ← B
A ← C

We notice that B, C ← A is not a Horn clause, nor can we transform the clause into one. Thus we cannot try to represent equivalences in Prolog programs with the help of transformations into Horn form. This means that we cannot achieve a closed world in Prolog unless we use a metalanguage. At the metalevel in the Prolog system we can make a *closed world assumption*. All information is assumed to be included in the Prolog world.

On the other hand, if we choose to have an open world, we can make an *open world assumption*. What assumption we make has a bearing on what conclusions we can draw. Regardless of whether we have an open or a closed world, we can draw the same conclusions about Feline. But if we ask the question ← Domestic_cat(Garfield) we get different answers depending on our assumption about the world. If the world is closed, the answer will be "no" as we do not have a clause Domestic_cat(Garfield) ← and all information is contained in our world. If the world is open, we cannot reply to the question. Garfield may be a domestic cat although our world has no information about it. We only know that Domestic_cat(Garfield) cannot be deduced from the premises.

[1] See Round 1.

4.1.2 Negation

We know from (138) that Juba is a hunting leopard. Hence it is possible to draw the conclusion, with the help of (137), that Juba is a cheetah. If we know that Elsa is not a hunting leopard, can we draw the conclusion that Elsa is not a cheetah? It is not possible from (137). But after looking up the word cheetah in an encyclopedia, we know that an animal is a cheetah if, and only if, it is a hunting leopard.

$$\forall x\,(\mathit{Cheetah}(x) \leftrightarrow \mathit{Hunting\_leopard}(x))$$

Now it is possible to demonstrate by a resolution proof that Elsa is not a cheetah. We must first transform the above expression to Horn form and formulate the information that Elsa is not a hunting leopard.

$$\begin{array}{ll} \mathsf{Cheetah(x)} \leftarrow \mathsf{Hunting\_leopard(x)} & (137)\\ \mathsf{Hunting\_leopard(x)} \leftarrow \mathsf{Cheetah(x)} & (144)\\ \leftarrow \mathsf{Hunting\_leopard(Elsa)} & (145) \end{array}$$

We want to show that Elsa is not a cheetah, i.e. $\leftarrow$ Cheetah(Elsa) and so we assume the opposite since resolution is founded on proof by contradiction. The opposite is

$$\mathsf{Cheetah(Elsa)} \leftarrow$$

A bottom-up resolution proof can be formulated as follows:

$$\dfrac{\dfrac{\mathsf{Cheetah(Elsa)} \leftarrow \qquad \mathsf{Hunting\_leopard(x)} \leftarrow \mathsf{Cheetah(x)} \quad (144)}{\theta_1 = \{\langle \mathsf{x,Elsa}\rangle\}} \qquad \mathsf{Hunting\_leopard(Elsa)} \leftarrow \qquad \leftarrow \mathsf{Hunting\_leopard(Elsa)} \quad (145)}{\leftarrow}$$

Since our assumption yielded a contradiction, the negation of the assumption is true, i.e. $\leftarrow$ Cheetah(Elsa) holds.

We can now observe that to show that an animal is a cheetah, the "if" part of the definition is sufficient, i.e. $\forall x\,(\mathit{Cheetah}(x) \leftarrow \mathit{Hunting\_leopard}(x))$. But to demonstrate the negation, that an animal is not a cheetah, the "only-if" part is necessary, i.e. $\forall x\,(\mathit{Cheetah}(x) \rightarrow \mathit{Hunting\_leopard}(x))$. Generally, we cannot represent "only if" in a Prolog program, as we saw in Sect. 4.1.1. Nor can we express (145) in a program because the clauses there are limited to Horn clauses.

Still, we want to be able to represent negation somehow. One way is to define new predicates Non_Hunting_leopard and Non_Cheetah instead of trying to negate the predicates Hunting_leopard and Cheetah.

Non_Hunting_leopard(Elsa) ← (146)
Non_Cheetah(x) ← Non_Hunting_leopard(x) (147)

This is one possible method of solving the negation problem but it is a very awkward one. We have to define the negated variant of all the predicates we believe to be necessary for the deduction. This would naturally lead to longer programs. Another drawback to this solution is that the predicate Non_Cheetah is an entirely new predicate with no logical connection to Cheetah.

Another way of representing negation is to use information at the metalevel in the same way as we did in the previous section. At the metalevel we assume the world in our Prolog program to be a closed world, i.e. we have all the information about the world. Hence it is possible to draw the conclusion, at the metalevel, that if we cannot demonstrate that a question is true, then the question is false. This is called *negation as failure.*

We have chosen to use "negation as failure" in our Prolog system to solve the negation problem. We use a built-in construction Not with a metalevel significance. Not can only be used in front of atomic formulas. If we ask the question ← Not P(x) we may have three different results:

- If the Prolog system cannot derive P(x) the question succeeds.
- If P(x) succeeds without any variable being bound, the question fails.
- If P(x) can succeed only by binding the variable x during the derivation, an execution error will follow.

We illustrate this by a few questions to the system. We shall first ask if there is any domestic cat by the name of Garfield.

```
← Domestic_cat(Garfield)
no
```

The program contains no Horn clause Domestic_cat(Garfield) ← and the answer is no.

Next question is: is it true that there is no domestic cat called Garfield?

```
← Not Domestic_cat(Garfield)
yes
```

We have no clause Domestic_cat(Garfield) ← in our world and so we fail to show Domestic_cat(Garfield). Since we have assumed in Prolog that we possess all information about the world, we know the names of all domestic cats. Garfield is not among these and the Prolog system concludes that Garfield is not a domestic cat and thus the negation is true.

Then we ask if it is true that there is no domestic cat by the name of Tom.

```
← Not Domestic_cat(Tom)
no
```

Since Domestic_cat(Tom) ← is actually in the program, we manage to show Domestic_cat(Tom) and thus Not Domestic_cat(Tom) cannot be true.

We ask if it is true that there is somebody who is not a domestic cat.

```
← Not Domestic_cat(x)
```

This question leads to an execution error because Domestic_cat(x) can be shown only if x is instantiated. We know one domestic cat, namely, Tom. But we know nothing to contradict the assertion that all are domestic cats. The Prolog system cannot decide whether the question is true or false. But should we have the statement in the program that all individuals are domestic cats, Prolog would be able to answer the question above.

$$\text{Domestic\_cat(x)} \leftarrow \tag{148}$$

```
← Not Domestic_cat(x)
no
```

Let us ask if it is true that there is no feline by the name of Garfield.

```
← Not Feline(Garfield)
yes
```

The Prolog system cannot show that Garfield is a feline and so the answer to the question will be positive.

"Negation as failure" and logic negation are not equivalent from a logic point of view, but it can be proven that "negation as failure" is sound in relation to logic negation[2]. This means that one can prove that "negation as failure" yields only logically correct answers. But "negation as failure" cannot give all the answers that are logically correct. This distinction can be illustrated by the following example. We want to show that P holds if we know that $P \leftarrow \neg P$ is true. An implication $A \leftarrow B$ can be transformed into $A \vee \neg B$. $\neg\neg A$ is the same as A. The symbol $\Leftrightarrow$ means that the expression in front of the sign is equivalent to, has the same logical meaning as, the expression following the sign.

$$P \leftarrow \neg P \quad \Leftrightarrow \quad P \vee \neg\neg P \quad \Leftrightarrow \quad P \vee P \quad \Leftrightarrow \quad P$$

Thus, if we have the premise $P \leftarrow \neg P$, we can prove logically that P is true. But in the following Prolog program it is impossible to demonstrate P ← .

$$\text{P} \leftarrow \text{Not P} \tag{149}$$

We ask the question ← P to (149). To show that Not P is true, the Prolog system must first try to demonstrate P. To achieve this the system must demonstrate Not P, etc. The Prolog system ends up in an infinite evaluation chain and thus cannot prove that P holds.

---

[2] K.L. Clark, "Negation as Failure." in H. Gallaire and J. Minker (eds.), *Logic and Data Bases*, Plenum Press, New York, 1978.

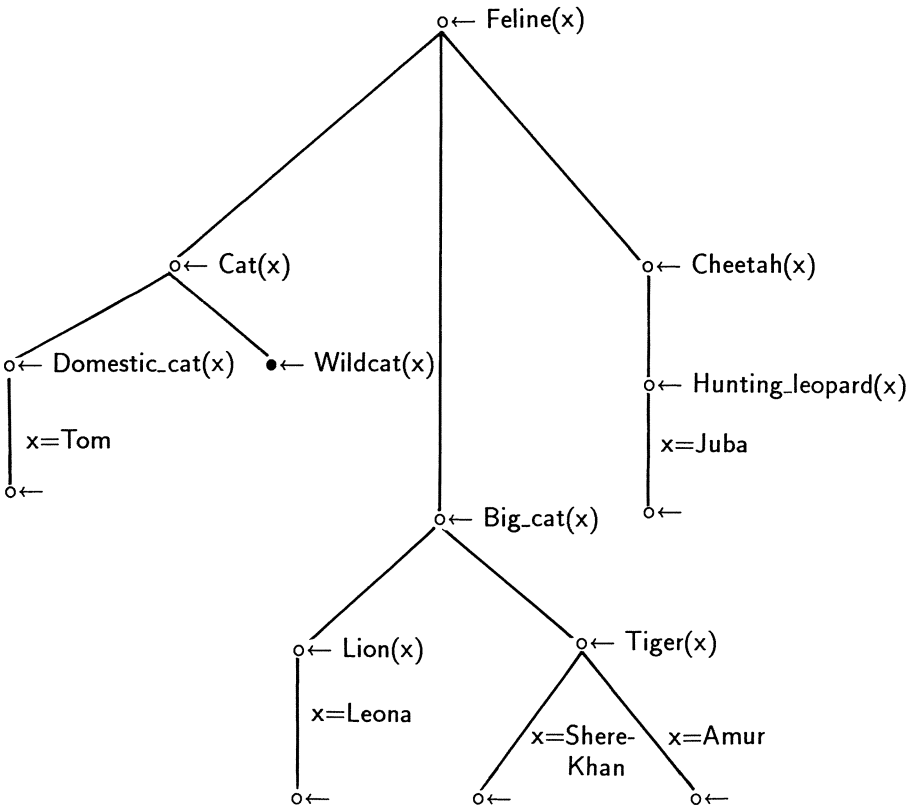

Figure 4.1: Top-down search space for Feline(x).

4.1.3 All Answers

Our questions to the Prolog system have been in the form

do there exists $x_1 ... x_n$ so that P is true?

So far we have had one solution as the answer. However, we may also be interested in all the answers to a question, i.e. in *exhausting the search space.*

We are interested in an exhaustive search of the search space for felines. What individuals in our set (130) – (142) are felines? We construct a search space top-down for the question ← Feline(x) (see Figure 4.1). All valid conclusions about felines are Feline(Tom), Feline(Leona), Feline(Shere-Khan), Feline(Amur), and Feline(Juba).

In Prolog we can, of course, also ask who is a feline. Once we have been given one answer, we can ask the system for an alternative solution. We can go on asking until the Prolog system replies that there are no more alternatives and the search space is then exhausted.

Let us use the Prolog system to find all the felines in our world.

```
← Feline(x)
x = Tom ;          ";" to find an alternative answer
x = Leona ;
x = Shere-Khan ;
x = Amur ;
x = Juba ;
no                 no more alternatives
```

The above is a complicated way of getting alternative information, especially when there are many alternative answers. Moreover, we have access to only one solution at a time. It is sometimes practical to have access to all the alternatives at the same time. Our Prolog system therefore has the built-in metaconstruction All_answers that for a certain question tries to find all alternative substitutions for the variables which will verify the question, and assembles them in a list. Given a question P(x), we may be interested, for instance, in all substitutions for the variable x. All substitutions for x are gathered in a list list_of_x if we ask the Prolog system the following,

All_answers(x,P(x),list_of_x)

in which x is a term. x may be a variable or a structure of variables and constants. The question P(x) may be composed of several subquestions. If there is no x satisfying P(x), the answer from the Prolog system will be "no". The variables in the question are implicitly universally quantified. This means that a variable which was instantiated before All_answers is permitted in the question P.

Once again we ask what felines we have, but this time using All_answers.

```
← All_answers(x,Feline(x),list_of_felines)
list_of_felines = Tom.Leona.Shere-Khan.Amur.Juba.∅
x = _          x is not instantiated
yes
```

We want to know what big cats we have.

```
← All_answers(x,Big_cat(x),list_of_big_cats)
list_of_big_cats = Leona.Shere-Khan.Amur.∅
x = _
yes
```

What wildcats do we have in the database?

```
← All_answers(x,Wildcat(x),list_of_wildcats)
no
```

We have no wildcats so the answer is no. Let us find all pairs of tigers.

```
← All_answers(Pair(x,y),(Tiger(x),Tiger(y)),
```

```
                    list_of_pair_of_tigers)
list_of_pair_of_tigers = Pair(Shere-Khan.Shere-Khan).
                         Pair(Shere-Khan,Amur).
                         Pair(Amur,Shere-Khan).
                         Pair(Amur,Amur).∅
x = _
y = _
yes
```

We want all pairs of tigers, but we do not want a tiger to form a pair with itself.

```
← All_answers(Pair(x,y),(Tiger(x),Tiger(y),x ≠ y),
                    list_of_pair_of_tigers)
list_of_pair_of_tigers = Pair(Shere-Khan,Amur).
                         Pair(Amur,Shere-Khan).∅
x = _
y = _
yes
```

The felines in our world belong to different zoos. We define a new predicate that states in which zoo an animal is to be found and the price the zoo had to pay for it.

```
Zoo(Bronx_Zoo,Shere-Khan,30000) ←
Zoo(Childrens_Zoo,Juba,20000) ←
Zoo(Childrens_Zoo,Tom,500) ←
Zoo(Central_Park_Zoo,Leona,30000) ←
Zoo(Bronx_Zoo,Amur,28000) ←
```

We can now ask the Prolog system which animals are in the different zoological gardens. We want a list of elements where each element is the name of the zoo and the animal which is there.

```
← All_answers((zoo,animal),Zoo(zoo,animal,price),zoo_list)
zoo_list = (Bronx_Zoo,Shere-Khan).(Central_Park_Zoo,Leona).∅
price = 30000
zoo = _
animal = _;

zoo_list = (Childrens_Zoo,Juba).∅
price = 20000
zoo = _
animal = _
yes
```

We can get the remaining answers if we keep typing ";", but we content ourselves with these. The result has not turned out the way we specified above, however. We wanted all the zoological gardens linked to the animals in the answer. If we investigate the answers we find that the variable price has been instantiated. This means that the Prolog system is trying to find more instances of animals and zoological gardens at a particular price. But we want the combination of park and animal, independent of the price. We can have it if we state that there exists a price such that Park(zoo,animal,price) is true. There exists an x such that P(x) we render as

$$x\hat{}P(x)$$

Now we can reformulate the question.

```
← All_answers((zoo,animal),
                price^(Zoo(zoo,animal,price)),zoo_list)
zoo_list = (Bronx_Zoo,Shere-Khan).(Childrens_Zoo,Juba).
           (Childrens_Zoo,Tom).(Central_Park_Zoo,Leona).
           (Bronx_Zoo,Amur).∅
price = _
zoo = _
animal = _
yes
```

It is even better to link a park to a list in which all the animals in the park are given. It is possible if we first collect all animals in a park in a list animal_list and then tie it to the animal park and let the system find all such alternatives and save them in zoo_list.

```
← All_answers((zoo,list_of_animals),
              All_answers(animal,
                          price^(Zoo(zoo,animal,price)),
                          list_of_animals),
              zoo_list)
zoo_list = (Bronx_Zoo,Shere-Khan.Amur.∅).
           (Childrens_Zoo,Juba.Tom.∅).
           (Central_Park_Zoo,Leona.∅).∅
price = _
zoo = _
list_of_animals = _
animal = _
yes
```

We can examine what the parks have had to pay for the animals and use a structure Price for this purpose.

```
← All_answers(Price(price,animal),
              zoo^(Zoo(zoo,animal,price)),pricelist)
```

```
zoo = _
pricelist = Price(30000,Shere-Khan).Price(20000,Juba).
            Price(500,Tom).Price(30000,Leona).
            Price(28000,Amur).∅
price = _
animal = _
yes
```

With All_answers we may get duplicates in the response list if the variables can be instantiated by the same value in alternative evaluations. If we would like to avoid duplicates and, moreover, have our elements in the response list sorted, we can use the built-in construction

Sorted_answers(x,P(x),list_of_x)

Let us ask how much the animals have cost and have the response sorted after the price on an ascending scale.

```
← Sorted_answers(Price(price,animal),
                 zoo^(Zoo(zoo,animal,price)),pricelist)
zoo = _
pricelist = Price(500,Tom).Price(20000,Juba).
            Price(28000,Amur).Price(30000,Leona).
            Price(30000,Shere-Khan).∅
price = _
animal = _
yes
```

In Sect. 4.2 on databases there are further examples on how to use the metaconstruction All_answers.

4.2 Databases

4.2.1 Introduction

A collection of information, or data, can be called a *database*. Besides viewing a Prolog program as a set of premises from which we draw conclusions, we can also view it as a database from which we ask questions. A database can be used to answer questions asked in a *query language*. For Horn form databases we do not have to create a special query language, as we can use the Horn form directly. Thus we avoid transformation problems between the query language and the descriptional language. We can, of course, easily add a query language if the database user wants a custom-made communication.

Information on objects can be represented in several different ways in a Prolog program, using clauses or data structures. Let us first look at the situation when we represent all information using clauses. The information is given in

the form of unconditional clauses, i.e. *facts* about the world and by conditional clauses giving *rules* for how to deduce facts. A Prolog program can be called a *deductive database*, because we deduce facts from the database. The information in the database is not changed — the database is static. In the real world, however, it is usual for information to change, the world is dynamic. This is why we want to be able to remove obsolete information, change the information, and add new information to the database. We need to look at the database from a metalevel to be able to carry out these operations, i.e. we treat the database as an object and manipulate it with the help of metaconstructions.

Another possibility is to represent the database as an object in the form of a data structure. We define programs for searching the structure, and for adding, changing, or removing information from the data structures. We need definitions of

- a relation for membership
- relations defining the connection between a data structure and a structure with one element less or one element more than the first data structure
- relations defining the connection between a data structure and a structure in which one of the nodes has been changed

4.2.2 Data as Relations

When we define the database using clauses, we can compare a Prolog database to a relational database. We have a generalization of the relational model since we can use recursion to express rules in the database.

A relational database and a Prolog database both consist of a number of relations. We can view these relations as arrays. A relation with n arguments and m clauses forms a table with m rows and n columns. A relation with arity n is called an n-ary relation. All values in a certain argument position, i.e. in a certain column in the table, are taken from the same domain. A relation is a subset of the cartesian product of the domains from which the attributes are taken. When we want to build up a database we start from objects we are interested in storing and their various attributes.

We can, for example, create a database with the objects cars, persons, brands, and colors. Persons and cars may be related to each other in an ownership relation. One or several people are owners of a certain car. One person may also own several cars. A car is of a certain brand and has a certain color. Several cars may be of the same brand and several cars may have the same color. Information about a certain car can be represented in a table with columns for license number, owner, brand, and color. We can describe a car with the help of the relation Car with four arguments. An instance of the relation may look as follows:

Car(ABC123,Jones,Volvo,Green) ←

Each column corresponds to an attribute of a car. The order of the arguments in the relation is important. We can refer to an attribute by its relative position in the relation or we can name the columns. An attribute that unequivocally identifies a certain instance of a relation is called the *key* or *identification* of the relation. A car may also be represented by a set of relations Class, Owner, Brand, and Color, instead of a single relation Car. These relations all have two arguments and are therefore binary relations.

Class(ABC123,Car) ←
Owner(ABC123,Jones) ←
Brand(ABC123,Volvo) ←
Color(ABC123,Green) ←

The terms in relations thus stand in a relationship to each other. The relations between the terms can be of different types. A married person, for instance, has only one spouse; the relation is of the *one-to-one* type. For a given first argument there is only one second argument that satisfies the relation.

Married(Freya,Odin) ←
Married(Siv,Thor) ←
Married(Gerd,Frey) ←

Values for the first column of the relation are taken from the domain of all women and the values for the second column from the domain of all men. Suppose that the domain of all women is Freya, Siv, Gerd and the domain of all men is Odin, Thor, Frey. The cartesian product of these domains is

Column 1	**Column 2**
Freya	Odin
Freya	Thor
Freya	Frey
Siv	Odin
Siv	Thor
Siv	Frey
Gerd	Odin
Gerd	Thor
Gerd	Frey

Out of these combinations it is only the first, the fifth, and the ninth that are included in the relation. We notice that the relation is a subset of the cartesian product.

Let us return to a study of various types of relations. A child has only one mother but a mother may have several children. The relation Mother which we have used, has the child as its first argument. The relation is thus of the *many-to-one* type, if we read it proceeding from the first argument.

Mother(Thor,Earth) ←
Mother(Ve,Bestla) ←

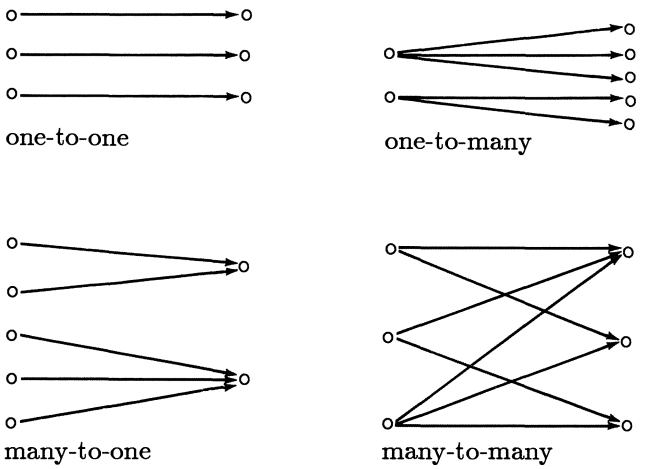

Figure 4.2: Types of relations.

Mother(Vile,Bestla) ←
Mother(Odin,Bestla) ←

A firm may have many employees but it is less usual for someone to be employed in several firms. A relation between the firm and an employee is then of the *one-to-many* type.

Employee(FAB_Inc,Smith) ←
Employee(FAB_Inc,Jones) ←
Employee(FOA_Co,Brown) ←

A car may be owned by more than one person and one person may own several cars. An ownership relation is thus of the *many-to-one* type. The various types of relations are illustrated in Figure 4.2.

Owner(NM2524,Smith) ←
Owner(NM2524,Wright) ←
Owner(XY2789,Smith) ←

Let us look at a simple database example. We will explore the world of delicatessen-type restaurants. Questions about delis that we may want to have answered include: Where is this deli? What are the business hours? Do they have a garden or sidewalk café? Do they serve ice cream? We can describe all this information in a relation Deli. We need to be able to identify the different delis to answer the questions. We will use the name as the key of the relation. Thus, the relation deli needs six arguments: one for the key and five for the other attributes necessary to answer the questions. We must decide how to interpret the Deli relation. We can make the layout in Figure 4.3a to state the interpretation of the arguments in the Deli-relation.

Argument No.	Interpretation
1	name
2	address
3	opening time
4	closing time
5	garden
6	ice cream

Figure 4.3a: Intended interpretation of the arguments to the Deli-relation.

```
Deli('Ovens of Brittany', 'State St.',          '6.45am' 10pm, No,  No) ←
Deli('Memorial Union',    'Langdon St.',        7am,     10pm, Yes, Yes) ←
Deli('Mothers',           'Williamson St.',     8am,     1pm,  No,  Yes) ←
Deli('Ellas Deli',        'E. Washington Ave.', 10am,    11pm, Yes, Yes) ←
Deli('Bernies',           'University Ave.',    8am,     1pm,  No,  Yes) ←
```

Figure 4.3b: The Deli-relation.

We have six domains $D_1, \ldots, D_6$ of objects. D_1 is the domain of all names, D_2 all addresses, D_3 and D_4 are both the domain of hours and finally, D_5 and D_6 have the values Yes and No as their domain. The relation Deli is a subset of the cartesian product $D_1 \times D_2 \times \cdots \times D_6$.

We can describe some delis in a midwestern university town with the help of the Deli relation, see Figure 4.3b. We can view the lay-out as a table with six columns and five lines. The order of the clauses is irrelevant but the order of the arguments is important.

We can ask questions from the database as usual. If we want to find a deli with a garden that serves ice cream we have to ask the question

```
← Deli(name,address,opens,closes,Yes,Yes)
name = 'Memorial Union'
address = 'Langdon St.'
opens = 6.45am
closes = 10pm
```

To find two delis located on the same street we ask the question

```
← Deli(name1,address,o1,c1,g1,i1),
  Deli(name2,address,o2,c2,g2,i2)
no
```

Should we want all possible answers to a question instead of only one, we construct the question with the predefined metaconstruction All_answers. We asked for a deli with a garden that served ice cream and obtained this conclusion from the database.

$$Deli('Memorial\ Union','Langdon\ St.', 7am, 10pm, Yes, Yes)$$

The following existentially quantified expression is thus true.

$$\exists name \exists address \exists opens \exists closes\ Deli(name, address, opens, closes, Yes, Yes)$$

To find all the delis with a garden that serve ice cream, we can use the All_answers construction. We collect the variables whose instantiation we want to know in a structure in the first argument of the relation All_answers. We want all values for names such that

$$\exists address \exists opens \exists closes\ Deli(name, address, opens, closes, Yes, Yes)$$

is valid. We can say that we want all values for name independent of the values of the variables address, open, closes, by existentially quantifying these in the question. Hence, the question for all the delis with gardens that serve ice cream, will appear like this:

```
← All_answers(name,
                address^opens^closes^
                   (Deli(name,address,opens,closes,Yes,Yes)),
                solutions)
solutions = 'Memorial Union'.'Ellas Deli'.∅
name = _
address = _
opens = _
closes = _
```

Certain questions can be very frequent and we can define rules to facilitate the questioning. We define a rule Open between a point in time and the name of a deli. The relation Open is valid if the time is between the opening and the closing time. When the rule is defined, we only have to remember two arguments to the Open relation.

```
Open(time,deli) ←
        Deli(deli,address,opens,closes,garden,ice),
        open ≤ time, time ≤ closes
```

Should we prefer to avoid the problem of the order of the arguments altogether, we can describe the information using properties and binary relations. We can give the relations mnemotechnical names to facilitate the formulation of questions to the database. The same information as was described in the Deli relation can be described by the following definitions:

```
Address('Ovens of Brittany','State St.') ←
Address('Memorial Union','Langdon St.') ←
Address('Mothers', 'Williamson St.') ←
Address('Ellas Deli','E. Washington Ave.') ←
Address('Bernies','University Ave.') ←
```

```
Opens('Ovens of Brittany',6.45am) ←
Opens('Memorial Union',7am) ←
Opens('Mothers',8am) ←
Opens('Ellas Deli',10am) ←
Opens('Bernies',8am) ←

Closes('Ovens of Brittany',10pm) ←
Closes('Memorial Union',10pm) ←
Closes('Mothers',1pm) ←
Closes('Ellas Deli', 11pm) ←
Closes('Bernies',1pm) ←

Garden('Memorial Union') ←
Garden('Ellas Deli') ←

Ice('Memorial Union') ←
Ice('Mothers') ←
Ice('Ellas Deli') ←
Ice('Bernies') ←
```

We can formulate the questions about gardens and ice cream in this way

```
← Garden(name), Ice(name)
name = 'Memorial Union'
```

The question for two delis on the same street is rendered like this

```
← Address(name1,address), Address(name2,address)
```

The answer to this question will be all delis that serve ice cream and serve outdoors.

```
← All_answers(name,(Garden(name),Ice(name)),solutions)
solutions = 'Memorial Union'.'Ellas Deli'.∅
name = _
```

If we want to find all delis that open at the same time, we cannot existentially quantify the argument representing opening hours or we will have all delis regardless of the opening time. If we thus leave y unquantified, we will have three sets of answers, one for every opening time in the database.

```
← All_answers(name,Opens(name,y),Solutions)
solutions = 'Ovens of Brittany'.∅
y = 6.45am;        more solutions?

solutions = 'Memorial Union'.∅
y = 7am;           more solutions?
```

```
solutions = 'Mothers'.'Bernies'.∅
y = 8am;          more solutions?

solutions = 'Ellas Deli'.∅
y = 10am;         more solutions?

no
```

We can assemble all these different answers using a new All_answers construction so that the elements in the answer list consist of a structure with the constructor Opening_time whose first argument is an opening time and whose second argument is a list of names of those delis that open at this hour.

```
← All_answers(Opening_time(y,solutions),
              All_answers(name,Opens(name,y),solutions),
              solutionlist)
solutions = _
name = _
y = _

solutionlist = Opening_time(6.45am,'Ovens of Brittany'.∅).
               Opening_time(7am,'Memorial Union'.∅).
               Opening_time(8am,'Mothers'.'Bernies'.∅).
               Opening_time(10am,'Ellas Deli'.∅).∅
```

Suppose that we wish to increase the information about delis. We may feel a need to know the phone numbers of the delis. In the first n-ary representation we will have to add one more argument to the Deli relation. In the latter binary representation we add a new relation Phone with two arguments, the name of the deli and its phone number. In this representation we do not have to change any of the relations already given.

So far we have discussed how we can ask questions of the database. But we also need to make other operations on it. Changes making the database obsolete may very well occur. The database depicts our information at a given moment. New information may be added, the information may change, or information may become invalid. If we look at our deli example, a new deli may open, a deli may move, or change its business hours or a deli may go out of business. We must be able to add new information to the database, change information, and delete information from the database. Prolog systems contain two built-in metaconstructions to change the database: Assert which adds a clause to the database and Retract which removes a clause.

Assert(clause)
Retract(clause)

In both cases the argument has to be instantiated to a program clause when the constructions are evaluated. To add a new deli to the n-ary representation where we use the Deli relation, we write

```
← Assert((Deli('Fess Hotel','E. Doty St.',
                  11am,11pm,No,Yes) ← ))
```

The evaluation always succeeds and there are no alternative ways of evaluating the construction on backtracking.

If we want to remove a deli we write

```
←  Retract((Deli('Mothers',a,o,c,g,i) ← ))
a = Williamson St.
o = 10am
c = 1pm
g = No
i = Yes
```

The first clause in the database that can be unified with the argument of Retract is removed. Retract creates a branch in the search space when several clauses can be matched against the argument of Retract. Thus, if we backtrack, the next matching clause is removed.

We have now changed the database by adding and removing information from it. With the help of the Assert and Retract constructions we can define a relation to change a database relation. We can call the relation Change. The Change relation has two arguments of which the first is the current aspect of the clause we want to change, the second is the new appearance. The definition of Change will be as simple as this

```
Change(old_clause,new_clause) ←
        Retract(old_clause),
        Assert(new_clause)
```

Entire program clauses have to be given as arguments to Change. To avoid this we can define an Update relation which takes the name of the relation to be changed, the key of the relation, and the arity of the relation as its arguments. Moreover, we have to know the position of the argument to be changed and the new value of the argument. In the Update relation we create a predicate from the name of the relation, viz. relation, the key of the relation, and its arity. The predefined relation Structure is used, on one hand, to construct a structure corresponding to a predicate, given a relational name and an argument list, and, on the other hand, given a construction, to separate it into a constructor and an argument list. Examples of these uses of Structure are

```
←  Structure(x,Address,'Ovens of Brittany'.y.∅)
x = Address('Ovens of Brittany',y)
y = _
```

```
← Structure(Opens('Memorial Union',6.45am),x,y)
x = Opens
y = 'Memorial Union'.6.45am.∅
```

The Update relation is a relation between two atomic formulas with the same relational name, the same arity, and the same key. The difference is the value of the argument given by Position.

```
Update(old_clause,new_clause,relation,key,arity,position,value) ←
        Length(arglist,arity), arglist = key.restlist,
        Structure(old_clause,relation,arglist),
        Replaced(position,value,arglist,new_arglist),
        Structure(new_clause,relation,new_arglist),
        Change(old_clause,new_clause)
```

Length is a relation between a list and a number denoting the length of the list.

```
Length(∅,0) ←
Length(elem.list,no) ←
        Value(no-1,previous_no),
        Length(list,previous_no)
```

Replaced is a relation defined for a number representing a position in a list, an element, and two lists. The two lists shall be identical except the element in the given position. elem1 shall be in that position in the second list.

```
Replaced(1,elem1,elem2.list1,elem1.list1) ←
Replaced(pos,elem1,elem2.list1,elem2.list2) ←
        Value(pos-1,previous_pos),
        Replaced(previous_pos,elem1,list1,list2)
```

To change the closing time for Ovens of Brittany we write

```
← Update(old,new,Deli,'Ovens of Brittany',6,3,17.30)
```

4.2.3 Data as Terms

If we think of the operations we want to perform on a database, it is natural to regard the database as an object and represent the information in data structures. In this case we look at the database at a uniform level and avoid bringing two levels into the discussion. In Round 3 we encountered various data structures, such as lists, binary trees, and arrays. The type of data structure that is suitable depends on what type of predicate we wish to define for the structure. Different representations entail differences in length of the derivations.

In the previous section we saw that different types of searches, information changes, and adding and removing information are necessary operations.

Regardless of what data structures we choose, we can represent information as elements in the structure. We can let each element be an n-ary term containing all attributes of a certain object, or we can let the elements be binary terms.

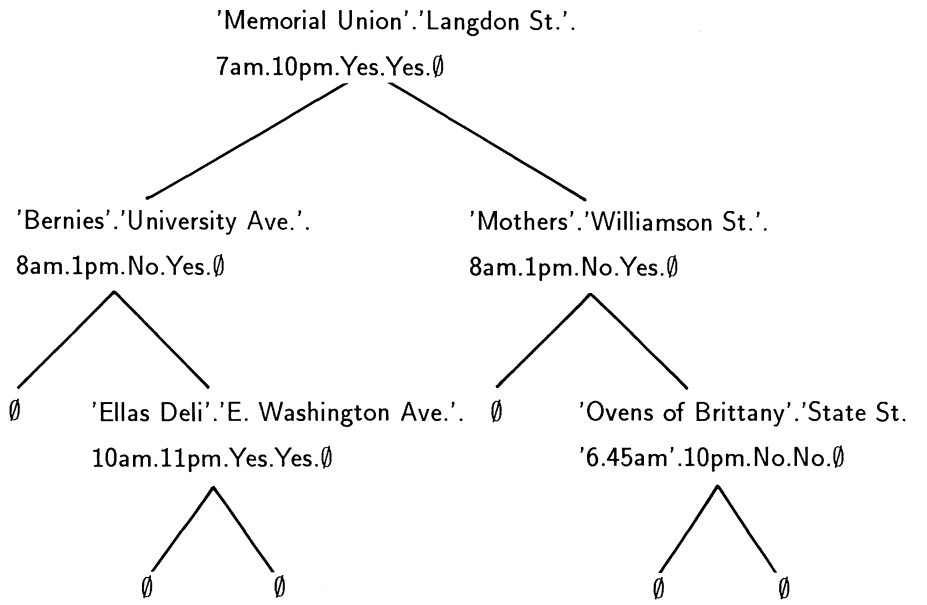

Figure 4.4: The Delirelation represented in a binary tree.

Each relation is represented in a data structure. If we choose to represent the information in n-ary terms, we get a common data structure, and if we choose to represent it in binary terms, we get a structure for each attribute. To add one more attribute entails, in the first case, updating the data structure, and in the second case, increasing the number of data structures.

Delis can be represented as an n-ary term if we connect address, business hours, etc., to the name of the deli. The name of the deli can be used as a key for searching. In this case, when we have information with a certain key, we can order the information after its key value to facilitate searching. We can shorten the search time by choosing suitable search orders and appropriate data structures. A structure well suited for searching ordered sets is the binary tree, as we saw in Round 3.

Let us investigate how we can search for all information about a deli, given the search key. The tree is ordered in inorder. Member_identification is a relation between a search key, all information connected to the search key, and an ordered binary tree. Each deli forms a node in the tree. Let us represent the information on each deli in a list; let the search key be the first element in the list. We can define the relation Member_identification thus:

```
Member_identification(skey,info,T(l,skey.info,r)) ←
Member_identification(skey,info,T(l,key.root,r)) ← skey < key,
        Member_identification(skey,info,l)
```

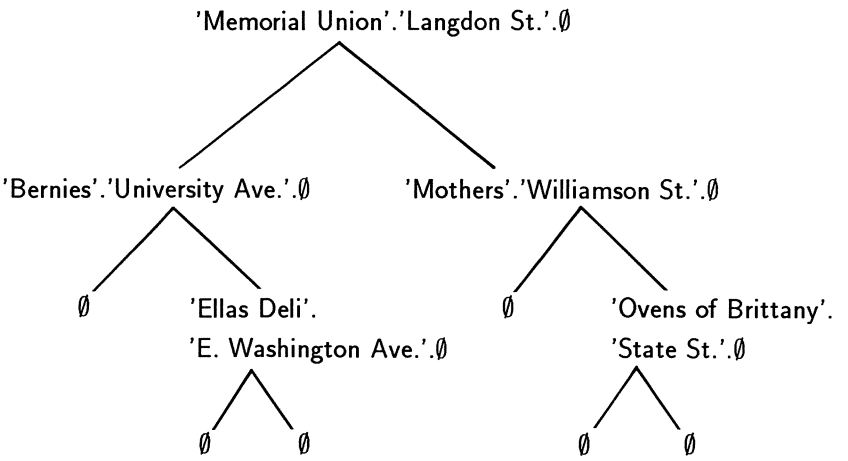

Figure 4.5: The relation Address represented in a binary tree.

Member_identification(skey,info,T(l,key.root,r)) ← skey > key,
Member_identification(skey,info,r)

If we search in a tree that is ordered after a certain key we can achieve an efficient search. But if we prefer to search from some other information given, i.e. search for a deli with a garden that serves ice cream, we get an unordered search. We cannot, when we are at a specific node in the tree, decide that the desired node has to be in the left-hand or the right-hand subtree, and limit our search to that subtree. The node can be in either left or right subtree.

When the searched element has the same structure as the nodes of the tree, and we search for a node that matches the searched element, we can use the relation Member. The searched element can contain variables. If the tree contains information on delis we can search for a deli with a garden that serves ice cream by instantiating the element with the structure name.address.z.w.Yes.Yes.∅. Hence, we use the uninstantiated variables to answer to the question. The tree containing information on delis is denoted by the variable deli_tree. We formulate a question for a deli that serves ice cream and has an outdoor facility in this manner:

← Member(name.address.opens.closes.Yes.Yes.∅,deli_tree)

And the question for two delis on the same street can be rendered like this:

← Member(name1.address.rest1,deli_tree),
Member(name2.address.rest2,deli_tree)

If we choose to represent information on delis in binary terms or unary terms for properties, we can use a set of binary trees. Figures 4.5 and 4.6 display trees representing the address and opening time relations. Each of the properties of having a garden and ice cream can be represented in an ordered binary tree.

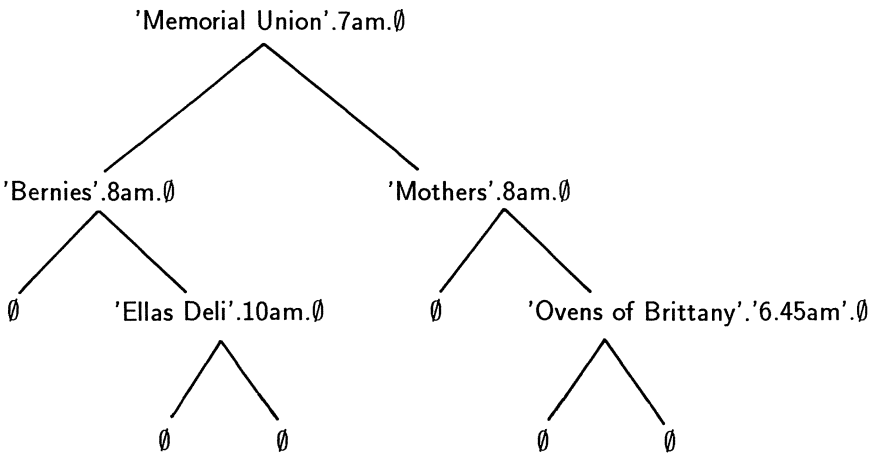

Figure 4.6: The relation Opens represented in a binary tree.

With the help of the membership definition we can now formulate a question for a deli that has ice cream on the menu as well as a garden to serve it in. The variable g_tree represents the binary tree with all gardens and ice_tree, all delis serving ice cream.

```
← Member(name,g_tree), Member(name,ice_tree)
```

To find all delis we can also use the construction All_answers.

```
← All_answers(name,
              (Member(name,g_tree),Member(name,ice_tree)),
              solutions)
```

We can write a program to achieve the same result as All_answers. We define a relation Every_member[3] whose arguments are a binary tree, a value, and a list of all elements in the tree that satisfy this value.

```
Every_member(∅,value,∅) ←
Every_member(T(l,id.value.∅,r),value,solutions) ←
        Every_member(l,value,lsolutions),
        Every_member(r,value,rsolutions),
        Append(lsolutions,id.rsolutions,solutions)
Every_member(T(l,root,r),value,solutions) ←
        root ≠ id.value.∅,
        Every_member(l,value,lsolutions),
        Every_member(r,value,rsolutions),
        Append(lsolutions,rsolutions,solutions)
```

[3] See also the relation Inorder defined in Sect. 3.3. Inorder is a relation between a binary tree, a value, and a list of all elements in the tree.

To add new information, i.e. to enter new nodes into the tree, we can use the relation Insert which is defined in the Round 3. We will also have to remove nodes from the tree, so we shall define a relation Remove between an ordered binary tree, an element, and an ordered binary tree from which the element is missing. The tree is ordered so that we can search for the appropriate node by comparing with the root; the searched element is either in the left or right subtree. When we have found the node that is to be removed, its subtrees have to be reorganized into a tree that can replace the old tree. If one of the subtrees is empty, the tree with the removed element is equal to the second subtree. In the case when we have two non-empty subtrees, we can use an auxiliary relation Reorder for the reorganization.

```
Remove(T(l,root,r),element,T(newl,root,r)) ← element < root,
        Remove(l,element,newl)
Remove(T(l,root,r),element,T(l,root,newr)) ← root < element,
        Remove(r,element,newr)
Remove(T(∅,element,r),element,r) ←
Remove(T(l,element,∅),element,l) ←
Remove(T(l,element,r),element,T(l,root,newr)) ←
        Reorder(r,root,newr)
```

Reorder is a relation between a tree and the tree divided into an element and a tree without the element. The element is the first node in the tree, seen in inorder. If the tree has an empty left subtree we let the root be the element, and the right subtree the smaller tree. In the recursive case, when the left subtree is not empty, we continue traversing this subtree downwards.

```
Reorder(T(∅,element,r),element,r) ←
Reorder(T(l,root,r),element,T(newl,root,r)) ← l ≠ ∅,
        Reorder(l,element,newl)
```

And, finally, we can define a relation to change a node in a tree. The relation New_node has four arguments, the tree with the old node, the node to be changed, the new node, and the tree that contains the changed node.

```
New_node(T(l,info,r),info,element,T(l,element,r)) ←
New_node(T(l,root,r),info,element,T(newl,root,r)) ← node < root,
        New_node(l,info,element,newl)
New_node(T(l,root,r),info,element,T(l,root,newr)) ← root > node,
        New_node(r,info,element,newr)
```

4.3 Expert Systems

4.3.1 Introduction

A database, as we described the concept above, could be a component in an *expert system.* Expert systems are examples of *knowledge-based systems, KBS.*

KBS are often referred to as *rule-based systems.* Knowledge-based systems also provide a means for introducing computer support for such tasks where difficulties in formulating methods of solution in an exact way make traditional computer techniques infeasible. Such a system is entirely dependent on the need to use facts and practical knowledge in a way that resembles the way experts reason. This kind of knowledge is stored in a database in the system.

Let us look at an example of an expert system. MYCIN[4] is one of the best known and also one of the first of the expert systems. It was developed at Stanford University during the 1970s. The system diagnoses infection diseases in the blood and recommends appropriate therapy. MYCIN is used in medical schools in the USA.

Let us follow a part of a MYCIN consultation. The physician's answers are underlined.

Patient's name: Bill Instance

Sex: Male

Age: 42

Have you been able to obtain positive cultures from a site at which Bill Instance has an infection? Yes

What is the infection? Primary-Bacteremia

Please give the date and approximate time when signs or symptoms of the primary-bacteremia (called INFECTION-1) first appeared? May 5, 1987

From what site was the specimen for the positive culture (CULTURE-1) taken? Blood

Please give the date and time when this blood culture (CULTURE-1) was obtained. May 9, 1987

The first significant organism from this culture will be referred to as ORGANISM-1. Enter the identity of ORGANISM-1. Unknown

Is the ORGANISM-1 a rod or a coccus? Rod

.

.

My recommendations will be based on the following probable infections and potential causative organisms ... (enumeration of six different items)

On a scale of 0 to 4, where higher numbers indicate increasing severity how would you rate Bill Instance's degree of sickness? 3

Does Bill Instance have a clinically significant allergic reaction to any antimicrobial agent? No

Bill Instance's weight in kilograms. 70

---

[4] B.G. Buchanan and E.H. Shortliffe, *Rule-based Expert Systems*, Addison-Wesley, Reading, MA, 1985.

My preferred therapy recommendation is as follows:
In order to cover for Items 1, 2, 3, 5 and 6 give Gentamycin 119 mg (1.7 mg/kg) for 10 days.
Modify dose in renal failure.
In order to cover for Item 4 give Clindamycin 595 mg (8.5 mg/kg) for 14 days.
If diarrhoea develops, patient should be evaluated for possible pseudomembranous colitis.

Do you want to see the next choice therapy? No

The system interacts with the user in a "natural" way and reaches the diagnosis with the help of the physician's replies. The system was developed as a support for the physician and not intended to supplant him or her. The responsibility for diagnosis and therapy still rests with the physician.

An estimation of the MYCIN system showed that it has achieved results that to 90 % were comparable to what a panel of physicians was able to recommend. One of the motives for developing the system was the fact that one fourth of the population of the United States is treated with penicillin each year and of these cases nine out of ten are believed to be unnecessary.

Let us look at some characteristics of knowledge-based systems in general. A knowledge-based system should be able to store a *knowledge database* containing knowledge of a specific domain. This knowledge can be divided into facts, heuristic knowledge, and rules of reasoning. The mechanism a KBS uses for drawing conclusions from facts using rules is called *inference mechanism.* It should imitate the human inference process in so far that the conclusions a human draws on the basis of knowledge about facts and rules should be the same as conclusions which the inference mechanism produces on the basis of stored knowledge about these facts and rules. Thus it is not necessary that the reasoning be performed in a similar way. It is essential, however, that the knowledge database and the inference mechanism be separate.

A further property of the type of problem domain a KBS may handle is the fact that the knowledge about the domains is very often vague. There is an uncertainty as to whether a fact or a rule holds. The system should therefore have some sort of *certainty handling.* The conversation with the user should be carried out in a natural way; it should *conduct the dialog in the terminology of the domain.* A KBS should be able to *explain its conclusions* and justify its choice of a certain question to the user. And finally, it should be *simple to modify the knowledge* in the knowledge data base.

Applications where the expert system technique is suitable are characterized by the fact that the technical knowledge in the area, the domain, can be expressed only to a limited extent in a well-structured, algorithmic manner. Let us look at some examples of tasks requiring expert knowledge, where expert

systems might provide good support, and some applications[5] with such aims.

- *Diagnosis*
  Identification of the reason for a pathological state and recommendation of a therapy. Examples of such systems are MYCIN, mentioned above, INTERNIST, which diagnoses diseases in internal medicine and can also handle concurrent diseases, and CASNET, which models the course of disease of glaucoma and gives therapy recommendations.
- *Interpretation*
  Evaluation of uncertain, vague, erroneous data, etc. and interpretation of their meaning. PROSPECTOR is a system for geological prospecting that supports the evaluation of what minerals are likely to be found in a certain area.
- *Supervision*
  Control and supervision of processes where the problem consists in discovering when an abnormal state occurs that demands some kind of action and perhaps control of the action to be taken. The VM expert system supervises a patient's respirator and changes the oxygen flow depending on any change in the patient's state.
- *Design*
  Design of a solution or construction that satisfies given requirements and restrictions. DENDRAL is frequently used by American chemists as an aid for deciding the structure of molecules with the help of a spectrogram. Another example is XCON which configures VAX computers according to customer demands and the technical requirements that exist for the computer system.
- *Planning*
  Design of a strategy that can be carried out in order to reach a given or wanted goal with limited resources. The FOLIO system gives recommendations to stockbrokers on what economic goals should be set for a client and what the client's portfolio should contain for these goals to be reached. TAXADVISOR assists a lawyer with tax and estate planning for clients with large estates.

Facts within the domain in question are stored in the *knowledge data base*. An example of a fact in MYCIN is

(Site Culture-1 Blood 1.0)

The expression means that the culture Culture-1 is found in the patient's blood. The factor 1.0 is a certainty factor and stands for definite. We will return to certainty factors later. A fact in MYCIN always contains a triple followed by a certainty factor. The triples consist of a "context-parameter-value".

---

[5] D. A. Waterman, *A Guide to Expert Systems*, Addison-Wesley, Reading, MA, 1986.

A knowledge database not only contains facts within the domain but also heuristic knowledge, rules of thumb, which the expert uses. These rules of thumb may be rules of reasoning and intuitive knowledge that are usually difficult to formalize. Usually, these rules are stored in the form of a number of relatively independent rules, *production rules*. A production rule has one or several *premises* and one or several *actions* that follow if the premises can be proven to be valid. A production rule has the form

IF premise THEN action

An example of a production rule in MYCIN is

IF the gramstain of the organism is gramneg
AND the morphology of the organism is rod
AND the aerobicity of the organism is anaerobic
THEN there is a suggestive evidence (0.6)
that the identity of the organism is bacteroides

The rule is stored in the knowledge database in a more condensed form.

One also has need for *metarules* in an expert system. These rules contain information on the rules in the knowledge database. The metarules describe, for instance, when a specific rule or set of rules should be applied, or, alternatively, when a set of rules should not be applied at all.

An example of a metarule in MYCIN is

IF there are rules relevant to positive cultures
AND there are rules which are relevant to negative cultures
THEN it is definite (1.0) that the former should be done before the latter.

The *inference mechanism* draws conclusions from the knowledge stored in the database. At *forward chaining* known facts are compared to the premises of the rules and the corresponding conclusion may be drawn if all premises are satisfied. At *backward chaining* one tries to verify a hypothesis by finding rules whose conclusion supports the hypothesis. If the facts for evaluating the premises are not known, the system can ask the user.

The MYCIN system makes use of backward chaining. Some systems have both kinds of control strategies; the choice of strategy depends on the task at hand. Backward chaining is often used for diagnoses and interpretation, because it seems that experts often reason in this manner and make a hypothesis quite rapidly and then try to verify it. If it is not possible to verify the hypothesis the expert chooses another, and so on, until he succeeds. Systems for planning and design usually make use of forward chaining. Such tasks often contain many given restrictions and it may be more efficient to see what conclusions can be deduced from these premises.

In many types of everyday reasoning one often uses words like probably, possibly, likely, usually, generally, rarely, in exceptional cases, etc. Such expressions often need to be represented in an expert system. The knowledge does

not have to be absolute in an expert system. The premises for a rule may be established with some certainty while the conclusion will then follow with only a limited degree of certainty.

The *handling of certainty* may be done according to statistical methods by computations of probability as in Baye's theorem. In certain domains not all the necessary data may be present for probability computations to be feasible, or not all conditions for Baye's theorem to hold may be satisfied. Several systems therefore have their own solutions for handling vague information. MYCIN, for instance, uses certainty factors which are a measure of how certain one can be of the conclusion if the premises are satisfied. These factors may assume values between -1, certainly false, to $+1$, certainly true. Both facts and rules may have certainty factors and the certainty factor of the conclusion is a weighting of these. In the examples from MYCIN's knowledge database we saw a fact and a metarule that had the certainty factor $+1$, i.e. definite, and a rule with certainty factor 0.6. The following fact means that Organism-1 is not at all sensitive to penicillin.

(Sensitivity Organism-1 Penicillin -1.0)

Even a comparatively computer-naive user should be able to use an expert system. This requires that the user shall be able to communicate with the system in the terms of the field in question. Making computers understand our language to the full presents great problems. But it is desirable that the communication with the user can take place in *limited natural language*, limited to the terminology of the domain.

For a conclusion from an expert system to be acceptable, the system must be able to explain *how a result was achieved.* The user should be told the facts and rules behind a certain conclusion. At the same time as the user is being asked about something, he should also get to know *why a question was asked*, or, as the case may be, why a question was not asked.

Some examples of questions which may be interesting to ask an expert system are

- how the program reached its conclusion
- how a certain piece of information was used
- what conclusion was drawn about a specific subproblem
- why the program inquires about a certain piece of information
- why a certain piece of information was not used
- why a hypothesis was rejected that the user may have found reasonable

If we return to our MYCIN conversation we may ask the following question, for instance, after having had our diagnosis:

Why did you rule out streptococcus as a possibility for ORGANISM-1?
RULE 33 could have been used to determine that the identity of ORGANISM-1 is streptococcus. However, clause 2 of RULE 33 (the morphology of

the organism is coccus) was already known to be false for ORGANISM-1, so the rule was never tried.

An example of a question about general knowledge in the system is whether a test taken from the blood is a sterile site.

Is blood a sterile source?
Yes. Blood is one of those sites that is normally sterile.

In general, a *knowledge database is builtup incrementally* and the behavior of the expert system is tested as facts and rules are added to the database. Some systems give support during the building of the database, for instance by translating rules from natural language to the syntax the system uses and by trying to control the database in various ways. Moreover, the user should have the opportunity to *modify the knowledge database* in a convenient way.

There are a number of expert systems without domain-specific knowledge, called *expert system shells*. These systems supply the inference mechanism and often also certainty handling, dialog support, explanation facilities, etc. It is important that the system also gives support during the building up of the knowledge database. One advantage of such shells is that it takes much less time to develop an expert system using them, provided that the design of the shell suits the current domain in the first place. There are specially developed expert system shells for nearly all kinds of computers.

An example of an expert system shell is EMYCIN, the part of MYCIN that does not contain the medical knowledge. EMYCIN consists of a comprehensive set of support systems that give versatile support at various stages in the development and maintenance of an expert system. The application area is more or less limited to diagnosis, i.e. the task of diagnosing a state on the basis of available observations and suggesting appropriate measures for treatment. The system is not suitable for design and planning.

Some examples of systems constructed using EMYCIN are

PUFF - diagnoses lung diseases
HEADMED - a psychopharmacological advisor
ONCOCIN - gives advice on lymphatic cancer therapy
SACON - advisor to construction engineers

Knowledge-based systems should not be implemented in traditional programming languages because of the complexity of representing facts and rules generally as well as implementing and warranting the correctness of the inference mechanism. In certain cases it is also not advisable to use existing expert system shells. All shells have built-in restrictions as to what domain may be handled. It is often not possible to modify the type of facts and rules that the system supports or to choose another inference strategy or certainty handling.

If one has a task for which no existing shell is exactly suitable or one wants great freedom of design, there are some general programming languages suitable for developing KBS. Prolog is an example of such a programming language.

An advantage of KBS compared to traditional system development is that the expert himself can build up a knowledge database, either with the help of an expert system shell that supports the construction, or in co-operation with a system expert, a *knowledge engineer*. The knowledge as such may be difficult to reach and to formalize so it is still no easy task. The great advantage, however, is that the database may be tested as new knowledge is added to it. The expert discovers at an early stage whether it must be modified. Furhermore, the expert can follow the construction in a simple way and control it in the desired direction.

The knowledge database consists of *declarative knowledge*, i.e. what holds for the domain. Ordinary systems usually contain *procedural knowledge*, i.e. what is to be done and how. Depending on the domain it may be easier to formalize the knowledge in a descriptive, declarative manner, and not in the form of algorithms. No all information can be expressed in algorithmic form, either. In traditional systems all information must always be categorical and all data must be available. In a KBS there is a possibility of certainty handling and in some cases it is possible to answer a question from the system with "I don't know" and the system is still able to continue its deduction.

Since the knowledge database is separated from the inference mechanism, it is not so complicated a procedure if data needs to be changed. In traditional systems data changes may mean that the programs must be re-programmed in several places depending on where the data is inserted into the programs.

A difficulty which is common to all larger systems is how to make certain that the data in the system is not contradictory, that the system is *consistent*, that all necessary information is included in the system, that it is *complete*, and that the system does not contain superflous information, *redundancy*.

To sum up, the expert system technique has meant a new way of developing systems and one of the intentions of these systems is to carry out the conversation with the user in an "intelligent" way. Another purpose is to spread expert knowledge, not to supplant the expert.

4.3.2 Prolog for Expert Systems

Prolog is a suitable programming language for developing expert systems. One might argue that even an ordinary logic program is a simple knowledge-based system since it contains a knowledge database, the program, and an inference mechanism, the interpreter, which are separated from each other. We will study the connections between the characteristic parts of KBS more closely and how they might be realized in Prolog.

We said above that the *knowledge database* in an expert system consists of domain-specific facts and heuristic knowledge. A *fact* is represented in Prolog as a relation which is always true. If we want to describe the information that the identity of an organism is probably neisseria, we define a Hornclause for the predicate Identity.

```
Identity(organism-2, Neisseria,0.6) ←
```

The first argument of the predicate stands for the organism, the second for the identity, i.e. neisseria, and the last argument is the certainty factor. In MYCIN the word "probably" could correspond to the certainty factor 0.6.

Heuristic knowledge is usually stored as *production rules*. A production rule has the form

```
IF premise THEN action
```

Let us look at one of the rules in MYCIN that tries to determine what kind of bacteria a patient is infected by.

```
IF the stain of the organism is gramneg
      AND the morphology of the organism is coccus
THEN there is strong evidence (0.8)
      that the genus of the organism is Neisseria
```

A Horn clause in a Prolog program has the same form as a production rule with one action. Thus it is simple to express the rule above as a Horn clause.

```
Rule(no, organism, Neisseria,0.8) ←
      Stain(organism,Gramneg),
      Morphology(organism,Coccus)
```

We also need metarules in the knowledge database, rules which can specify what holds for other rules. Metarules can also be represented in Prolog. A rule, i.e. a Horn clause, can be treated as a term in a program which means that we can refer to a rule in the form of a variable.

As we saw above, the knowledge of the rule-based system is represented in a simple way in Prolog. There are always inherent limitations when one wants to represent knowledge in a formal system. However, since Prolog is founded on predicate logic, and since the purpose of predicate logic is to be a tool for representing knowledge, on one hand, and to represent correct conclusions from knowledge, on the other, Prolog is a suitable language for developing expert systems.

In an expert system an *inference mechanism* must be implemented for drawing the conclusions on the basis of the knowledge database and the user's answers. In a Prolog system this mechanism is given since the interpreter is in itself an inference mechanism. The Prolog interpreter is based on backward chaining since the deductions in Prolog are constructed top-down. The search strategy is depth-first. This form of deduction is common in systems suitable for diagnosis and classification. Should one want the deduction carried out in some other way, one may implement an interpreter with the required search strategy. Prolog is a suitable tool for constructing interpreters.

In a knowledge-based system it is interesting to derive alternative solutions. This possibility exists in Prolog since the system backtracks to find alternative evaluations.

In Prolog we can represent a *certainty factor*, connected to a fact or a rule, as an extra argument connected to the predicates. The validity of the rule can then be computed depending on what certainty handling we have chosen, e.g. by computing a certainty factor depending on the certainty factors of the premises and the certainty factor of the entire rule.

Limited natural language may be desirable in the *dialog* with a KBS. Natural language is best described through "context-dependent" grammars, grammars in which the categorizations of certain sentence constructions are dependent on the context in which they appear.

We can easily express "context-free" grammars in Prolog, i.e. grammars which are dependent on their own structure only. The grammar we use in Prolog is called DCG (Definite Clause Grammar). The example in clauses (150), (151) and (152) determines whether a string is a palindrome (a text that reads the same backwards and forwards) or not. A palindrome may consist of an empty string, a single character (arbitrary), or a string of several characters. When the string contains several characters the first and the last have to be the same and the string in between in its turn has to be a palindrome. Instead of the implication arrow used in the Horn clauses we use the arrow "$\longrightarrow$ " which can be read as "consists of". Rule (152) can be read as: a palindrome consists of a character, a palindrome and again the same character as at the beginning. Symbols given within the brackets [and] are terminal symbols, i.e. symbols in the language described by the grammar. The question whether a particular string is a palindrome is formulated by DCG as Palindrome("anna",""). This means that we want to know if the string that is the difference between the first and the second is a palindrome (compare with d-lists). The Palindrome concept in grammar implicitly has two additional arguments not stated in the grammar, but they have to be given when using the grammar.

```
Palindrome ⟶ []                          (150)
Palindrome ⟶ [a]                         (151)
Palindrome ⟶ [a], Palindrome, [a]        (152)
```

A DCG can be complemented by ordinary Prolog clauses. These clauses are marked by the parentheses { and }. The Composite relation has one explicit argument denoting a construction that has been built up during the analysis of the string. In this case the argument denotes the evaluated value of the arithmetic expression represented by the string. For example:

```
← Composite(x,"3+5*5+2","")
x = 24
```

```
Composite(z) ⟶ Expression(x), ['+'], Composite(y), {Value(x+y,z)}
Composite(z) ⟶ Expression(x), ['-'], Composite(y), {Value(x-y,z)}
Composite(z) ⟶ ['('], Composite(z), [')']
Composite(z) ⟶ Expression(z)
Expression(z) ⟶ Number(x), ['*'], Expression(y), {Value(x*y,z)}
```

```
Expression(z) ⟶ Number(x), ['/'], Expression(y), {Value(x/y,z)}
Expression(z) ⟶ ['('], Composite(z), [')']
Expression(z) ⟶ Number(z)
Number(z) ⟶ ['('], Composite(z), [')']
Number(z) ⟶ [z], { Integer(z)}
```

A simple example translating a program clause to a list of words is given below.

```
Fact_interpret(The.x.'of'.y.'is'.z.∅) ⟶
        Predname(x), ['('], Organism(y), [','], Class(z), [')']
Fact_interpret(The.'identity'.'of'.x.'is'.y.z.∅) ⟶
        [Identity,'('], Organism(x), [','], Class(y), [','], Certainty(z), [')']
Predname(x) ⟶ [y], {Make_word(y,x)}
Class(x) ⟶ [y], {Make_word(y,x)}
Certainty('probably') ⟶ [0,'.',6]
Certainty('almost certainly') ⟶ [0,'.',8]
```

To the above grammar we can formulate questions like:

```
← Fact_interpret(textlist,"Stain(organism,Gramneg)","")
textlist = The.'stain'.'of'.'organism'.'is'.'gramneg'.∅


← Fact_interpret(textlist,
                 "Identity(organism,Neisseria,0.6)","")
textlist = The.'identity'.'of'.'organism'.'is'.'probably'.
           'neissaria'.∅
```

Using DCG we can analyze sentences in subsets of natural language. If whole texts in unlimited natural language are to be analyzed, this type of grammar is not sufficient.

The drawing of conclusions in an expert system may be viewed as a derivation. This derivation can be easily stored in the Prolog system. The *explanations* can then be made in the terms of the proof. As answer to the question on how the system arrived at a particular result, one only has to traverse the proof tree upwards, i.e. the derivation backwards.

The domain knowledge is expressed as an axiom set in the form of Horn clauses. From the execution of a question we can construct a proof tree. See, for example, in Sect. 2.1.4 on Proof Trees, the proof tree for the problem of finding a sibling of Magni.

Let us study an AND-tree constructed for the proof of the formula Sibling(Magni,Trud). The root of the proof tree is Sibling(Magni,Trud). When both Parent(Magni,z) and Parent(Trud,z) are deduced we can apply a rule to obtain Sibling(Magni,Trud). In Figure 4.7, the complete proof tree is shown.

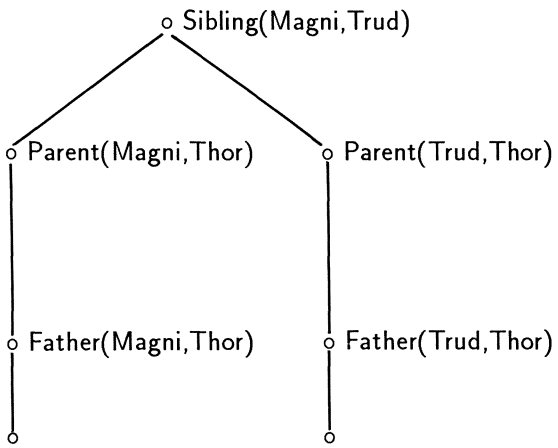

Figure 4.7: A proof tree for the formula Sibling(Magni,Trud)

When we have a non-trivial proof tree it is essential to make the explanation sufficiently abstract. We make use of the tree structure for a stepwise presentation[6] of the proof tree. The first step of the explanation includes the root and its sons. The next step focuses on some of the sons. This also offers the users possibilities of directing the presentation according to their interests.

In order to make the explanation easier to read we present a clause using grammar rules connected to the relations. For example we can give the following grammar for the relations Father, Parent and Sibling.

```
Rel_interpret(x.'and'.y.'are'.'siblings'.∅) ⟶
        [Sibling,'(',x,',',y,')']
Rel_interpret(The.w.'of'.x.'is'.y.∅) ⟶
        Pred(w), {Type(w,'many-to-one')}, ['(',x,',',y,')']
Rel_interpret(An.w.'of'.x.'is'.y.∅) ⟶
        Pred(w), {Type(w,'many-to-many')}, ['(',x,',',y,')']
```

An instance of a rule, i.e. a node with its directly underlying nodes, we express as "A since B" if the rule is A ← B. For example,

$$\text{Parent(Magni,Thor)} \leftarrow \text{Father(Magni,Thor)}$$

we express as "A parent of Magni is Thor since the father of Magni is Thor".

In a proof tree, the arguments of the relations have been instantiated to values that satisfy the formula in the root node. In an explanation we are not interested in all of the values. To present all of them is inconvenient.

We can eliminate "uninteresting" values by reformulating relations as functions and using these as values. For example, a conjunction of two relations

[6] A. Eriksson and A-L. Johansson,"Neat Explanations of Proof Trees", in proceedings from IJCAI 1985, Los Angeles, 1985.

sharing a value can be reformulated as one of the relations taking on a representative function of the other as value.

Parent(Magni,x), Parent(Trud,x)

"A parent of Magni is also a parent of Trud". In this explanation, "a parent of Trud" is a representative function for the parent relation and is substituted for x in the first parent relation.

One approach is to avoid including instantiations that do not appear in the current root. We want to explain the conclusion without including values appearing only in the precondition of a rule.

An important issue for obtaining clarity of explanations is to avoid presenting facts that are already known to the user. General knowledge in the domain of Norse gods could be what names are names of women and what are names of men. If we are going to present the following rule,

Sister(Trud,Magni) ← Father(Trud,Thor), Father(Magni,Thor), Woman(Trud)

we can assume that the fact that Trud is a woman is general knowledge and suppress it. The explanation will be "Trud is a sister of Magni since the father of Trud is the same as the father of Magni".

Specific knowledge possessed by the user would also be taken care of. It is possible that the user is not totally naive; some facts may be well known to her and it would only be negative to repeat them.

From previous chapters we know that we often build up our world description incrementally in a database. This technique is desirable in a KBS also. When constructing the knowledge database it is convenient and maybe necessary to be able to test the behavior of the system as new knowledge is added to it. Prolog supports this feature. For every new Horn clause added to the knowledge database it is possible to test the conclusions of the system and how its behavior is affected.

Another desirable property in a knowledge-based system is that the user should have the possibility of changing the knowledge database. Something in the knowledge database may be wrong — information may be missing or new information may be available. It is also possible for the system to learn by saving information from previous sessions. If the knowledge database is to be modified in the course of a consultation, this is also simple in Prolog. We can update the knowledge database during execution with the built-in predicates Assert and Retract.

MYCIN was originally implemented in Lisp and PROSPECTOR in Interlisp. To test the capacity of Prolog in the knowledge-based systems area, smaller MYCIN and PROSPECTOR systems have been implemented at Imperial College in London.[7] The systems are smaller in that they contain only parts of

[7] P. Hammond, *Logic Programming for Expert Systems*, Imperial College, Technical Report, DOC 82/4, 1982

the knowledge database. The systems were implemented in Micro-Prolog (see Round 8). The experience from these projects was very good. The lesson learned was that the Prolog programming language offered positive support at implementation.

The APES expert system shell is implemented in Micro-Prolog and may be used on many types of personal computers. APES supplies a user interface for developing knowledge-based systems in Prolog. Rules and facts in the knowledge database are expressed as Prolog clauses. APES supports certainty handling in the form of Baye's calculus of probability, certainty factors of MYCIN type or user-defined modules. The system has an explanatory mechanism. Moreover, it is easy to use the underlying Prolog system if the user should want to extend his system with his own Prolog modules.

ESTA is another expert system shell in Prolog. ESTA is implemented in Turbo Prolog (see Round 8).

4.4 An Example

Let us construct a small knowledge-based system for the classification of some felines. We suppose that we get an opportunity to study a feline closer and want to find out what kind of feline it is. In our knowledge database we have information on the felines jaguar, leopard and cheetah. It is easy to extend the knowledge database with information on more felines. Do try this!

We use a built-in construction | in the program. This construction is used to control the backtracking. See Round 6 for a description of the construction.

```
/* The system tries to classify the animal according to the user's description.
      If the description does not fit in with any animal in the knowledge database,
      the user will be informed about this. */
Classification(name,animal)←
      Description_of_Felines(name,animal) |
Classification(name,'is not described in the system') ←

/* Our descriptions of felines contain information on typical length,
      tail length, weight, and the characteristics of the fur. */
Description_of_Felines(name,animal) ←
      Length(name,animal),
      Tail(name,animal),
      Weight(name,animal),
      Fur(name,animal)

/* The length of the animal is checked against the information
      in the knowledge database. */
Length(name,animal) ←
      Answer(name,'length','in centimeters',length),
      Length_check(animal,length)
```

```
/* The tail length we state is compared to the tail length of the animal that was
      instantiated in Length. If it coincides the checking continues, otherwise the
      system backtracks and tries to find an alternative feline. */
Tail(name,animal) ←
      Answer(name,'tail','in centimeters',length),
      Tail_check(animal,length)

Weight(name,animal) ←
      Answer(name,'weight','in kilograms',weight),
      Weight_check(animal,weight)

/* The fur is very characteristic and should be carefully described. */
Fur(name,Jaguar) ←
      Answer(name,'fur color','golden yellow',Yes),
      Answer(name,'fur spots','small black',Yes),
      Answer(name,'fur comments','rosettes of broken rings of black having
      yellow centers and enclosing small black dots',Yes)
Fur(name,Leopard) ←
      Answer(name,'fur color','tawny',Yes),
      Answer(name,'fur spots','small black',Yes),
      Answer(name,'fur comments','spots as rosettes on the back',Yes)
Fur(name,Cheetah) ←
      Answer(name,'fur color','tawny',Yes),
      Answer(name,'fur spots','small, black',Yes),
      Answer(name,'fur comments',
      'black stripes as tears from the eyes to the nose',Yes)

/* The user gets a chance to answer the system's questions
      about the look of the animal. The answers are saved
      in the database to avoid posing the same question all over again. */
Answer(name,t1,t2,answer) ←
      Answered(name,t1,t,a) | ,t2 = t, answer = a
Answer(name,text1,text2,answer) ←
      Out_term(name), Tab(5),
      Out_term(text1), Tab(2), Out_term(text2), Line_shift,
      In_term(answer), Assert(Answered(name,text1,text2,answer))

/* The check of the animal's length is within the interval limits.*/
Length_check(Jaguar,length) ← 110 ≤ length, length ≤ 180
Length_check(Leopard,length) ← 112 ≤ length, length ≤ 135
Length_check(Cheetah,length) ← 120 ≤ length, length ≤ 190

/* The check of the length of the tail */
Tail_check(Jaguar,length) ← 60 ≤ length, length ≤ 90
Tail_check(Leopard,length) ← 90 ≤ length, length ≤ 110
Tail_check(Cheetah,length) ← 70 ≤ length, length ≤ 95
```

```
/* The check of the weight of the animal */
Weight_check(Jaguar,weight) ← 57 ≤ weight, weight ≤ 113
Weight_check(Leopard,weight) ← 30 ≤ weight, weight ≤ 60
Weight_check(Cheetah,weight) ← 39 ≤ weight, weight ≤ 65
```

Let us test the program with a description of Onca.

```
← Classification(Onca,animal)
length in centimeters
*>> 134.
tail in centimeters
*>> 78.
weight in kilograms
*>> 70.
fur color golden yellow
*>> Yes.
animal = Jaguar ;              is there an alternative?
no
```

Onca is a jaguar. Now we describe an animal named Tom.

```
← Classification(Tom,animal)
length in centimeters
*>> 40.
animal = is not described in the system
yes
```

Tom was too short to be one of our animals in the system. Let us ask about Charlie instead.

```
← Classification(Charlie,animal)
length in centimeters
*>> 130.
tail in centimeters
*>> 90.
weight in kilograms
*>> 50.
fur color tawny
*>> Yes.
fur spots small, black
*>> Yes.
fur comments spots as rosettes on the back
*>> No.
fur comments black stripes as tears from the eyes to the nose
*>> Yes.
animal = Cheetah
```

```
yes
```

Charlie is a cheetah.

4.5 Exercises

Exercise 1:

Express a relation between a list of numbers and a list in which all odd numbers in the list have been assembled. Note that there will only be a list of odd numbers; no new list shall be generated on backtracking.
Hint: the built-in construction x Mod y will give the remainder in an integer division of x by y.

Exercise 2:

Let us extend our database on felines.

```
Big_cat(x) ← Leopard(x)
Colour(Jarvis,Tawny) ←
Colour(Leo,Tawny) ←
Colour(Louis,Tawny) ←
Colour(Blackie,Black) ←
Spotted(Jarvis) ←
Spotted(Leo) ←
```

Define in the database what holds for a leopard:

(i) An animal is a leopard if its color is tawny and the animal is spotted.

(ii) An animal is a leopard if the color of the animal is black and the animal is not spotted.

Ask the following questions from the system with the help of All_answers or Sorted_answers.

a) What are the names of the leopards?

b) Assemble the name and color of each leopard in a list.

c) Which of the felines are not spotted?

d) What different zoological gardens are represented in our database?

e) In what zoological gardens are the big cats? What will the Prolog system reply to the following questions?

f) ← Not Leopard(Louis)

g) ← Not Leopard(Blackie)

h) ← Not Spotted(x)

Exercise 3:

A relation for distances between various cities is given in this way:

Distance(Washington,Baltimore,37) ←
Distance(Washington,Cleveland,346) ←
Distance(Washington,Philadelphia,133) ←
Distance(Washington,New-York,233) ←
Distance(Washington,Pittsburgh,221) ←
Distance(Cleveland,Baltimore,497) ←
Distance(Cleveland,New-York,647) ←
Distance(Cleveland,Pittsburgh,287) ←
Distance(Philadelphia,Baltimore,96) ←
Distance(Philadelphia,New-York,100) ←
Distance(New-York,Baltimore,196) ←
Distance(New-York,Pittsburgh,368) ←

Define a relation Shortest_route between two cities and an itinerary. The itinerary is a list of the cities passed on the shortest route between the two cities.

Round 5. Program Methodology

5.1 Derivation of Programs

5.1.1 Specifications - Programs

We have seen that programs in Horn form can be written in a simple and declarative way. Prolog programs without extralogical constructions have a direct connection to formulas in first-order predicate logic.

In our programs we are restricted to a given form, i.e. the Horn form with only one atomic formula as consequence. We use the connectives $\wedge$ and $\leftarrow$ and the universal quantifier $\forall$. Variables are universally quantified over the entire clause. The limitation set on the mode of expression is explained by our wish to be able to construct derivations mechanically from the program. There are several computer-based Prolog systems that use Horn form and resolution[1].

The standard form of first-order predicate logic provides richer means of expression than the Horn form. We can express the clauses using the quantifiers $\forall$, $\exists$ and the logical connectives $\wedge$, $\vee$, $\rightarrow$, $\leftarrow$, $\leftrightarrow$, and $\neg$. We have thus more linguistic instruments at our disposal for describing a relation. The advantage of defining a relation in standard form is that we use a language that is close to natural language. The disadvantage is that the specification is not always *computationally useful*, i.e. it cannot be used for mechanically generating all instances of the relation we want to compute. There are not yet enough possibilities for executing derivations from premises in standard form. However, a specification in standard form, even if not computationally useful, will help us clarify the concepts to be computed. If we can relate the standard form specification to the program in Horn form in such a way that we can be sure that the program corresponds to the specification, we have the benefit of both the naturalness of standard form and the mechanized derivations possible in Horn form. The standard form definition can thus be said to constitute the specification of a program. The program replaces the specification when instances of the relation are computed.

---

[1] See Round 8.

When we deduce a clause C from premises $P_1, \ldots, P_n$, we say that C follows from $P_1, \ldots, P_n$, which is written as

$$P_1, \ldots, P_n \vdash C$$

We know, generally, that if we can derive a program P_{prog} from the specification for the program P_{spec} it holds that

$$P_{spec} \vdash P_{prog}$$

All conclusions C which we can deduce from the program can also be deduced from the specification.

$$\text{If } P_{prog} \vdash C \text{ then } P_{spec} \vdash C$$

This follows from the transitivity of $\vdash$. Since $P_{spec} \vdash P_{prog}$ and $P_{prog} \vdash C$ then $P_{spec} \vdash C$. The program is thus *correct* as regards the specification. Another question is thus whether or not the program yields all conclusions which are valid according to the specification, i.e. whether the program is complete.

$$\text{If } P_{spec} \vdash C \text{ then } P_{prog} \vdash C$$

If a part of the program clauses which are valid consequences of the specification is missing in the program, we have a program that is correct but not complete. Completeness will be treated in more detail further on in this chapter.

5.1.2 Natural Deduction

Specification and program are expressed in the same logical formalism, the only difference being that the linguistic means in the program are limited. We can relate the program to the specification by logical derivations. The premises of the derivation, i.e. the specification, are given in first-order predicate logic, which means that we cannot use resolution for constructing derivations without transforming the formulas[2]. We use a proof system whose inference rules support *hypothetical reasoning*. In hypothetical reasoning we make a *hypothesis*, an assumption, from which we try to deduce a conclusion. If the hypothesis is valid, the conclusion will be valid, too. This fits in well with the conditional clauses in the Horn form. The consequence in a Horn clause is valid *if* the conditions are satisfied. The conclusion is deduced by reasoning from the premises; the proof procedure is direct.

---

[2] We can apply resolution if we transform the clauses into clausal form. In Round 2 we presented the resolution rule applied to Horn clauses. The resolution rule is also applicable to clauses. For Horn clauses the resolution rule is complete. In the case of clauses we have to add one more rule to reach completeness. A problem with resolution and program derivation is that we do not know the appearance of the conclusion for the proof by contradiction.

A *proof system* is determined by the permitted language, a set of inference rules, and *axioms*. Axioms are the clauses that are considered valid in the system. The system we will use is called *natural deduction*. The derivations are natural in the sense that they imitate human reasoning. The language is first-order predicate logic in standard form. The system contains no axioms.

We describe inference rules in the system by means of a figure. C is the conclusion we are permitted to draw from the premises $P_1,\ldots, P_n$.

$$\frac{P_1 \quad \ldots \quad P_n}{C}$$

A natural deduction system contains inference rules for introduction and elimination of the logical symbols $\forall$, $\exists$, $\wedge$, $\vee$, $\rightarrow$, $\leftarrow$, $\leftrightarrow$, and $\neg$. We name the rules by writing the logical symbol followed by either I for introduction or E for elimination. See, for example rule, $\wedge$I , "and-introduction", which denotes that if we have premises A and B we are permitted to draw the conclusion $A \wedge B$.

$\wedge$I)

$$\frac{A \quad B}{A \wedge B}$$

An expression within parentheses signifies a hypothesis. Examples are the rules for introduction of $\rightarrow$. If we can deduce B given the hypothesis A, we are permitted to draw the conclusion $A \rightarrow B$, or equivalently, $B \leftarrow A$.

$\rightarrow$I)

$$\frac{\begin{array}{c}(A)\\ B\end{array}}{A \rightarrow B} \qquad \frac{\begin{array}{c}(A)\\ B\end{array}}{B \leftarrow A}$$

The notation A_t^x denotes the formula where all free occurrences of x are substituted for t, that is, the formula we will get by applying the substitution $\{\langle x, t\rangle\}$ on formula A. Rule $\forall$E means that we can generate instances from a universally quantified formula by eliminating the quantifier and replacing occurrences of x by a term t.

$\forall$E)

$$\frac{\forall x A}{A_t^x}$$

We present here a summary of the rules in a natural deduction system:

$\wedge$I)

$$\frac{A \quad B}{A \wedge B}$$

$\wedge$E)

$$\frac{A \wedge B}{A} \qquad \frac{A \wedge B}{B}$$

$\vee$I)

$$\frac{A}{A \vee B} \qquad \frac{B}{A \vee B}$$

$\vee$E)

$$\frac{A \vee B \quad \begin{matrix}(A)\\ C\end{matrix} \quad \begin{matrix}(B)\\ C\end{matrix}}{C}$$

$\rightarrow$I)

$$\frac{\begin{matrix}(A)\\ B\end{matrix}}{A \rightarrow B} \qquad \frac{\begin{matrix}(A)\\ B\end{matrix}}{B \leftarrow A}$$

$\rightarrow$E)

$$\frac{A \quad A \rightarrow B}{B} \qquad \frac{A \quad B \leftarrow A}{B}$$

$\leftrightarrow$I)

$$\frac{A \rightarrow B \quad A \leftarrow B}{A \leftrightarrow B}$$

$\leftrightarrow$E)

$$\frac{A \leftrightarrow B}{A \rightarrow B} \qquad \frac{A \leftrightarrow B}{A \leftarrow B}$$

$\forall$I)

$$\frac{A}{\forall x A^{\,a}_{\,x}}$$

a must not be included in any assumption on which A is dependent.

$\forall$E)

$$\frac{\forall x A}{A^{\,x}_{\,t}}$$

$\exists$I)

$$\frac{A^{\,x}_{\,t}}{\exists x A}$$

$\exists$E)

$$\frac{\exists x A \quad \begin{matrix}(A^{\,x}_{\,a})\\ B\end{matrix}}{B}$$

a must not be included in $\exists x A$, in B or in any assumption which the upper B is dependent on except $A^{\,x}_{\,a}$.

$\perp_{int}$)

$$\frac{\perp}{A}$$

$\neg$E)

$$\frac{A \quad \neg A}{\perp}$$

A derivation in natural deduction is performed by breaking down the premises into parts using elimination rules and constructing the theorem from these parts using the introduction rules.

Let us deduce a simple program to demonstrate the inference rules and program derivation. Our starting point is a definition of the concept paternal grandparent in standard form and our aim is to deduce a program in Horn form. The relation *Paternal_grandparent* between x and y is valid if, and only if, there is a z such that z is father of x and y is father or mother of z.

$$\forall x \forall y \, (Paternal\_grandparent(x, y) \leftrightarrow \exists z(Father(x, z) \wedge (Father(z, y) \vee Mother(z, y)))) \qquad (s1)$$

We saw in Sect. 1.2.1 on Horn form that a program clause has this general appearance:

$$\forall x_1 \ldots \forall x_n (A \leftarrow B_1 \wedge \cdots \wedge B_j)$$

One or several clauses with the same predicate in the consequence constitute the program for the relation.

The relation we are interested in is *Paternal_grandparent*. We want to deduce clauses with this appearance:

$$\forall x \forall y \ldots (Paternal\_grandparent(x, y) \leftarrow \cdots)$$

We know what form the clause shall have but the formulas in the conditional part are unknown. Therefore, we cannot make any hypotheses yet. Let us reason from the conclusion and its form. We can start with the *main logical symbol* in the logical expression, i.e. the logical symbol that determines the form of the clause. The program clause we wish to deduce is a universally quantified formula. The main logical symbol is thus $\forall$. We can study the rules for natural deduction and see what rule will result in a universally quantified formula. On condition that we have the formula A, the rule for $\forall$I yields a $\forall x A\,{}^{a}_{x}$ where a must not be included in any assumption on which A depends.

$\forall$I)

$$\frac{A}{\forall x A\,{}^{a}_{x}}$$

We can deduce $\forall x \forall y \, (Paternal\_grandparent(x, y) \leftarrow \cdots)$ by applying $\forall$I to the expression $\forall y \, (Paternal\_grandparent(a, y) \leftarrow \cdots)$.

Every step in a derivation constructed according to the natural deduction rules is either

i) a hypothesis

ii) a consequence of preceding steps or

iii) a consequence of some of the premises of the derivation.

Let us present the derivation steps in sequence. Every step is assigned a number and a commentary on which rule has been applied to deduce the step. The steps are usually numbered consecutively from top to bottom, but when we do not know what number the step will be assigned in such an order, we will use a letter instead. We do not know what number the conclusion will eventually get in the end; we have given it the number (n). The assumption for the inference step must have a lower number, so we will give it the number (n-1). If we apply the ∀I - rule again we get an expression whose form is a conditional clause.

(n-2)	$Paternal\_grandparent(a, b) \leftarrow \cdots$	
(n-1)	$\forall y\,(Paternal\_grandparent(a, y) \leftarrow \cdots)$	∀I (n-2)
(n)	$\forall x \forall y\,(Paternal\_grandparent(x, y) \leftarrow \cdots)$	∀I (n-1)

The main symbol in the clause is ←. To deduce a conditional clause we can apply the rule ←I .

←I)

$$\frac{\begin{matrix}(A)\\ B\end{matrix}}{B \leftarrow A}$$

We shall thus make a hypothesis and then deduce *Paternal_grandparent*.

$$\frac{\begin{matrix}(\cdots)\\ Paternal\_grandparent(a, b)\end{matrix}}{Paternal\_grandparent(a, b) \leftarrow \cdots}$$

We mark in the layout that derivation steps are dependent on a hypothesis by enclosing those steps in a bracket. The bracket begins at the hypothesis and ends when we have deduced a clause no longer dependent on the hypothesis. The step (n-2) in the lay-out below is no longer dependent on the hypothesis (k).

(k)	$\cdots$	hyp
	$\cdots$	
(n-3)	$Paternal\_grandparent(a, b)$	
(n-2)	$Paternal\_grandparent(a, b) \leftarrow \cdots$	←I (k)(n-3)

We cannot continue the decomposition of the conclusion because we have reduced an atomic formula in step (n-3). We still do not know what hypotheses we shall have to introduce. Let us reason from the premises for a while. We want to break down the premises into subformulas that we can tie together with the wanted consequence in step (n-3). Our only premise is a specification of *Paternal_grandparent*. The specification is a universally quantified formula. Let us select once more a rule from the main logical symbol in the formula. We decompose the specification using the rules for ∀E and ↔E. In the reasoning from the conclusion we have used the internal variables or parameters a and b. Let us therefore instantiate the specification with the same parameters.

(1) $Paternal\_grandparent(a,b) \leftrightarrow$ $\exists z\,(Father(a,z) \wedge (Father(z,b) \vee Mother(z,b)))$	$\forall$E $(s1)$
(2) $Paternal\_grandparent(a,b) \leftarrow$ $\exists z\,(Father(a,z) \wedge (Father(z,b) \vee Mother(z,b)))$	$\leftrightarrow$E (1)

Step (2) gives us a hint as to how step (n-3) shall be deduced. Step (2) is one of the premises for applying $\leftarrow$E and step (n-3) is the conclusion. Step (2) corresponds to the expression $B \leftarrow A$ and step (n-3) corresponds to B. The formula $\exists z\,(Father(a,z) \wedge (Father(z,b) \vee Mother(z,b))$ thus corresponds to A. If we can deduce this expression, we will obtain step (n-3) by applying rule $\leftarrow$E. We now have a new, wanted consequence that is non-atomic. The main logical symbol in the formula is $\exists$. An existentially quantified formula $\exists xA$ is deduced from formula A. We introduce one more parameter c.

(1) $Paternal\_grandparent(a,b) \leftrightarrow$ $\exists z\,(Father(a,z) \wedge (Father(z,b) \vee Mother(z,b)))$	$\forall$E $(s1)$
(2) $Paternal\_grandparent(a,b) \leftarrow$ $\exists z\,(Father(a,z) \wedge (Father(z,b) \vee Mother(z,b)))$	$\leftrightarrow$E (1)
(k) $\cdots$ $\cdots$	hyp
(n-5) $Father(a,c) \wedge (Father(c,b) \vee Mother(c,b))$	
(n-4) $\exists z\,(Father(a,z) \wedge (Father(z,b) \vee Mother(z,b)))$	$\exists$I (n-5)
(n-3) $Paternal\_grandparent(a,b)$	$\rightarrow$E (2)(n-4)
(n-2) $Paternal\_grandparent(a,b) \leftarrow \cdots$	$\leftarrow$I (k)(n-3)
(n-1) $\forall y\,(Paternal\_grandparent(a,y) \leftarrow \cdots)$	$\forall$I (n-2)
(n) $\forall x \forall y\,(Paternal\_grandparent(x,y) \leftarrow \cdots)$	$\forall$I (n-1)

Step (n-5) is a conjunction. The assumptions for deducing a conjunction are the parts of the conjunction, in this case $Father(a,c)$ and $Father(c,b) \vee Mother(c,b)$. We cannot deduce $Father(a,c)$ and $Father(c,b) \vee Mother(c,b)$ from the premises. We cannot proceed any further in the derivation without making a hypothesis. Let us first assume the hypothesis $Father(a,c)$. This is a prerequisite for deducing step (n-5). The other assumption is in disjunctive form. To deduce a disjunction it is enough to have any of the parts of the disjunction as assumption. To deduce the disjunction $Father(c,b) \vee Mother(c,b)$, we can assume $Father(c,b)$ or $Mother(c,b)$. We decide to introduce the hypothesis $Father(c,b)$. We can deduce the wanted formula from the two hypotheses by applying $\vee$I and $\wedge$I .

(3) $Father(a,c)$	hyp
(4) $Father(c,b)$	hyp
(5) $Father(c,b) \vee Mother(c,b)$	$\vee$I (4)
(6) $Father(a,c) \wedge (Father(c,b) \vee Mother(c,b))$	$\wedge$I (3)(5)

(1)	$Paternal\_grandparent(a,b) \leftrightarrow$ $\exists z\,(Father(a,z) \wedge (Father(z,b) \vee Mother(z,b)))$	$\forall$E $(s1)$
(2)	$Paternal\_grandparent(a,b) \leftarrow$ $\exists z\,(Father(a,z) \wedge (Father(z,b) \vee Mother(z,b)))$	$\leftrightarrow$E (1)
(3)	$Father(a,c)$	hyp
(4)	$Father(c,b)$	hyp
(5)	$Father(c,b) \vee Mother(c,b)$	$\vee$I (4)
(6)	$Father(a,c) \wedge (Father(c,b) \vee Mother(c,b))$	$\wedge$I (3)(5)
(7)	$\exists z\,(Father(a,z) \wedge (Father(z,b) \vee Mother(z,b)))$	$\exists$I (6)
(8)	$Paternal\_grandparent(a,b)$	$\leftarrow$E (2)(7)
(9)	$Paternal\_grandparent(a,b) \leftarrow$ $Father(a,c) \wedge Father(c,b)$	$\leftarrow$I (3)(4)(8)
(10)	$\forall y\,(Paternal\_grandparent(a,y) \leftarrow$ $Father(a,c) \wedge Father(c,y))$	$\forall$I (9)
(11)	$\forall x \forall y\,(Paternal\_grandparent(x,y) \leftarrow$ $Father(x,c) \wedge Father(c,y))$	$\forall$I (10)
(12)	$\forall z \forall y \forall x\,(Paternal\_grandparent(x,y) \leftarrow$ $Father(x,z) \wedge Father(z,y))$	$\forall$I (11)

Figure 5.1: Derivation of a clause for the relation paternal grandparent.

The step (6) corresponds to step (n-5) so let us assemble the different parts to make up a whole derivation (see Figure 5.1).

We can deduce one more program clause from the specification. To deduce the formula disjunction $Father(c,b) \vee Mother(c,b)$, we assumed $Father(c,b)$. The alternative, to assume $Mother(c,b)$, will yield the remaining program clause. We have now arrived at a program with the following appearance in Horn form:

```
Paternal_grandparent(x,y) ← Father(x,z), Father(z,y)      (153)
Paternal_grandparent(x,y) ← Father(x,z), Mother(z,y)      (154)
```

We have performed the derivation by reasoning alternatively from the conclusion and from the assumptions. We have introduced missing assumptions as hypotheses. These hypotheses form the conditional part of the Horn clause in the deduced program.

5.1.3 A Programming Calculus

A relation can be expressed at a higher level of abstraction in a specification than in a program. We can, for instance, express a relation in a specification without explicitly stating the representation of structures or how they should be treated.

Let us define the relation *Intersection* between three sets x, y, and z. The

relation is valid if, and only if, for every element holds that it belongs to z if, and only if, it belongs to both x and y. For the time being we leave the question of what structures we use to represent the sets and how membership shall be demonstrated.

$$\forall x \forall y \forall z \, (Intersection(x, y, z) \leftrightarrow \forall v \, (Member(v, z) \leftrightarrow Member(v, x) \wedge Member(v, y))) \quad (s2)$$

In a similar way, let us define a relation *Subset* between two sets w and z. The set w is a subset of set z if, and only if, all elements belonging to w also belong to z.

$$\forall w \forall z \, (Subset(w, z) \leftrightarrow \forall v \, (Member(v, w) \rightarrow Member(v, z))) \quad (s3)$$

In a definition meant for execution, it is necessary to know what the structure looks like and the relation between the parts of the structure. We need definitions of data structures that occur frequently in programs to be able to derive programs from specifications.

A proof system that contains definitions of data structures as axioms and in which we are primarily interested in derivations concerning programming can be called a *programming calculus*[3]. The language is first-order predicate logic with identity and the the inference rules are those of natural deduction.

Definitions of identity are fundamental in the system. We know that all objects are identical with themselves. We also know that if a relation holds for an object x, and x is identical with an object y, the relation holds for the object y. This is a statement which we want to hold independently of what formula we are discussing: it shall hold for all formulas. We formulate an *axiom schema.* In an axiom schema formulas are variables. By instantiating the axiom schema, we can get axioms that hold for a particular formula. We formulate the axiom schema for the predicate P.

$$\forall x \; x = x \quad (d1)$$

$$\forall x \forall y \, (P(x) \wedge x = y \rightarrow P(y)) \quad (d2)$$

We can instantiate in the schema and let $P(x)$ correspond to the formula $Q(x, z) \vee R(x)$.

$$\forall x \forall y \, ((Q(x, z) \vee R(x)) \wedge x = y \rightarrow (Q(y, z) \vee R(y)))$$

We also use transitivity for identity and relations like $<$.

$$\forall x \forall y \forall z \, (x = y \wedge y = z \rightarrow x = z) \quad (d3)$$

$$\forall x \forall y \forall z \, (x < y \wedge y < z \rightarrow x < z) \quad (d4)$$

[3] See Å. Hansson and S-Å. Tärnlund, *A Natural Programming Calculus*, Proc. IJCAI-6, Tokyo, 1979

In this section we will concentrate on the data structures simple lists and binary trees. Programs for these structures and definitions have been treated in Round 3. Apart from data structures, the most fundamental of the relations concerning these data structures will be defined. It is fundamental to determine whether two structures are identical or not, and to decide whether an element is a member of the structure or not.

The constant $\emptyset$ denotes an empty list. A term composed of the constructor ".", of elements, and the constant $\emptyset$, is called a list. The number of elements in the list may be arbitrary. We define the concept recursively.

$$\forall w\,(List(w) \leftrightarrow w = \emptyset \vee \exists x \exists y\,(w = x.y \wedge Element(x) \wedge List(y))) \qquad (d5)$$

Elements may be arbitrary terms.

$$\forall x\; Element(x)$$

The definition of list makes use of identity. We require the possibility to decide whether lists are identical or not. An empty list is never identical with a constructed list. Two terms composed by the constructor "." are identical if the two parts are identical. For a list w, an element u is a member of the list if, and only if, w has elements, i.e. if the list is constructed. The element u has to be identical with the first element in the list w, or be a member of the rest of the list.

$$\forall x \forall y \neg\, \emptyset = x.y \qquad (d6)$$

$$\forall x \forall y \forall v \forall z\,(x.y = v.z \leftrightarrow x = v \wedge y = z) \qquad (d7)$$

$$\forall w \forall u\,(List(w) \rightarrow (Member(u,w) \leftrightarrow \exists x \exists y\,(w = x.y \wedge (u = x \vee Member(u,y))))) \qquad (d8)$$

From the definition of membership for lists we can deduce two lemmas. A *lemma* is a conclusion from axioms or definitions and is used to facilitate the derivations. It is a reformulation of axioms or definitions and therefore does not introduce any new concepts as the definition does. The lemmas we can deduce from the definition of list are simpler to use in the derivations than the definition itself. The first lemma tells us that no elements belong to an empty list. The second lemma tells us that an element u belongs to a constructed list $x.y$ if, and only if, u is either identical with the first element x or belongs to the list y.

$$\forall u \neg\, Member(u, \emptyset) \qquad (d9)$$

$$\forall x \forall y \forall u\,(List(x.y) \rightarrow (Member(u, x.y) \leftrightarrow u = x \vee Member(u,y))) \qquad (d10)$$

Suppose that we want to prove that a statement holds for all lists. Properties of lists are proved in a way analogous to our formulation of definitions over the

data structure list. Recursive definitions have at least one base case and at least one recursive case. If we want to prove that all lists satisfy a particular property P, we first prove that the property holds for the empty list $\emptyset$, i.e. the base case.

$$P(\emptyset)$$

The next step is to demonstrate that if the property holds for an arbitrary list y, it also holds for a list increased by one more element x. Thus we prove, for all x and y, that $P(x.y)$ holds on condition that $P(y)$ holds, i.e. the recursive case.

$$\forall x \forall y \, (P(y) \rightarrow P(x.y))$$

If we can prove both these formulas we can draw the conclusion that the property holds for all lists. We have formulated a template or *schema* for performing proofs for lists. A proof carried out according to this template is called *proof by induction* and the schema is called *induction schema.* The induction schema for lists looks like this:

$$P(\emptyset) \wedge \forall x \forall y \, (List(y) \wedge P(y) \rightarrow P(x.y)) \rightarrow \forall w \, (List(w) \rightarrow P(w))$$

Let us now advance to the characterization of binary trees. A variable w represents a binary tree if, and only if, w is either an empty tree $\emptyset$, or a construction with the constructor T and there exist x, y, and z, such that y is an element, x and z are binary trees, and w is equal to $T(x, y, z)$.

$$\begin{aligned} &\forall w \, (Binary\_tree(w) \leftrightarrow \\ &\quad w = \emptyset \vee \exists x \exists y \exists z \, (w = T(x, y, z) \\ &\qquad\qquad \wedge Binary\_tree(x) \wedge Element(y) \\ &\qquad\qquad \wedge Binary\_tree(z))) \end{aligned} \tag{$d11$}$$

As in the case of lists, an empty tree differs from a constructed tree. Furthermore, two terms composed by the constructor T are identical if the constituent parts are identical.

$$\forall x \forall y \forall z \neg \emptyset = T(x, y, z) \tag{$d12$}$$

$$\begin{aligned} &\forall x \forall y \forall z \forall v \forall u \forall w \, (T(x, y, z) = T(v, u, w) \leftrightarrow \\ &\qquad x = v \wedge y = u \wedge z = w) \end{aligned} \tag{$d13$}$$

Let us define the fundamental relations for binary trees. An element u belongs to a binary tree w if, and only if, w is a construction $T(x, y, z)$ and u is either identical to y or u belongs to any of the subtrees x and z. In a manner analogous to that of list membership, two lemmas for tree membership can be deduced from the definition. No element belongs to the empty tree. If we have a binary tree $T(x, y, z)$, u belongs to the tree if, and only if, u is either equal

to y or u belongs to any of the subtrees x or z.

$$\forall w \forall u\,(Binary\_tree(w) \rightarrow \qquad (d14)$$
$$(Member(u,w) \leftrightarrow$$
$$\exists x \exists y \exists z\,(w = T(x,y,z) \wedge (u = y \vee Member(u,x) \vee Member(u,z)))))$$
$$\forall u \neg Member(u,\emptyset) \qquad (d15)$$
$$\forall x \forall y \forall z \forall u\,(Binary\_tree(T(x,y,z)) \rightarrow \qquad (d16)$$
$$(Member(u,T(x,y,z)) \leftrightarrow u = y \vee Member(u,x) \vee Member(u,z)))$$

The definitions for trees are given for the empty tree and then a constructed tree from two trees x and z together with an additional element y, using the constructor T, i.e. $T(x,y,z)$. The induction schema for binary trees looks like this:

$$P(\emptyset)$$
$$\wedge \forall x \forall y \forall z\,(Binary\_tree(x) \wedge Binary\_tree(z) \wedge P(x) \wedge P(z) \rightarrow P(T(x,y,z)))$$
$$\rightarrow \forall w\,(Binary\_tree(w) \rightarrow P(w))$$

Let us now look at a derivation of a program for the relation *Subset*. Derivations of programs in the calculus can make use of, apart from the program specification, the definitions of the calculus. We deduce the program from the specification and definitions for data structures.

$$\forall w \forall z\,(Subset(w,z) \leftrightarrow \forall v\,(Member(v,w) \rightarrow Member(v,z))) \qquad (s3)$$

We choose to represent the sets w and z by lists. A list can be either empty or constructed. We can look at the various combinations of arguments that may present themselves.

Argument 1	**Argument 2**
empty list	empty list
empty list	constructed list
constructed list	empty list
constructed list	constructed list

When the first list is empty, we can deduce a clause using the fact that no element belongs to an empty list. When the first list is constructed and the second list is empty, the subset relation is not valid. The formula in the definiens of the definition becomes false, since the antecedent is true and the consequence false. The remaining case is when both lists are constructed.

The relation shall hold for all lists in the first argument under certain conditions. We can define the program recursively over the first argument. We should thus get at least two Horn clauses. The schematic appearance of the program is

$$\forall z\,(Subset(\emptyset,z) \leftarrow \cdots)$$
$$\forall u \forall z \forall x \forall y\,(Subset(x.y,u.z) \leftarrow Subset(y,u.z) \wedge \cdots)$$

(1)	$Subset(\emptyset, list) \leftarrow \forall v\,(Member(v, \emptyset) \rightarrow Member(v, list))$	$\forall$E ,$\leftrightarrow$E $(s3)$
(2)	$Member(elem, \emptyset)$	hyp
(3)	$\neg\, Member(elem, \emptyset)$	$\forall$E $(d9)$
(4)	$\bot$	$\neg$E (2)(3)
(5)	$Member(elem, list)$	$\bot_{int}$ (4)
(6)	$Member(elem, \emptyset) \rightarrow Member(elem, list)$	$\rightarrow$I (2)(5)
(7)	$\forall v\,(Member(v, \emptyset) \rightarrow Member(v, list))$	$\forall$I (6)
(8)	$Subset(\emptyset, list)$	$\leftarrow$E (1)(7)
(9)	$\forall z\; Subset(\emptyset, z)$	$\forall$I (8)

Figure 5.2: Derivation of the base case in a program for the relation subset.

We will start with the base case. We reason in the same way as in the derivation of the program for the *Paternal_grandparent* relation, i.e. from the program form. We continue our reasoning from given premises by instantiating $\emptyset$ in the specification and then eliminating the $\leftrightarrow$. We need to deduce a universally quantified implication to get step (n-1) from step (1) through $\leftarrow$E . We assume the antecedent of the implication as hypothesis, and then apply the rule for $\rightarrow$I, when we have derived the consequence of the implication.

(1)	$Subset(\emptyset, list) \leftarrow \forall v\,(Member(v, \emptyset) \rightarrow Member(v, list))$	$\forall$E ,$\leftrightarrow$E $(s3)$
(2)	$Member(elem, \emptyset)$	hyp
	...	
(n-4)	$Member(elem, list)$	
(n-3)	$Member(elem, \emptyset) \rightarrow Member(elem, list)$	$\rightarrow$I (2)(n-4)
(n-2)	$\forall v\,(Member(v, \emptyset) \rightarrow Member(v, list))$	$\forall$I (n-3)
(n-1)	$Subset(\emptyset, list)$	$\leftarrow$E (1)(n-2)
(n)	$\forall z\; Subset(\emptyset, z)$	$\forall$I (n-1)

From the definition $(d9)$ we know that no elements are members of the empty list. We eliminate the universal quantifier and receive a clause that is an immediate contradiction of the hypothesis in step (2). By applying the rules $\neg$E and $\bot_{int}$ we can get the desired clause.

(3)	$\neg\, Member(elem, \emptyset)$	$\forall$E $(d9)$
(4)	$\bot$	$\neg$E (2)(3)
(5)	$Member(elem, list)$	$\bot_{int}$ (4)

The entire derivation of the base case is shown in Figure 5.2. The result $\forall z\; Subset(\emptyset, z)$ is a Horn clause without a conditional part, since we introduced only temporary hypotheses during the derivation.

Let us turn to the recursive case. We instantiate in the specification of *Subset* and we put down as hypothesis conditions already known.

(1) $Subset(e1.list1, e.list) \leftarrow$ $\forall$E ,$\leftrightarrow$E $(s3)$
$\quad \forall v\,(Member(v, e1.list1) \rightarrow Member(v, e.list))$
(2) $Subset(list1, e.list)$ hyp
$\quad \ldots$
(n-2) $Subset(e1.list1, e.list)$
(n-1) $Subset(e1.list1, e.list) \leftarrow Subset(list1, e.list) \wedge \cdots$ $\leftarrow$I (2)(n-2)
(n) $\forall u \forall z \forall x \forall y\,(Subset(x.y, u.z) \leftarrow Subset(y, u.z) \wedge \cdots)$ $\forall$I (n-1)

To get the consequence $Subset(e1.list1, e.list)$ we need to deduce the antecedent of the formula in step (1). We have the hypothesis $Subset(list1, e.list)$; let us instantiate the specification of *Subset* with the same parameters $list1$ and $e.list$. From step (5) we shall try to deduce step (n-3). Reasoning from the top level yields

(3) $Subset(list1, e.list) \rightarrow$ $\forall$E $(s3)$
$\quad \forall v\,(Member(v, list1) \rightarrow Member(v, e.list))$
(4) $\forall v\,(Member(v, list1) \rightarrow Member(v, e.list))$ $\rightarrow$E (2)(3)
(5) $Member(e2, list1) \rightarrow Member(e2, e.list)$ $\forall$E (4)
$\quad \ldots$
(n-4) $Member(e2, e1.list1) \rightarrow Member(e2, e.list)$
(n-3) $\forall v\,(Member(v, e1.list1) \rightarrow Member(v, e.list))$ $\forall$I (n-4)

Both the steps (5) and (n-4) concern the relation *Member*. We instantiate in the definition of list membership.

$$List(e1.list1) \rightarrow (Member(e2, e1.list1) \leftrightarrow e2 = e1 \vee Member(e2, list1))$$

Here we shall have to introduce yet another condition for the program clause, $List(e1.list1)$. After adding this condition as hypothesis, we can have

(6) $List(e1.list1)$ hyp
(7) $List(e1.list1) \rightarrow$ $\forall$E $(d10)$
$\quad (Member(e2, e1.list1) \leftrightarrow e2 = e1 \vee Member(e2, list1))$
(8) $Member(e2, e1.list1) \leftrightarrow e2 = e1 \vee Member(e2, list1)$ $\rightarrow$E (6)(7)
(9) $Member(e2, e1.list1) \rightarrow e2 = e1 \vee Member(e2, list1)$ $\leftrightarrow$E (8)

Step (n-4) which we now want to deduce has the form of an implication. We assume the antecedent to carry out a hypothetical reasoning.

(k) $Member(e2, e1.list1)$
$\quad \ldots$
(n-5) $Member(e2, e.list)$
(n-4) $Member(e2, e1.list1) \rightarrow Member(e2, e.list)$ $\rightarrow$I (k)(n-5)

The hypothesis (k) can be used to get the definiens from the definition of *Member* in step (l). We continue the derivation from a disjunction by reasoning from the different parts of the disjunction, as can be seen from the rule below.

∨E)

$$\frac{A \vee B \quad \begin{matrix}(A)\\ C\end{matrix} \quad \begin{matrix}(B)\\ C\end{matrix}}{C}$$

We first assume the first part of the disjunction

(l)	$e2 = e1 \vee Member(e2, list1)$	→E (11)(k)
(m)	$e2 = e1$	hyp

There is no way in which we can continue the derivation from this clause without making an addition to the conditions of the Horn clause. From step (m) we want to deduce step (n-5). $e1$ is the first element of the first list in the relation *Subset*. All elements in the first list shall occur in the second for the *Subset* relation to be satisfied. Let us add the condition that $e1$ shall be a member of $e.list$. We can then deduce $Member(e2, e.list)$ under the hypothesis $e2 = e1$. The second hypothesis for ∨-elimination is $Member(e2, list1)$. We can obtain $Member(e2, e.list)$ from step (5). We have now proven $Member(e2, e.list)$ from both of the two parts of the disjunction. According to the rules for ∨-elimination we can then draw the conclusion in step (19).

(10)	$Member(e1, e.list)$	hyp
(11)	$Member(e2, e1.list1)$	hyp
(12)	$e2 = e1 \vee Member(e2, list1)$	→E (9)(11)
(13)	$e2 = e1$	hyp
(14)	$Member(e2, e.list)$	id (10)(13)
(15)	$e2 = e1 \rightarrow Member(e2, e.list)$	→I (13)(14)
(16)	$Member(e2, list1)$	hyp
(17)	$Member(e2, e.list)$	→E (5)(16)
(18)	$Member(e2, list1) \rightarrow Member(e2, e.list)$	→I (16)(17)
(19)	$Member(e2, e.list)$	∨E (12)(15)(18)

Step (19) was deduced under the hypothesis in step (11). We can make a → -introduction followed by a ∀ -introduction to obtain the formula we were interested in deducing.

(20)	$Member(e2, e1.list1) \rightarrow Member(e2, e.list)$	→I (11)(19)
(21)	$\forall v\,(Member(v, e1.list1) \rightarrow Member(v, e.list))$	∀I (20)

We have assembled all the parts of the derivation. The new conditions we have introduced are $List(e1.list1)$ and $Member(e1, e.list)$. The entire proof is shown in Figure 5.3.

(1) $Subset(e1.list1, e.list) \leftarrow$ $\forall v\,(Member(v, e1.list1) \rightarrow Member(v, e.list))$	$\forall$E ,$\leftrightarrow$E $(s3)$
(2) $Subset(list1, e.list)$	hyp
(3) $Subset(list1, e.list) \rightarrow$ $\forall v\,(Member(v, list1) \rightarrow Member(v, e.list))$	$\forall$E ,$\leftrightarrow$E $(s3)$
(4) $\forall v\,(Member(v, list1) \rightarrow Member(v, e.list))$	$\rightarrow$E (2)(3)
(5) $Member(e2, list1) \rightarrow Member(e2, e.list)$	$\forall$E (4)
(6) $List(e1.list1)$	hyp
(7) $List(e1.list1) \rightarrow$ $(Member(e2, e1.list1) \leftrightarrow e2 = e1 \vee Member(e2, list1))$	$\forall$E $(d10)$
(8) $Member(e2, e1.list1) \leftrightarrow e2 = e1 \vee Member(e2, list1)$	$\rightarrow$E (6)(7)
(9) $Member(e2, e1.list1) \rightarrow e2 = e1 \vee Member(e2, list1)$	$\leftrightarrow$E (8)
(10) $Member(e1, e.list)$	hyp
(11) $Member(e2, e1.list1)$	hyp
(12) $e2 = e1 \vee Member(e2, list1)$	$\rightarrow$E (9)(11)
(13) $e2 = e1$	hyp
(14) $Member(e2, e.list)$	id (10)(13)
(15) $e2 = e1 \rightarrow Member(e2, e.list)$	$\rightarrow$I (13)(14)
(16) $Member(e2, list1)$	hyp
(17) $Member(e2, e.list)$	$\rightarrow$E (7)(16)
(18) $Member(e2, list1) \rightarrow Member(e2, e.list)$	$\rightarrow$I (16)(17)
(19) $Member(e2, e.list)$	$\vee$E (12)(15)(18)
(20) $Member(e2, e1.list1) \rightarrow Member(e2, e.list)$	$\rightarrow$I (11)(19)
(21) $\forall v\,(Member(v, e1.list1) \rightarrow Member(v, e.list))$	$\forall$I (20)
(22) $Subset(e1.list1, e.list)$	$\leftarrow$E (1)(21)
(23) $Subset(e1.list1, e.list) \leftarrow$ $Subset(list1, e.list) \wedge List(e1.list1) \wedge Member(e1, e.list)$	$\leftarrow$I (2)(6)(10)(22)
(24) $\forall u \forall z \forall x \forall y\,(Subset(x.y, u.z) \leftarrow$ $Subset(y, u.z) \wedge List(x.y) \wedge Member(x, u.z))$	$\forall$I (23)

Figure 5.3: Derivation of the recursive program clause for the relation subset.

Type controls such as $List(x.y)$ are left out of the program.

```
Subset(∅,z) ←
Subset(x.y,u.z) ←
        Subset(y,u.z), Member(x,u.z)
```

In Round 2 we learnt how to obtain a result from a logic program by derivation. In this round we have seen how we can construct programs by derivations. We will now investigate how we can show properties of programs and how to transform programs. Let us make an outline of each of the different types of derivations that are of interest in logic programming.

- A program P_{prog} is derived by a derivation Π_i from definitions of data structures D and specifications of relations P_{spec}.

$$\frac{(D) \quad (P_{spec})}{\begin{array}{c}\Pi_i \\ P_{prog}\end{array}}$$

The program is a formal expression in the language of the calculus, i.e. the program is an object in the calculus. We can make other interesting derivations from the program.

- A program P_{prog} has a particular property; this can be shown by deriving theorem T for the property from D and P_{spec}.

$$\frac{(D) \quad (P_{spec}) \quad (P_{prog})}{\begin{array}{c}\Pi_{ii} \\ T\end{array}}$$

- A program P'_{prog} can be transformed from another P_{prog} and the definitions D and P_{spec}.

$$\frac{(D) \quad (P_{spec}) \quad (P_{prog})}{\begin{array}{c}\Pi_{iii} \\ P'_{prog}\end{array}}$$

- A conclusion C is derived from a program.

$$\frac{(P_{prog})}{\begin{array}{c}\Pi_{iv} \\ C\end{array}}$$

5.2 Proof of Program Properties

5.2.1 Completeness

In Sect. 5.1, we derived the program *Subset* from its specification. In this way we obtained a partially correct program as regards the specification, i.e. we know that the results computed from the program are valid according to the specification. But we cannot be certain that the program computes all values that hold according to the specification. To ensure this we will have to show a theorem for completeness.

We will express the fact that the program *Subset* is complete as regards the specification. The relation *Subset* shall hold for all lists w and z such that all elements in w shall also be present in z. We express this in standard form.

$$\forall w \forall z \, (\forall v \, (Member(v, w) \rightarrow Member(v, z)) \rightarrow \; Subset(w, z))$$

When proving the theorem in the programming calculus we use axioms describing the list structure, the program specification, and the program. The derivation is of the type called Π_{ii} in the preceding section.

The program *Subset* we derived above looks like this:

$$\forall z\, Subset(\emptyset, z)$$
$$\forall x \forall y \forall z\, (Subset(x.y, z) \leftarrow Subset(y, z) \wedge Member(x, z))$$

In the conditional part in the second clause there is a relation *Member* between an element and a list. In the programming calculus there is a definition of the *Member* relation that corresponds directly to the program we expressed in Horn form in Round 3.

$$\forall x \forall y\, Member(x, x.y)$$
$$\forall x \forall y \forall z\, (Member(x, y.z) \leftarrow Member(x, z))$$

We will prove, through induction, that the theorem $\forall w \forall z\, (\forall v\, (Member(v, w) \rightarrow Member(v, z)) \rightarrow Subset(w, z))$ holds for all lists. We use the induction schema for lists and carry the inductive reasoning over the first argument. We get two cases, a base case and a case for the induction step. In the base case we shall derive the theorem where we have substituted $\emptyset$ for w

$$\forall z\, (\forall v\, (Member(v, \emptyset) \rightarrow Member(v, z)) \rightarrow Subset(\emptyset, z))$$

In the induction step we assume that the theorem holds for an arbitrary list y, and show that under this assumption it also holds for a list $x.y$ that contains one more element. We instantiate P in the induction schema with the theorem we shall prove and thereafter we substitute y for w in the theorem to get the conditional of the induction step. Furthermore, we substitute $x.y$ for w in the theorem to get the consequence of the induction step.

$$\forall x \forall y\, (\forall z\, (\forall v\, (Member(v, y) \rightarrow Member(v, z)) \rightarrow Subset(y, z)) \rightarrow$$
$$\forall z\, (\forall v\, (Member(v, x.y) \rightarrow Member(v, z)) \rightarrow Subset(x.y, z)))$$

We begin by deriving the base case. The logical clause to be proven for the base case is a universally quantified implication: $\forall x_1 \ldots \forall x_n (P \rightarrow Q)$. We can outline the proof from the form of the theorem. To prove the formula $\forall x_1 \ldots \forall x_n\, (P \rightarrow Q)$ we assume the hypothesis P, derive Q, arrive at the clause $P \rightarrow Q$ through $\rightarrow$I , and finally apply $\forall$I to get the theorem.

(1)	P	hyp
	...	
(n-2)	Q	
(n-1)	$P \rightarrow Q$	$\rightarrow$I (1)(n-2)
(n)	$\forall x_1 \ldots \forall x_n (P \rightarrow Q)$	$\forall$I (n-1)

We apply the reasoning to the derivation of the base case for our theorem about *Subset*:

$$
\begin{array}{|ll}
(1)\ \forall v\,(Member(v,\emptyset) \rightarrow Member(v,z)) & \text{hyp} \\
\quad \ldots & \\
(\text{n-2})\ Subset(\emptyset,z) & \\
\hline
(\text{n-1})\ \forall v\,(Member(v,\emptyset) \rightarrow Member(v,z)) \rightarrow Subset(\emptyset,z) & \rightarrow\text{I}\ (1)(\text{n-2}) \\
(\text{n})\ \forall z\,(\forall v\,(Member(v,\emptyset) \rightarrow Member(v,z)) \rightarrow Subset(\emptyset,z)) & \forall\text{I}\ (\text{n-1})
\end{array}
$$

In the outline of the proof the derivation of step (n-2) is missing. We notice that *Subset* can be directly derived from the first clause in the program *Subset* through ∀E . We are thus finished with the derivation of the base case of the theorem.

$$
\begin{array}{|ll}
(1)\ \forall v\,(Member(v,\emptyset) \rightarrow Member(v,z)) & \text{hyp} \\
(2)\ Subset(\emptyset,z) & \forall\text{E program} \\
\hline
(3)\ \forall v\,(Member(v,\emptyset) \rightarrow Member(v,z)) \rightarrow Subset(\emptyset,z) & \rightarrow\text{I}\ (1)(2) \\
(4)\ \forall z\,(\forall v\,(Member(v,\emptyset) \rightarrow Member(v,z)) \rightarrow Subset(\emptyset,z)) & \forall\text{I}\ (3)
\end{array}
$$

The induction step remains to be derived. It has also the form of a universally quantified implication: $\forall x \forall y\,(P(y) \rightarrow P(x.y))$. The outline of the proof takes on the typical form for a derivation of a formula with this logical form.

$$
\begin{array}{|ll}
(1)\ P(y) & \text{hyp} \\
\quad \ldots & \\
(\text{n-2})\ P(x.y) & \\
\hline
(\text{n-1})\ P(y) \rightarrow P(x.y) & \rightarrow\text{I}\ (1)(\text{n-2}) \\
(\text{n})\ \forall x \forall y\,(P(y) \rightarrow P(x.y)) & \forall\text{I}\ (\text{n-1})
\end{array}
$$

In our proof of the theorem the condition $P(y)$ will be instantiated to the formula

$$\forall z\,(\forall v\,(Member(v,y) \rightarrow Member(v,z)) \rightarrow Subset(y,z))$$

The consequent $P(x.y)$ will be instantiated to the formula

$$\forall z\,(\forall v\,(Member(v,x.y) \rightarrow Member(v,z)) \rightarrow Subset(x.y,z))$$

We reason in the same manner as before for the outline of the derivation of the induction step.

$$
\begin{array}{|ll}
(1)\ \forall z\,(\forall v\,(Member(v,y) \rightarrow Member(v,z)) \rightarrow Subset(y,z)) & \text{hyp} \\
\quad \ldots & \\
(\text{n-2})\ \forall z\,(\forall v\,(Member(v,x.y) \rightarrow Member(v,z)) \rightarrow Subset(x.y,z)) & \\
\hline
(\text{n-1})\ \forall z\,(\forall v\,(Member(v,y) \rightarrow Member(v,z)) \rightarrow & \rightarrow\text{I}\ (1)(\text{n-2}) \\
\qquad Subset(y,z)) \rightarrow & \\
\qquad \forall z\,(\forall v\,(Member(v,x.y) \rightarrow Member(v,z)) \rightarrow Subset(x.y,z)) & \\
(\text{n})\ \forall x \forall y\,(\forall z\,(\forall v\,(Member(v,y) \rightarrow Member(v,z)) \rightarrow & \forall\text{I}\ (\text{n-1}) \\
\qquad\qquad Subset(y,z)) \rightarrow & \\
\qquad \forall z\,(\forall v\,(Member(v,x.y) \rightarrow Member(v,z)) \rightarrow Subset(x.y,z))) &
\end{array}
$$

We notice that (n-2) has the form of a universally quantified implication and add steps (2), (n-4), and (n-3) which will result in (n-2) if we succeed in deriving step (n-4).

	(2) $\forall v\,(Member(v, x.y) \rightarrow Member(v, z))$	hyp
	...	
	(n-4) $Subset(x.y, z)$	
	(n-3) $\forall v\,(Member(v, x.y) \rightarrow Member(v, z)) \rightarrow Subset(x.y, z)$	$\rightarrow$I (2) (n-4)
	(n-2) $\forall z\,(\forall v\,(Member(v, x.y) \rightarrow Member(v, z)) \rightarrow Subset(x.y, z))$	$\forall$I (n-3)

To derive $Subset(x.y, z)$ we can use the second clause in the definition of the program *Subset*. Our new goal is to derive a conjunction.

$$Subset(y, z) \wedge Member(x, z)$$

We can obtain the first part of the conjunction, $Subset(y, z)$, from step (1) if we can derive $Member(v, y) \rightarrow Member(v, z)$. We assume $Member(v, y)$ and use the second clause in the definition of *Member* as well as step (2), and can derive $Member(v, z)$ from these steps.

	(1) $\forall z\,(\forall v\,(Member(v, y) \rightarrow Member(v, z)) \rightarrow Subset(y, z))$	hyp
	(2) $\forall v\,(Member(v, x.y) \rightarrow Member(v, z))$	hyp
	(3) $Member(v, y)$	hyp
	(4) $Member(v, x.y)$	$\rightarrow$E (3) Def
	(5) $Member(v, x.y) \rightarrow Member(v, z)$	$\forall$E (2)
	(6) $Member(v, z)$	$\rightarrow$E (4)(5)
	(7) $Member(v, y) \rightarrow Member(v, z)$	$\rightarrow$I (3)(6)
	(8) $\forall v\,(Member(v, y) \rightarrow Member(v, z))$	$\forall$I (7)
	(9) $\forall v\,(Member(v, y) \rightarrow Member(v, z)) \rightarrow Subset(y, z)$	$\forall$E (1)
	(10) $Subset(y, z)$	$\rightarrow$E (8)(9)

We can get the second part of the conjunction, $Member(x, z)$, by using the first clause in the definition of *Member*. The clause $\forall x \forall y\, Member(x, x.y)$ and step (2) will thus give us $Member(x, z)$ in steps (11) – (13).

	(11) $Member(x, x.y)$	$\forall$E Def
	(12) $Member(x, x.y) \rightarrow Member(x, z)$	$\forall$E (2)
	(13) $Member(x, z)$	$\rightarrow$E (11)(12)

Taken together these two partial derivations will result in the steps we need to be able to complete the proof. The entire proof of the induction step is shown in Figure 5.4.

	(14) $Subset(y, z) \wedge Member(x, z)$	$\wedge$I (10)(13)
	(15) $Subset(y, z) \wedge Member(x, z) \rightarrow Subset(x.y, z)$	$\forall$E Def
	(16) $Subset(x.y, z)$	$\rightarrow$E (14)(15)

(1)	$\forall z\,(\forall v\,(Member(v,y) \rightarrow Member(v,z)) \rightarrow Subset(y,z))$	hyp
(2)	$\forall v\,(Member(v,x.y) \rightarrow Member(v,z))$	hyp
(3)	$Member(v,y)$	hyp
(4)	$Member(v,x.y)$	$\rightarrow$E (3) Def
(5)	$Member(v,x.y) \rightarrow Member(v,z)$	$\forall$E (2)
(6)	$Member(v,z)$	$\rightarrow$E (4)(5)
(7)	$Member(v,y) \rightarrow Member(v,z)$	$\rightarrow$I (3)(6)
(8)	$\forall v\,(Member(v,y) \rightarrow Member(v,z))$	$\forall$I (7)
(9)	$\forall v\,(Member(v,y) \rightarrow Member(v,z)) \rightarrow Subset(y,z)$	$\forall$E (1)
(10)	$Subset(y,z)$	$\rightarrow$E (8)(9)
(11)	$Member(x,x.y)$	$\forall$E Def
(12)	$Member(x,x.y) \rightarrow Member(x,z)$	$\forall$E (2)
(13)	$Member(x,z)$	$\rightarrow$E (11)(12)
(14)	$Subset(y,z) \wedge Member(x,z)$	$\wedge$I (10)(13)
(15)	$Subset(y,z) \wedge Member(x,z) \rightarrow Subset(x.y,z)$	$\forall$E Def
(16)	$Subset(x.y,z)$	$\rightarrow$E (14)(15)
(17)	$\forall v\,(Member(v,x.y) \rightarrow Member(v,z)) \rightarrow Subset(x.y,z)$	$\rightarrow$I (2)(16)
(18)	$\forall z\,(\forall v\,(Member(v,x.y) \rightarrow Member(v,z)) \rightarrow Subset(x.y,z))$	$\forall$I (17)
(19)	$\forall z\,(\forall v\,(Member(v,y) \rightarrow Member(v,z)) \rightarrow Subset(y,z)) \rightarrow \forall z\,(\forall v\,(Member(v,x.y) \rightarrow Member(v,z)) \rightarrow Subset(x.y,z))$	$\rightarrow$I (1)(18)
(20)	$\forall x \forall y\,(\forall z\,(\forall v\,(Member(v,y) \rightarrow Member(v,z)) \rightarrow Subset(y,z)) \rightarrow \forall z\,(\forall v\,(Member(v,x.y) \rightarrow Member(v,z)) \rightarrow Subset(x.y,z)))$	$\forall$I (19)

Figure 5.4: The induction step in a proof of completeness for the relation subset.

5.2.2 Termination

We will study the relation *Insert* between a node and two ordered binary trees. We express the relation in standard form.

$$\begin{aligned}\forall x \forall u \forall y\,(&Insert(u,x,y) \leftrightarrow \\ &\forall v\,(Member(v,x) \vee v = u \leftrightarrow Member(v,y)) \\ &\wedge Ord\_binary\_tree(x) \wedge Ord\_binary\_tree(y))\end{aligned}$$

In the specification we use *Member* and *Ord_binary_tree* which we defined in Sects. 5.1 and 3.3. A program *Insert* for ordered binary trees was also

formulated. We express the same program in standard form.

$$\forall u\, Insert(u, \emptyset, T(\emptyset, u, \emptyset))$$
$$\forall u \forall x \forall y\, Insert(u, T(x, u, y), T(x, u, y))$$
$$\forall u \forall x \forall r \forall y \forall nx\, (Insert(u, T(x, r, y), T(nx, r, y)) \leftarrow u < r \wedge Insert(u, x, nx))$$
$$\forall u \forall x \forall r \forall y \forall ny\, (Insert(u, T(x, r, y), T(x, r, ny)) \leftarrow u > r \wedge Insert(u, y, ny))$$

We cannot derive all the theorems from the clauses in the *Insert* program that we can derive from the *Insert* specification; the *Insert* program does not compute the same values as the specification. The values the program computes are correct, but it does not compute all the values that the specification computes. Let us look at an example of a valid conclusion drawn from the specification that cannot be drawn from the program clauses.

$$Insert(2, T(T(\emptyset, 1, \emptyset), 3, \emptyset), T(T(\emptyset, 1, \emptyset), 2, T(\emptyset, 3, \emptyset)))$$

The *Insert* relation holds, according to the specification, for a node 2, an ordered binary tree consisting of the nodes 1 and 3, and an ordered binary tree consisting of the nodes 1, 2, and 3, where 2 is the root. But the same relation does not hold according to the program clauses which, moreover, require that the node that constitutes the first argument must be a leaf in the tree that forms the third argument.

If we use the program to insert a node in a binary tree, we are restricted to only inserting nodes as new leaves in the tree. According to the specification a node can be inserted anywhere in the tree. Thus the specification yields many different trees, while the program can only give one tree. From a specification of this type, several different programs that insert nodes in different ways can be derived. To express the program in the way we have means a short, i.e. an efficient, evaluation, as we do not have to move any nodes for the third argument to be an ordered tree.

Thus, we cannot show, which we did for the relation *Subset*, that the program is complete according to the specification. We cannot show that the program yields all the answers the specification yields, but we can show that it will always result in some answer. That is, we can show that the program will *terminate*, given arbitrary valid input data.

More often than not we do have a program that computes parts of the specification; we very often want a program to be more efficient than the specification. We are thus not always interested in showing completeness but we are nearly always interested in showing the less restricted termination condition. If we have shown that a program terminates and have derived the program formally from the specification or shown a theorem for partial correctness, we say that the program is correct according to the specification.

When we show that a program satisfies a termination condition, we often show that it terminates according to some input-output pattern. Should we

want to use the program in another way, with some other input-output patterns, we also have to show termination theorems expressed with regard to these applications.

We will formulate a termination theorem for *Insert* for use of the program with the first two arguments instantiated. We suppose that we use the program for inserting an element into an ordered binary tree and then we want to show that for any ordered binary tree and for any node there always exists a new tree such that the *Insert* relation holds.

$$\forall x \forall u \, (Ord\_binary\_tree(x) \wedge Node(u) \rightarrow \exists y \, Insert(u, x, y))$$

We can show the theorem by induction for binary trees over the first argument in *Insert*.

5.2.3 Partial Correctness

Suppose that we have the following definition of the relation *Intersection* between three sets represented by lists:

$$\begin{aligned}
&\forall x \forall y \forall z \, (Intersection(x, y, z) \leftrightarrow \\
&\quad x = \emptyset \wedge z = \emptyset \\
&\quad \vee \exists u \exists x1 \exists z1 \, (x = u.x1 \\
&\qquad \wedge ((Member(u, y) \wedge z = u.z1 \wedge Intersection(x1, y, z1)) \\
&\qquad \vee (\neg Member(u, y) \wedge Intersection(x1, y, z)))))
\end{aligned}$$

We see this definition as a program although it is not in Horn form. We note that it is easy to convert it into Horn form since the definiens is built up by $\vee$, $\exists$, and $\wedge$. When we show program properties, the Horn form does not always suffice; we need the only-if part of the definition, i.e. the completed program. This is the case since when we prove that, given a program, a certain property holds, we need to be able to derive that if something holds for the defined relation, it also holds for the antecedent (the definiens).

When we express definitions of programs without deriving them formally, we cannot be quite sure that the program follows from the specification. To make sure that a program computes values according to the specification, we show a theorem for *partial correctness*.

We saw the specification for the *Intersection* relation in Sect. 5.1.3 above.

$$\begin{aligned}
&\forall x \forall y \forall z \, (Intersection(x, y, z) \leftrightarrow \\
&\qquad \forall v \, (Member(v, x) \wedge Member(v, y) \leftrightarrow Member(v, z)))
\end{aligned}$$

We want to show that the values computed by the program are correct according to the specification and express a theorem for partial correctness. For all x, y, and z, if the relation *Intersection* holds, all elements in z shall be found in both x and y.

$$\begin{aligned}
&\forall x \forall y \forall z \, (Intersection(x, y, z) \rightarrow \\
&\qquad \forall v \, (Member(v, x) \wedge Member(v, y) \leftrightarrow Member(v, z)))
\end{aligned}$$

The theorem expresses a logical clause about a relation between three lists. In order to prove it we reason inductively over lists and, consequently, get two cases to prove. Let us begin with the base case. We instantiate the induction schema with our theorem for partial correctness and substitute $\emptyset$ for x to obtain the theorem for the base case.

$$\forall y \forall z \, (Intersection(\emptyset, y, z) \rightarrow \\ \forall v \, (Member(v, \emptyset) \wedge Member(v, y) \leftrightarrow Member(v, z)))$$

We derive the theorem from axioms about the list structure, the program *Intersection*, and the specification for *Member*. In the derivation we use some lemmas derived from the axioms about the relations we reason about. In this way we shorten the proof.

We make use of the first lemma for *Member*, i.e. $(d9)$ in Sect. 5.1.3 for the case when the list is empty.

$$\forall x \neg Member(x, \emptyset)$$

We also use a lemma for the case when the first argument in *Intersection* is the empty list.

$$\forall y \forall z \, (Intersection(\emptyset, y, z) \leftrightarrow z = \emptyset)$$

The derivation of the base case will in this way appear as in Figure 5.5.

We instantiate the induction schema for lists with our theorem for partial correctness and obtain the formula we shall show for the induction step from x to $u.x$.

$$\forall x \forall u (\forall y \forall z \, (Intersection(x, y, z) \rightarrow \\ \forall v \, (Member(v, x) \wedge Member(v, y) \leftrightarrow Member(v, z))) \\ \rightarrow \forall y \forall z (Intersection(u.x, y, z) \rightarrow \\ \forall v \, (Member(v, u.x) \wedge Member(v, y) \leftrightarrow Member(v, z))))$$

We outline the proof in the same manner as before, see figure 5.6a. Steps (1) – (3), (k-1), (k), (l), and (n-7) – (n-4) are rendered directly from the form of the formula.

The derivations of (k-1) and (n-7) will be lengthy, so we decide to carry them out separately and refer to them as Lemma 1 and Lemma 2, respectively. As in the proof of the base case, we use lemmas here for *Member* and for *Intersection*.

$$\forall u \forall v \forall x \, (Member(u, v.x) \leftrightarrow u = v \vee Member(u, x))$$

Besides this, a lemma for *Member* expressing that the first element is member of a list shortens the proof.

$$\forall u \forall x \; Member(u, u.x)$$

(1)	$Intersection(\emptyset, y, z)$	hyp
(2)	$Member(v, \emptyset) \wedge Member(v, y)$	hyp
(3)	$Member(v, \emptyset)$	$\wedge$E (2)
(4)	$\neg Member(v, \emptyset)$	$\forall$E lemma
(5)	$\bot$	$\neg$E (3)(4)
(6)	$Member(v, z)$	$\bot_{int}$ (5)
(7)	$Member(v, \emptyset) \wedge Member(v, y) \rightarrow Member(v, z)$	$\rightarrow$I (2)(6)
(8)	$Member(v, z)$	hyp
(9)	$Intersection(\emptyset, y, z) \leftrightarrow z = \emptyset$	$\forall$E lemma
(10)	$Intersection(\emptyset, y, z) \rightarrow z = \emptyset$	$\leftrightarrow$E (9)
(11)	$z = \emptyset$	$\rightarrow$E (1)(10)
(12)	$Member(v, \emptyset)$	id (8)(11)
(13)	$\neg Member(v, \emptyset)$	$\forall$E lemma
(14)	$\bot$	$\neg$E (12)(13)
(15)	$Member(v, \emptyset) \wedge Member(v, y)$	$\bot_{int}$ (14)
(16)	$Member(v, z) \rightarrow Member(v, \emptyset) \wedge Member(v, y)$	$\rightarrow$I (8)(15)
(17)	$Member(v, \emptyset) \wedge Member(v, y) \leftrightarrow Member(v, z)$	$\leftrightarrow$I (7)(16)
(18)	$\forall v\,(Member(v, \emptyset) \wedge Member(v, y) \leftrightarrow Member(v, z))$	$\forall$I (17)
(19)	$Intersection(\emptyset, y, z) \rightarrow$ $\forall v(Member(v, \emptyset) \wedge Member(v, y) \leftrightarrow Member(v, z))$	$\rightarrow$I (1)(18)
(20)	$\forall y \forall z\,(Intersection(\emptyset, y, z) \rightarrow$ $\forall v\,(Member(v, \emptyset) \wedge Member(v, y) \leftrightarrow Member(v, z)))$	$\forall$I (19)

Figure 5.5: The base case for a proof of partial correctness of the relation intersection.

At the end, in the induction step, we need the lemma for the recursive case of the program *Intersection*. The entire proof is shown in Figure 5.6.

$$\forall u \forall x \forall y \forall z 1 \forall z\,(Intersection(u.x, y, z1) \leftrightarrow$$
$$Member(u, y) \wedge z1 = u.z \wedge Intersection(x, y, z)$$
$$\vee \neg Member(u, y) \wedge z1 = z \wedge Intersection(x, y, z))$$

We will show Lemma 1 looks like this.

$$\forall x \forall y \forall z 1 \forall v \forall u$$
$$(\forall y \forall z\,(Intersection(x, y, z) \rightarrow$$
$$\forall w(Member(w, x) \wedge Member(w, y) \leftrightarrow Member(w, z)))$$
$$\wedge\, Member(v, u.x) \wedge Member(v, y)$$
$$\wedge\, Intersection(u.x, y, z1) \rightarrow Member(v, z1))$$

We begin by assuming the conditional part of the formula, step (1). This lemma is part of the proof of the induction step and we use the recursive clause

(1) $\forall y \forall z\,(Intersection(x,y,z) \rightarrow$ hyp
$\quad \forall v\,(Member(v,x) \wedge Member(v,y) \leftrightarrow Member(v,z)))$
(2) $Intersection(u.x,y,z1)$ hyp
(3) $Member(v,u.x) \wedge Member(v,y)$ hyp
$\quad \ldots$
(k-1) $Member(v,z1)$
(k) $Member(v,u.x) \wedge Member(v,y) \rightarrow Member(v,z1)$ →I (3)(k-1)
(l) $Member(v,z1)$ hyp
$\quad \ldots$
(n-7) $Member(v,u.x) \wedge Member(v,y)$
(n-6) $Member(v,z1) \rightarrow Member(v,u.x) \wedge Member(v,y)$ →I (l)(n-7)
(n-5) $Member(v,u.x) \wedge Member(v,y) \leftrightarrow Member(v,z1)$ ↔I (k)(n-6)
(n-4) $\forall v(Member(v,u.x) \wedge Member(v,y) \leftrightarrow Member(v,z1))$ ∀I (n-5)
(n-3) $Intersection(u.x,y,z1) \rightarrow$ →I (2)(n-4)
$\quad \forall v\,(Member(v,u.x) \wedge Member(v,y) \leftrightarrow Member(v,z1))$
(n-2) $\forall y \forall z\,(Intersection(u.x,y,z) \rightarrow$ ∀I (n-3)
$\quad \forall v\,(Member(v,u.x) \wedge Member(v,y) \leftrightarrow Member(v,z)))$
(n-1) $\forall y \forall z\,(Intersection(x,y,z) \rightarrow$ →I (1)(n-2)
$\quad \forall v\,(Member(v,x) \wedge Member(v,y) \leftrightarrow Member(v,z))) \rightarrow$
$\quad \forall y \forall z\,(Intersection(u.x,y,z) \rightarrow$
$\quad \forall v\,(Member(v,u.x) \wedge Member(v,y) \leftrightarrow Member(v,z)))$
(n) $\forall x \forall u$ ∀I (n-1)
$\quad (\forall y \forall z\,(Intersection(x,y,z) \rightarrow$
$\quad \forall v\,(Member(v,x) \wedge Member(v,y) \leftrightarrow Member(v,z))) \rightarrow$
$\quad \forall y \forall z\,(Intersection(u.x,y,z) \rightarrow$
$\quad \forall v\,(Member(v,u.x) \wedge Member(v,y) \leftrightarrow Member(v,z))))$

Figure 5.6a: Outline of the induction step in a proof of partial correctness for the relation intersection.

in the program, step (2), and the recursive lemma for *Member*, step (7). We will arrive at $v = u \vee Member(v,x)$

(1) $\forall y \forall z\,(Intersection(x,y,z) \rightarrow$ hyp
$\quad \forall v\,(Member(v,x) \wedge Member(v,y) \leftrightarrow Member(v,z)))$
$\quad \wedge Member(v,u.x) \wedge Member(v,y) \wedge Intersection(u.x,y,z1)$
(2) $Intersection(u.x,y,z1) \leftrightarrow$ ∀E Intersection
$\quad Member(u,y) \wedge z1 = u.z \wedge Intersection(x,y,z)$
$\quad \vee \neg Member(u,y) \wedge z1 = z \wedge Intersection(x,y,z)$
(3) $Intersection(u.x,y,z1) \rightarrow$ ↔E (2)
$\quad Member(u,y) \wedge z1 = u.z \wedge Intersection(x,y,z)$
$\quad \vee \neg Member(u,y) \wedge (z1 = z \wedge Intersection(x,y,z))$
(4) $Intersection(u.x,y,z1)$ ∧E (1)

(1) $\forall y \forall z\,(Intersection(x,y,z) \rightarrow$ hyp
$\quad \forall v\,(Member(v,x) \wedge Member(v,y) \leftrightarrow Member(w,z)))$

(2) $Intersection(u.x,y,z1)$ hyp

(3) $Member(v,u.x) \wedge Member(v,y)$ hyp

(4) $\forall y \forall z\,(Intersection(x,y,z) \rightarrow$ $\wedge$I (1)(2)(3)
$\quad \forall w(Member(w,x) \wedge Member(w,y) \leftrightarrow Member(v,z)))$
$\quad \wedge\, Member(v,u.x) \wedge Member(v,y) \wedge Intersection(u.x,y,z)$

(5) $\forall y \forall z\,(Intersection(x,y,z) \rightarrow$ $\forall$E Lemma 1
$\quad \forall v(Member(v,x) \wedge Member(v,y) \leftrightarrow Member(v,z)))$
$\quad \wedge\, Member(v,u.x) \wedge Member(v,y)$
$\quad \vee\, Intersection(u.x,y,z1) \rightarrow Member(v,z1)$

(6) $Member(v,z1)$ $\rightarrow$E (4)(5)

(7) $Member(v,u.x) \wedge Member(v,y) \rightarrow Member(v,z1)$ $\rightarrow$I (3)(6)

(8) $Member(v,z1)$ hyp

(9) $\forall y \forall z\,(Intersection(x,y,z) \rightarrow$ $\wedge$I (1)(2)(8)
$\quad \forall v(Member(v,x) \wedge Member(v,y) \leftrightarrow Member(v,z))) \wedge$
$\quad Member(v,z1) \wedge Intersection(u.x,y,z1)$

(10) $\forall y \forall z\,(Intersection(x,y,z) \rightarrow$ $\forall$E Lemma 2
$\quad \forall v(Member(v,x) \wedge Member(v,y) \leftrightarrow Member(v,z))) \wedge$
$\quad Member(v,z1) \wedge Intersection(u.x,y,z1) \rightarrow$
$\quad Member(v,u.x) \wedge Member(v,y)$

(11) $Member(v,u.x) \wedge Member(v,y)$ $\rightarrow$E (9)(10)

(12) $Member(v,z1) \rightarrow Member(v,u.x) \wedge Member(v,y)$ $\rightarrow$I (8)(12)

(13) $Member(v,u.x) \wedge Member(v,y) \leftrightarrow Member(v,z1)$ $\leftrightarrow$I (7)(12)

(14) $\forall v\,(Member(v,u.x) \wedge Member(v,y) \leftrightarrow Member(v,z1))$ $\forall$I (13)

(15) $Intersection(u.x,y,z1) \rightarrow$ $\rightarrow$I (2)(14)
$\quad \forall v(Member(v,u.x) \wedge Member(v,y) \leftrightarrow Member(v,z1))$

(16) $\forall y \forall z\,(Intersection(u.x,y,z) \rightarrow$ $\forall$I (15)
$\quad \forall v(Member(v,u.x) \wedge Member(v,y) \leftrightarrow Member(v,z)))$

(17) $\forall y \forall z(Intersection(x,y,z) \rightarrow$ $\rightarrow$I (1)(16)
$\quad \forall v(Member(v,x) \wedge Member(v,y) \leftrightarrow Member(v,z))) \rightarrow$
$\forall y \forall z\,(Intersection(u.x,y,z) \rightarrow$
$\quad \forall v(Member(v,u.x) \wedge Member(v,y) \leftrightarrow Member(v,z)))$

(18) $\forall x \forall u$ $\forall$I (17)
$(\forall y \forall z\,(Intersection(x,y,z) \rightarrow$
$\quad \forall v\,(Member(v,x) \wedge Member(v,y) \leftrightarrow Member(v,z))) \rightarrow$
$\forall y \forall z\,(Intersection(u.x,y,z) \rightarrow$
$\quad \forall v\,(Member(v,u.x) \wedge Member(v,y) \leftrightarrow Member(v,z))))$

Figure 5.6: The induction step in a proof of partial correctness for the relation intersection.

(5)	$Member(u,y) \wedge z1 = u.z \wedge Intersection(x,y,z)$ $\vee \neg Member(u,y) \wedge z1 = z \wedge Intersection(x,y,z)$	$\rightarrow$E (3)(4)
(6)	$Member(v,u.x)$	$\wedge$E (1)
(7)	$Member(v,u.x) \leftrightarrow v = u \vee Member(v,x)$	$\forall$E lemma
(8)	$Member(v,u.x) \rightarrow v = u \vee Member(v,x)$	$\leftrightarrow$E (7)
(9)	$v = u \vee Member(v,x)$	$\rightarrow$E (6)(8)

We want to derive the formula $Member(v,z1)$. We can arrive at it by $\vee$ - elimination if we can derive it from $v = u$ and from $Member(v,x)$. To derive $Member(v,z1)$, we must use the definiens in the program clause or step (5) that we have derived. It is a disjunction so we get a $\vee$ - elimination, on one hand, for the case $Member(u,y)$ and $z1 = u.z$, on the other, for the case $\neg Member(u,y)$ and $z1 = z$.

(10)	$v = u$	hyp
(11)	$Member(u,y) \wedge z1 = u.z \wedge Intersection(x,y,z)$	hyp
(12)	$z1 = u.z$	$\wedge$E (11)
(13)	$z1 = v.z$	id (10)(12)
(14)	$Member(v,v.z)$	$\forall$E lemma
(15)	$Member(v,z1)$	id (12)(13)
(16)	$Member(u,y) \wedge z1 = u.z \wedge Intersection(x,y,z) \rightarrow$ $Member(v,z1)$	$\rightarrow$I (11)(15)
(17)	$\neg Member(u,y) \wedge z1 = z \wedge Intersection(x,y,z)$	hyp
(18)	$\neg Member(u,y)$	$\wedge$E (17)
(19)	$Member(v,y)$	$\wedge$E (1)
(20)	$Member(u,y)$	id (10)(19)
(21)	$\bot$	$\bot_{int}$
(22)	$Member(v,z1)$	$\neg$E
(23)	$\neg Member(u,y) \wedge z1 = z \wedge Intersection(x,y,z) \rightarrow$ $Member(v,z1)$	$\rightarrow$I (17)(23)
(24)	$Member(v,z1)$	$\vee$E (5)(16)(23)
(25)	$v = u \rightarrow Member(v,z1)$	$\rightarrow$I (10)(24)

We continue with the other derivation which also contains a $\vee$-elimination. We first assume $Member(u,y)$ and $z1 = u.z$.

(26)	$Member(v,x)$	hyp
(27)	$Member(u,y) \wedge z1 = u.z \wedge Intersection(x,y,z)$	hyp
(28)	$Intersection(x,y,z)$	$\wedge$E (27)
(29)	$Intersection(x,y,z) \rightarrow$ $\forall v(Member(v,x) \wedge Member(v,y) \leftrightarrow Member(v,z))$	$\wedge$E (1)
(30)	$\forall v(Member(v,x) \wedge Member(v,y) \leftrightarrow Member(v,z))$	$\rightarrow$E (28)(29)
(31)	$Member(v,x) \wedge Member(v,y) \leftrightarrow Member(v,z)$	$\forall$E (30)
(32)	$Member(v,y)$	$\wedge$E (1)
(33)	$Member(v,x) \wedge Member(v,y)$	$\wedge$I (26)(32)

(34)	$Member(v,x) \wedge Member(v,y) \rightarrow Member(v,z)$	$\leftrightarrow$E (31)
(35)	$Member(v,z)$	$\rightarrow$E (33)(34)
(36)	$z1 = u.z$	$\wedge$E (27)
(37)	$Member(v,u.z) \leftrightarrow v = u \vee Member(v,z)$	$\forall$E lemma
(39)	$v = u \vee Member(v,z) \rightarrow Member(v,u.z)$	$\leftrightarrow$E (37)
(40)	$v = u \vee Member(v,z)$	$\vee$I (26)
(41)	$Member(v,u.z)$	$\rightarrow$E (39)(40)
(42)	$Member(v,z1)$	id (36)(41)
(43)	$Member(u,y) \wedge z1 = u.z \wedge Intersection(x,y,z) \rightarrow Member(v,z1)$	$\rightarrow$I (27)(42)

We then assume $Member(u,y)$ and $z1 = z$.

(44)	$\neg Member(u,y) \wedge z1 = z \wedge Intersection(x,y,z)$	hyp
(45)	$Intersection(x,y,z)$	$\wedge$E (44)
(46)	$Intersection(x,y,z) \rightarrow \forall v(Member(v,x) \wedge Member(v,y) \leftrightarrow Member(v,z))$	$\wedge$E (1)
(47)	$\forall v(Member(v,x) \wedge Member(v,y) \leftrightarrow Member(v,z))$	$\rightarrow$E (45)(46)
(48)	$Member(v,x) \wedge Member(v,y) \leftrightarrow Member(v,z)$	$\forall$E (47)
(49)	$Member(v,x) \wedge Member(v,y) \rightarrow Member(v,z)$	$\leftrightarrow$E (48)
(50)	$Member(v,y)$	$\wedge$E (1)
(51)	$Member(v,x) \wedge Member(v,y)$	$\wedge$I (26)(50)
(52)	$Member(v,z)$	$\rightarrow$E (49)(51)
(53)	$z1 = z$	$\wedge$E (44)
(53)	$Member(v,z1)$	id (52)(53)
(54)	$\neg Member(u,y) \wedge z1 = z \wedge Intersection(x,y,z) \rightarrow Member(v,z1)$	$\rightarrow$I (44)(53)
(55)	$Member(v,z1)$	$\vee$E (5)(43)(54)
(56)	$Member(v,x) \rightarrow Member(v,z1)$	$\rightarrow$I (26)(55)

We have the steps we need to make a $\vee$-elimination using step (9). And finally, there remains only introduction of implication and universal quantifiers.

(57)	$Member(v,z1)$	$\vee$E (9)(25)(56)
(58)	$\forall y \forall z(Intersection(x,y,z) \rightarrow \forall v(Member(v,x) \wedge Member(v,y) \leftrightarrow Member(v,z))) \wedge (Member(v,u.x) \wedge Member(v,y) \wedge Intersection(u.x,y,z1)) \rightarrow Member(v,z1)$	$\rightarrow$I (1)(57)
(59)	$\forall x \forall y \forall z1 \forall v \forall u \forall y \forall z(Intersection(x,y,z) \rightarrow \forall v(Member(v,x) \wedge Member(v,y) \leftrightarrow Member(v,z))) \wedge (Member(v,u.x) \wedge Member(v,y) \wedge Intersection(u.x,y,z1)) \rightarrow Member(v,z1)$	$\forall$I (58)

We omit the derivation of Lemma 2 which is carried out in the same way.

5.2.4 Other Properties

We have shown the properties completeness, termination, and correctness for some programs. Besides these properties which are fundamental for programs

we may also be interested in other properties. Examples of other properties are that a program is equivalent to another or that it is transitive, associative, or commutative.

Except when we show completeness and termination for programs, the theorems have the form $P_{prog} \rightarrow property$. Thus, we want to show that, given a program, a particular property holds. To be able to make this kind of reasoning we need the "only if" part of the definition. This means that we may be forced to reformulate the program from an "if" expression to an "if and only if" expression when we shall perform the proof.

We can show, for instance, that the program *Subset* is transitive. The theorem expresses the fact that for arbitrary x, y, and z, if x is a subset of y and y is a subset of z, then x is a subset of z.

$$\forall x \forall y \forall z \, (Subset(x, y) \wedge Subset(y, z) \rightarrow Subset(x, z))$$

To perform the proof we need the program *Subset* in the completed form.

$$\begin{aligned} &\forall x \forall y (Subset(x, y) \leftrightarrow \\ &\quad x = \emptyset \\ &\quad \vee \exists u \exists v (x = u.v \wedge Subset(v, y) \wedge Member(u, y))) \end{aligned}$$

In Round 3 we defined the relation *Append* between three lists, where the third list comprises the list one gets when the second list is appended to the first list.

$$\begin{aligned} &\forall x \forall y \forall z \, (Append(x, y, z) \leftrightarrow \\ &\quad x = \emptyset \wedge y = z \\ &\quad \vee \exists u \exists x1 \exists z1 (x = u.x1 \wedge z = u.z1 \wedge Append(x1, y, z1))) \end{aligned}$$

When we append three lists, x, y, and z, in a certain order we can set about it in two ways. We may begin by appending y, to x and then append z at the rear. Or, alternatively, we may append z appended to y to x. Regardless of the way we select the result will be the same and we say that the relation *Append* is associative.

The theorem for the associativity of *Append* is formulated thus: given three lists x, y, and z, we can show that $Append(x, y, u)$ and $Append(u, z, w)$ are equivalent to $Append(y, z, v)$ and $Append(x, v, w)$.

$$\begin{aligned} \forall x \forall y \forall u \forall z \forall v \forall w \, &(Append(x, y, u) \wedge Append(u, z, w) \leftrightarrow \\ &Append(y, z, v) \wedge Append(x, v, w)) \end{aligned}$$

The proof that the theorem follows from the program is performed using induction.

5.3 Program Transformation

We have previously derived programs from specificationsz expressed in a more abstract form than the programs themselves. A derivation of a program from

a specification that is not abstract, i.e. a program, is called *program transformation.*

When we derive programs and formulate definitions of programs directly, certain forms of definitions may be preferable and natural. In program derivations performed as in Sect. 5.1 above, we arrive at recursive expressions. Also in formulations of the programs directly in Horn form, we use the recursive form. However, we can be interested in other forms of definitions for efficient computation, such as iterative forms. Furthermore, one data structure may be preferable to another in a derivation or in a formulation of a definition while another may be preferable for computations. Thus, it is not a matter of course that the same form is the best both for expressing programs and for drawing conclusions; it may also be important to obtain programs using different data structures but computing values according to the same abstract specification. When we have a program in one form, we can transform the program into another form, instead of deriving or defining a new program.

If we are able to carry out the transformations in a formal way, we may be sure that the derived program will express the same thing as the original program. We can carry out formal transformations in the programming calculus and we call this type of derivation Π_{iii}.

$$\frac{(D) \quad (P_{spec}) \quad (P_{prog})}{P'_{prog}} \; \Pi_{iii}$$

We will study an example of a program in recursive form which we transform into iterative form. When speaking of recursive programs we have meant self-referring programs. In this case we take recursive programs to mean such programs as make use of new memory space for every recursion. When the evaluation has reached the bottom of the recursion it successively makes memory space available. A self-referring program that uses the same memory space at every iteration we call *iterative.*

To express a definition recursively is more natural than to express it iteratively. But an iterative form may often be preferable for computation since it consumes less memory space.

We defined the relation *Reverse* recursively in Round 3. It holds for two lists when one list contains the same elements as the other, but in reversed order.

$$Reverse(\emptyset, \emptyset)$$
$$\forall u \forall x \forall y \forall y1 \, (Reverse(u.x, y) \leftarrow Reverse(x, y1) \wedge Append(y1, u.\emptyset, y))$$

The program contains the *Append* relation which we have met before in Round 3 as well as in the previous section.

$$\forall x \; Append(\emptyset, x, x)$$
$$\forall u \forall x \forall y \forall z \, (Append(u.x, y, u.z) \leftarrow Append(x, y, z))$$

(1) $Reverse(\emptyset, \emptyset)$	def
(2) $Append(\emptyset, y, y)$	$\forall$E def
(3) $Reverse(\emptyset, \emptyset) \wedge Append(\emptyset, y, y)$	$\wedge$I (1)(2)
(4) $Reverse_{it}(\emptyset, y, y) \leftrightarrow Reverse(\emptyset, \emptyset) \wedge Append(\emptyset, y, y)$	$\forall$E def
(5) $Reverse(\emptyset, \emptyset) \wedge Append(\emptyset, y, y) \rightarrow Reverse_{it}(\emptyset, y, y)$	$\leftrightarrow$E (4)
(6) $Reverse_{it}(\emptyset, y, y)$	$\rightarrow$E (3)(5)

Figure 5.7: Derivation of the base case of the iterative version of the relation reverse.

To transform a program P to a program P' in another form, we have to define the relation between P and P'. We name the relation, which will have an iterative formulation, $Reverse_{it}$ and we express how it relates to $Reverse$:

$$\forall x \forall y \forall z \forall x1 \, (Reverse_{it}(x, y, z) \leftrightarrow Reverse(x, x1) \wedge Append(x1, y, z))$$

We carry out the transformation as an induction proof over lists, in this case, and begin with the base case where we proceed from the program clause for the empty list for $Reverse$, see Figure 5.7.

We continue with the derivation of the recursive case. To shorten the derivation we use the following lemma about $Append$.

$$\forall u \forall x \; Append(u.\emptyset, x, u.x)$$

In Sect. 5.2.4 we expressed a theorem for the property associativity for the relation $Append$. We will use this theorem in the derivation of the recursive case for $Reverse_{it}$. The derivation is shown in Figure 5.8.

$$\forall x \forall y \forall u \forall z \forall v \forall w \, (Append(x, y, u) \wedge Append(u, z, w) \leftrightarrow Append(y, z, v) \wedge Append(x, v, w))$$

The program we get after transformation is

$$\forall y \; Reverse_{it}(\emptyset, y, y)$$
$$\forall u \forall x \forall y \forall z \, (Reverse_{it}(u.x, y, z) \leftarrow Reverse_{it}(x, u.y, z))$$

As we can see, the iterative version is also one of the programs expressed using difference lists in Round 3. We wanted a formulation that yielded the same result but with fewer computation steps by use of matching. We also notice that it utilizes less memory space.

5.4 An Example

We will carry out a derivation of a program to check that a list is ordered. We can specify the program in this way: a list w is ordered if, and only if, it is

(1)	$Reverse_{it}(x, u.y, z)$	def
(2)	$Reverse_{it}(x, u.y, z) \leftrightarrow Reverse(x, x1) \wedge Append(x1, u.y, z)$	$\forall$E def
(3)	$Reverse_{it}(x, u.y, z) \rightarrow Reverse(x, x1) \wedge Append(x1, u.y, z)$	$\leftrightarrow$E def
(4)	$Reverse(x, x1) \wedge Append(x1, u.y, z)$	$\rightarrow$E (1)(3)
(5)	$Reverse(x, x1)$	$\wedge$E (4)
(6)	$Append(x1, u.y, z)$	$\wedge$E (4)
(7)	$Append(u.\emptyset, y, u.y)$	$\forall$E lemma
(8)	$Append(u.\emptyset, y, u.y) \wedge Append(x1, u.y, z)$	$\wedge$I (6)(7)
(9)	$Append(x1, u.\emptyset, x2) \wedge Append(x2, y, z) \leftrightarrow$ $Append(u.\emptyset, y, u.y) \wedge Append(x1, u.y, z)$	$\forall$E teorem assoc
(10)	$Append(u.\emptyset, y, u.y) \wedge Append(x1, u.y, z) \rightarrow$ $Append(x1, u.\emptyset, x2) \wedge Append(x2, y, z)$	$\leftrightarrow$E (9)
(11)	$Append(x1, u.\emptyset, x2) \wedge Append(x2, y, z)$	$\rightarrow$E (8)(10)
(12)	$Append(x1, u.\emptyset, x2)$	$\wedge$E (11)
(13)	$Reverse(x, x1) \wedge Append(x1, u.\emptyset, x2)$	$\wedge$I (5)(12)
(14)	$Reverse(u.x, x2)$	$\leftarrow$E def (13)
(15)	$Append(x2, y, z)$	$\wedge$E (11)
(16)	$Reverse(u.x, x2) \wedge Append(x2, y, z)$	$\wedge$I (14)(15)
(17)	$Reverse_{it}(u.x, y, z)$	$\rightarrow$E def (16)
(18)	$Reverse_{it}(u.x, y, z) \leftarrow Reverse_{it}(x, u.y, z)$	$\leftarrow$I (1)(17)

Figure 5.8: Derivation of the recursive case for the iterative version of the reverse relation.

empty or if there exist x and y such that $w = x.y$, and all elements in the list y are greater than x and the list y is ordered.

$$\forall w(Ordered(w) \leftrightarrow \quad (s4)$$
$$w = \emptyset \vee \exists x \exists y(w = x.y \wedge \forall v(Member(v, y) \rightarrow x < v) \wedge Ordered(y)))$$

The property shall hold for all lists. The program should be defined inductively over the list structure.

$$Ordered(\emptyset) \leftarrow \cdots$$
$$\forall x \forall y(Ordered(x.y) \leftarrow Ordered(y) \wedge \cdots)$$

We can carry out the derivation of the base case directly, see Figure 5.9. The base case for the empty list is without conditions. Let us continue and look at the case of the constructed list. We assume that the final derivation steps will appear like this:

$$Ordered(e.l) \leftarrow Ordered(l) \wedge \cdots$$
$$\forall x \forall y(Ordered(x.y) \leftarrow Ordered(y) \wedge \cdots)$$

The given condition is $Ordered(l)$. We instantiate the specification first with the list $e.l$ and then l.

(1) $Ordered(\emptyset) \leftarrow$ $\forall$E ,$\leftrightarrow$E ($s4$)
$\emptyset = \emptyset \vee \exists x \exists y(\emptyset = x.y \wedge \forall v(Member(v,y) \rightarrow x < v) \wedge Ordered(y))$
(2) $\emptyset = \emptyset$ $\forall$E ($d1$)
(3) $\emptyset = \emptyset$ $\vee$I (2)
$\vee \exists x \exists y(\emptyset = x.y \wedge \forall v(Member(v,y) \rightarrow x < v) \wedge Ordered(y))$
(4) $Ordered(\emptyset)$ $\leftarrow$E (1)(3)

Figure 5.9: Derivation of the base case for the relation ordered.

(1) $Ordered(e.l) \leftarrow$ $\forall$E ,$\leftrightarrow$E ($s4$)
$e.l = \emptyset$
$\vee \exists x \exists y\,(e.l = x.y \wedge \forall v\,(Member(v,y) \rightarrow x < v) \wedge Ordered(y))$
(2) $Ordered(l)$ hyp
(3) $Ordered(l) \rightarrow$ $\forall$E ,$\leftrightarrow$E ($s4$)
$l = \emptyset$
$\vee \exists x \exists y\,(l = x.y \wedge \forall v\,(Member(v,y) \rightarrow x < v) \wedge Ordered(y))$
(4) $l = \emptyset$ $\rightarrow$E (2)(3)
$\vee \exists x \exists y\,(l = x.y \wedge \forall v\,(Member(v,y) \rightarrow x < v) \wedge Ordered(y))$
$\cdots$
(n-3) $e.l = \emptyset \vee$
$\exists x \exists y\,(e.l = x.y \wedge \forall v\,(Member(v,y) \rightarrow x < v) \wedge Ordered(y))$
(n-2) $Ordered(e.l)$ $\leftarrow$E (1)(n-3)
(n-1) $Ordered(e.l) \leftarrow Ordered(l) \wedge \cdots$ $\leftarrow$I (n-2)
(n) $\forall x \forall y(Ordered(x.y) \leftarrow Ordered(y) \wedge \cdots)$ $\forall$I (n-1)

From step (4) we thus want to derive step (n-3). This cannot be done generally, however. We have to consider two cases. The first is when l is an empty list and the second is when l is non-empty.

Let us first add the condition that $l = \emptyset$. We know that l is ordered, $Ordered(l)$. If l is empty we can derive $\forall v\,(Member(v,l) \rightarrow e < v)$ by contradiction and by the identity axiom ($d1$) we get $e.l = e.l$, i.e we have

$$e.l = e.l \wedge \forall v\,(Member(v,l) \rightarrow e < v) \wedge Ordered(l)$$

This can be existentially quantified to

$$\exists x \exists y\,(e.l = x.y \wedge \forall v\,(Member(v,y) \rightarrow x < v) \wedge Ordered(y))$$

$\vee$-introduction finally yields

$$e.l = \emptyset \vee \exists x \exists y\,(e.l = x.y \wedge \forall v\,(Member(v,y) \rightarrow x < v) \wedge Ordered(y))$$

With two expressions as hypotheses, $Ordered(l)$ and $l = \emptyset$, we have derived the formula $Ordered(e.l)$. The entire derivation of the second program clause

(1) $Ordered(e.l) \leftarrow$ $\forall E, \leftrightarrow E$ $(s4)$
$e.l = \emptyset$
$\vee\ \exists x \exists y\,(e.l = x.y \wedge \forall v\,(Member(v,y) \rightarrow x < v) \wedge Ordered(y))$

(2) $Ordered(l)$ hyp

(3) $Ordered(l) \rightarrow$ $\forall E, \leftrightarrow E$ $(s4)$
$l = \emptyset$
$\vee\ \exists x \exists y\,(l = x.y \wedge \forall v\,(Member(v,y) \rightarrow x < v) \wedge Ordered(y))$

(4) $l = \emptyset$ $\rightarrow E$ (2)(3)
$\vee\ \exists x \exists y\,(l = x.y \wedge \forall v\,(Member(v,y) \rightarrow x < v) \wedge Ordered(y))$

(5) $l = \emptyset$ hyp

(6) $l = \emptyset \wedge Ordered(l) \rightarrow$ $\forall E$ Lemma
$e.l = \emptyset$
$\vee\ \exists x \exists y\,(e.l = x.y \wedge \forall v\,(Member(v,y) \rightarrow x < v) \wedge Ordered(y))$

(7) $l = \emptyset \wedge Ordered(l)$ $\wedge I$ (2)(5)

(8) $e.l = \emptyset$ $\rightarrow E$ (6)(7)
$\vee\ \exists x \exists y\,(e.l = x.y \wedge \forall v\,(Member(v,y) \rightarrow x < v) \wedge Ordered(y))$

(9) $Ordered(e.l)$ $\leftarrow E$ (1)(8)

(10) $Ordered(e.l) \leftarrow Ordered(l) \wedge l = \emptyset$ $\leftarrow I$ (2)(5)(9)

(11) $\forall x \forall y(Ordered(x.y) \leftarrow Ordered(y) \wedge y = \emptyset)$ $\forall I$ (10)

Figure 5.10: The derivation of the second program clause for the relation ordered.

is shown in Figure 5.10. The derivation of the lemma used above is shown in Figure 5.11.

Let us see if we can get one more recursive case. We assume that l is constructed, i.e. $l = f.fl$. We reason from the disjunction in step (6) in the derivation of the second program clause. $l = \emptyset$ yields a contradiction and we can obtain the wanted form directly.

$$\exists x \exists y\,(l = x.y \wedge \forall v\,(Member(v,y) \rightarrow x < v) \wedge Ordered(y))$$
$$l = d.dl \wedge \forall v\,(Member(v,dl) \rightarrow d < v) \wedge Ordered(dl)$$

Since $l = f.fl$ and $l = d.dl$, it follows that $f.fl = d.dl$ and according to the axiom for list identity $f = d$ and $fl = dl$, i.e.

$$\forall v\,(Member(v,fl) \rightarrow f < v) \wedge Ordered(fl)$$

We know from before that $l = f.fl$ is ordered. $\forall v\,(Member(v,f.fl) \rightarrow e < v)$ can be derived if we add the condition $e < f$.

$$e.f.fl = e.f.fl \wedge \forall v\,(Member(v,f.fl) \rightarrow e < v) \wedge Ordered(f.fl)$$

(1)	$l = \emptyset \wedge \mathit{Ordered}(l)$	hyp
(2)	$\mathit{Member}(c, l)$	hyp
(3)	$\mathit{Member}(c, \emptyset)$	id (1)(2)
(4)	$\neg\, \mathit{Member}(c, \emptyset)$	$\forall$E ($d9$)
(5)	$\bot$	$\neg$E (3)(4)
(6)	$e < c$	$\bot_{int}$ (5)
(7)	$\mathit{Member}(c, l) \rightarrow e < c$	$\rightarrow$I (2)(6)
(8)	$\forall v\, (\mathit{Member}(v, l) \rightarrow e < v)$	$\forall$I (7)
(9)	$e.l = e.l$	$\forall$E ($d1$)
(10)	$\mathit{Ordered}(l)$	$\wedge$E (1)
(11)	$e.l = e.l \wedge \forall v\, (\mathit{Member}(v, l) \rightarrow e < v) \wedge \mathit{Ordered}(l)$	$\wedge$I (9)(8)(10)
(12)	$\exists x \exists y\, (e.l = x.y \wedge \forall v(\mathit{Member}(v, y) \rightarrow x < v) \wedge \mathit{Ordered}(y))$	$\exists$I (11)
(13)	$e.l = \emptyset \vee \exists x \exists y\, (e.l = x.y \wedge \forall v(\mathit{Member}(v, y) \rightarrow x < v) \wedge \mathit{Ordered}(y))$	$\vee$I (12)
(14)	$l = \emptyset \wedge \mathit{Ordered}(l) \rightarrow e.l = \emptyset \vee \exists x \exists y\, (e.l = x.y \wedge \forall v(\mathit{Member}(v, y) \rightarrow x < v) \wedge \mathit{Ordered}(y))$	$\rightarrow$I (1)(13)
(15)	$\forall z \forall w\, (w = \emptyset \wedge \mathit{Ordered}(w) \rightarrow z.w = \emptyset \vee \exists x \exists y\, (z.w = x.y \wedge \forall v(\mathit{Member}(v, y) \rightarrow x < v) \wedge \mathit{Ordered}(y)))$	$\forall$I (14)

Figure 5.11: The derivation of the lemma used in the proof in Figure 5.10.

We can introduce quantifiers and complete the formula by $\vee$-introduction. Thus we have derived $\mathit{Ordered}(e.l)$ under the conditions $\mathit{Ordered}(l)$, $l = f.\mathit{fl}$ and $e < f$. The derivation of the third program clause is shown in Figure 5.12. The derivation of the lemma is omitted.

The formulas we have derived are:

$$\mathit{Ordered}(\emptyset) \qquad (p1)$$

$$\forall x \forall y\, (\mathit{Ordered}(x.y) \leftarrow \mathit{Ordered}(y) \wedge y = \emptyset) \qquad (p2)$$

$$\forall z \forall v \forall x \forall y\, (\mathit{Ordered}(x.y) \leftarrow \mathit{Ordered}(y) \wedge y = z.v \wedge x < z) \qquad (p3)$$

The program has the following appearance after derivation:

$$\begin{array}{l} \mathsf{Ordered}(\emptyset) \leftarrow \\ \mathsf{Ordered(x.y)} \leftarrow \mathsf{Ordered(y)},\ \mathsf{y} = \emptyset \\ \mathsf{Ordered(x.y)} \leftarrow \mathsf{Ordered(y)},\ \mathsf{y} = \mathsf{z.w},\ \mathsf{x} < \mathsf{z} \end{array}$$

From the above a new definition of Ordered can be derived in which the appearance of the list is carried over to the consequence of the clauses. We eliminate the conditions in the second program clause as this derivation demonstrates. In the third program clause we can transmit the condition $y = z.w$ to

(1) $Ordered(e.l) \leftarrow$ $\forall$E ,$\leftrightarrow$E ($s4$)
$e.l = \emptyset$
$\vee\ \exists x \exists y\,(e.l = x.y \wedge \forall v\,(Member(v,y) \rightarrow x < v) \wedge Ordered(y))$

(2) $Ordered(l)$ hyp

(3) $Ordered(l) \rightarrow$ $\forall$E , $\leftrightarrow$E ($s4$)
$l = \emptyset$
$\vee\ \exists x \exists y\,(l = x.y \wedge \forall v\,(Member(v,y) \rightarrow x < v) \wedge Ordered(y))$

(4) $l = \emptyset$ $\rightarrow$E (2)(4)
$\vee\ \exists x \exists y\,(l = x.y \wedge \forall v\,(Member(v,y) \rightarrow x < v) \wedge Ordered(y))$

(5) $l = f.fl \wedge e < f$ hyp

(6) $l = \emptyset$ hyp

(7) $f.fl = \emptyset$ id (5)(6)

(8) $\neg f.fl = \emptyset$ $\forall$E ($d6$)

(9) $\perp$ $\neg$E (7)(8)

(10) $e.l = \emptyset \vee$ $\perp_{int}$ (9)
$\exists x \exists y\,(e.l = x.y \wedge \forall v(Member(v,y) \rightarrow x < v) \wedge Ordered(y))$

(11) $l = \emptyset \rightarrow$ $\rightarrow$I (6)(10)
$e.l = \emptyset$
$\vee\ \exists x \exists y\,(e.l = x.y \wedge \forall v(Member(v,y) \rightarrow x < v) \wedge Ordered(y))$

(12) $\exists x \exists y\,(l = x.y \wedge \forall v(Member(v,y) \rightarrow x < v) \wedge Ordered(y))$ hyp

(13) $l = f.fl \wedge e < f \wedge Ordered(l)$ $\wedge$I (5)(2)(12)
$\wedge\ \exists x \exists y\,(l = x.y \wedge \forall v(Member(v,y) \rightarrow x < v) \wedge Ordered(y))$

(14) $l = f.fl \wedge e < f \wedge Ordered(l)$ $\forall$E Lemma
$\wedge\ \exists x \exists y\,(l = x.y \wedge \forall v(Member(v,y) \rightarrow x < v) \wedge Ordered(y)) \rightarrow$
$e.l = \emptyset$
$\vee\ \exists x \exists y\,(e.l = x.y \wedge \forall v(Member(v,y) \rightarrow x < v) \wedge Ordered(y))$

(15) $e.l = \emptyset$ $\rightarrow$E (13)(14)
$\vee\ \exists x \exists y\,(e.l = x.y \wedge \forall v(Member(v,y) \rightarrow x < v) \wedge Ordered(y))$

(16) $\exists x \exists y\,(l = x.y \wedge \forall v(Member(v,y) \rightarrow x < v) \wedge$ $\rightarrow$I (12)(15)
$Ordered(y)) \rightarrow$
$e.l = \emptyset \vee$
$\exists x \exists y\,(e.l = x.y \wedge \forall v\,(Member(v,y) \rightarrow x < v) \wedge Ordered(y))$

(17) $e.l = \emptyset \vee$ $\vee$E (4)(11)(16)
$\exists x \exists y\,(e.l = x.y \wedge \forall v\,(Member(v,y) \rightarrow x < v) \wedge Ordered(y))$

(18) $Ordered(e.l)$ $\leftarrow$E (2)(17)

(19) $Ordered(e.l) \leftarrow Ordered(l) \wedge l = f.fl \wedge e < f$ $\leftarrow$I (2)(5)(18)

(20) $\forall z \forall v \forall x \forall y\,(Ordered(x.y) \leftarrow Ordered(y) \wedge y = z.v \wedge x < z)$ $\forall$I (19)

Figure 5.12: The derivation of the third clause for the relation ordered.

the arguments of the consequence through this derivation. The derivations are shown in Figure 5.13.

(1) $Ordered(\emptyset)$	prog
(2) $\emptyset = \emptyset$	$\forall$E $(d1)$
(3) $\emptyset = \emptyset \wedge Ordered(\emptyset)$	$\wedge$I (2)(1)
(4) $Ordered(e.\emptyset) \leftarrow \emptyset = \emptyset \wedge Ordered(\emptyset)$	$\forall$E $(p2)$
(5) $Ordered(e.\emptyset)$	$\leftarrow$E (3)(4)
(6) $\forall x\ Ordered(x.\emptyset)$	$\forall$I (5)

(1) $Ordered(f.fl) \wedge e < f$	hyp
(2) $f.fl = f.fl$	$\forall$E $(d1)$
(3) $Ordered(f.fl) \wedge f.fl = f.fl \wedge e < f$	$\wedge$I (1)(2)
(4) $Ordered(e.f.fl) \leftarrow Ordered(f.fl) \wedge f.fl = f.fl \wedge e < f$	$\forall$E $(p3)$
(5) $Ordered(e.f.fl)$	$\leftarrow$E (3)(4)
(6) $Ordered(e.f.fl) \leftarrow Ordered(f.fl) \wedge e < f$	$\leftarrow$I (1)(5)
(7) $\forall x \forall z \forall w(Ordered(x.z.w) \leftarrow Ordered(z.w) \wedge x < z)$	$\forall$I (6)

Figure 5.13: Derivations moving identity-conditions to the consequence of a clause.

Our new program has this appearance

Ordered(∅) ←
Ordered(x.∅) ←
Ordered(x.z.w) ← Ordered(z.w), x < z

5.5 Exercises

Exercise 1:
Specify a relation $Computing\_science\_town(x)$ which is valid if, and only if, x is either Uppsala or Linköping, the Swedish cities that have a computing science program at their universities.

Exercise 2:
Derive a program from the specification of $Computing\_science\_town$.

Exercise 3:
Specify a relation $Max(x, m)$ where x is a structure and m is the greatest element in the structure.

Exercise 4:
Derive a program from the specification in the previous exercise when the structure is represented by a list.

Consider in particular what lists the relation shall hold for. What axioms are necessary?

Exercise 5:

Specify a relation $Maxnumber(x, y, z)$ such that z is the greater of the numbers x and y.

Exercise 6:

Derive a program for the relation *Max* in exercise 3 where *Maxnumber* is used as an auxiliary program.

Exercise 7:

We have expressed a specification for *Paternal_grandparent* in this round and derived the program. Express the theorem for completeness for this program.

Exercise 8:

A specification and program for the relation *Intersection* between three sets is to be found in Sect. 5.2.3.

(a) Express the theorem for termination of the program *Intersection*.

(b) Make an outline of the proof.

(c) Is the program complete as regards the specification? Explain the answer.

Exercise 9:

Express the theorem for partial correctness for the program Insert that is to be found in Sect. 5.2.2.

Exercise 10:

The program Append can be found Sect. 5.2.4, and elsewhere.

(a) Express the theorem that establishes that the program Append has the property that, if we append an empty list to the rear of an arbitrary list x, the composite list will be the list x.

(b) Make an outline of the proof of the theorem.

Round 6. Efficient Computation

6.1 Search Space Reduction

6.1.1 The Cut Construction

A definition may be expressed so that there are alternative evaluation possibilities reached by backtracking. We can sometimes conclude in advance that an alternative evaluation will not succeed. If we can *control* the search so that these fruitless alternatives are never tried, we will achieve a more efficient evaluation.

Let us study an example. The intersection between two lists list1 and list2 consists of a list with only such elements as are to be found in both list1 and list2. We define the relation recursively over the first list. If the list in the first argument is empty, the list in the third argument, which constitutes the intersection, shall also be empty. If the list in the first argument is non-empty and the first element belongs to the second list as well, it shall be included in the list that constitutes the intersection, otherwise not.

```
Intersection(∅,list,∅) ←                                          (155)
Intersection(elem.list1,list2,elem.list3) ← Member(elem,list2),   (156)
        Intersection(list1,list2,list3)
Intersection(elem.list1,list2,list3) ← Not Member(elem,list2),    (157)
        Intersection(list1,list2,list3)
```

Prolog's search order among the clauses entails that (156) will be tried before (157) for a question about Intersection. We regard (157) as an alternative case of (156) that will be tried on backtracking. Since Member(u,y) and Not Member(u,y) are not true at the same time for the same instantiations of the variables, we know that (157) is not a real alternative if Member(elem,list2) in (156) is true. There are thus branches in the proof tree which we are aware will not lead to any new proof. We want to be able to control the search so that these branches are not tried.

We use a *control language* to add information for a more efficient evaluation. In Prolog we can *cut* away those branches in the proof tree that we know will not be needed by using the control construction cut, "|". The symbol "|" is

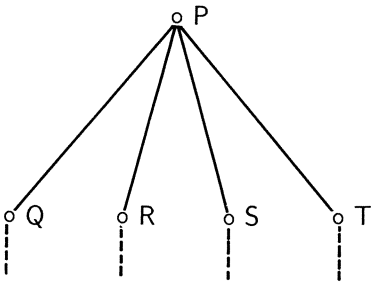

Figure 6.1: An AND-OR-tree for the clauses (158) and (159).

placed in the conditional parts of the definition of a predicate whose alternative evaluations will be limited. It is placed first in a conditional part if matching in the corresponding consequence part determines that no alternatives should be searched. If any of the conditions in the conditional part determine the limitation, "|" comes after that condition.

Suppose that we have the following two clauses for P; the AND-OR-tree is shown in Figure 6.1.

$$P \leftarrow Q, R \tag{158}$$
$$P \leftarrow S, T \tag{159}$$

If we know that it is not meaningful to try to prove S and T if Q is true, we can add "|" after Q in (158).

$$P \leftarrow Q|, R \tag{160}$$
$$P \leftarrow S, T \tag{161}$$

When Q succeeds during the execution of a question P, the right-hand OR-branch in the AND-OR-tree is cut off. If R fails or backtracking occurs for some other reason, no alternative evaluation of P will occur and the question P will thus not be considered true.

If Q does not succeed, (161) remains as an alternative for evaluating P. It is when "|" has been passed in the evaluation that possible alternatives are removed.

Up to this chapter all the building blocks of clauses have been related only to other building blocks in the same clause and not to anything outside. In this respect cut is different. Cut is a control construction related to the clause in which it is placed and also to subsequent clauses in the same definition. In (160) and (161) cut is related to (160) and also to (161). Cut is related to the clause in which it is placed because its position is important. Furthermore, it is related to the entire definition because it can exclude clauses that occur later in the definition from being tried.

In the conditional part cut works as a partition. The evaluation of the conditions placed before "|" differs from the evaluation of those conditions coming

after it. On backtracking, alternative evaluations are searched for the conditions placed after "|", but not for those that occur before. In (160) and (161) alternative evaluations for R can thus be searched, but not for Q, and, as we are already aware, not for P either, once an evaluation of Q has succeeded.

In our example Intersection in (155) – (157) above, we know that no alternatives should be searched if Member(elem,list2) in (156) succeeds. We can thus place an "|" after Member(elem,list2) in (156) and express the relation as follows.

```
Intersection(∅,list,∅) ←                                              (162)
Intersection(elem.list1,list2,elem.list3) ← Member(elem,list2)|,      (163)
        Intersection(list1,list2,list3)
Intersection(elem.list1,list2,list3) ← Not Member(elem,list2),        (164)
        Intersection(list1,list2,list3)
```

The first clause (162) becomes identical to (155). The second is logically identical to (156) but it contains a cut and changes the evaluation of the program. The third clause (164) is identical to (157). This means that the program (162) – (164) is logically indentical to (155) – (157) and computes the same values but the computation from the clauses (162) – (164) will be more efficient because alternatives that will prove of no avail will never be tried.

In the third clause the condition Not Member expresses the fact that an element not included in the second list should not be in the third list either. The search order of Prolog does not bring up that clause until the others have failed. In (157) we have included the condition Not Member, not only to make the clause logically correct, but also to avoid having Intersection incorrectly evaluated when backtracking occurs. When we use "|" after Member there is no possibility of incorrect answers, as long as we keep the order between the clauses, and we can omit the condition Not Member.

```
Intersection(∅,list,∅) ←                                              (165)
Intersection(elem.list1,list2,elem.list3) ← Member(elem,list2)|,      (166)
        Intersection(list1,list2,list3)
Intersection(elem.list1,list2,list3) ←                                (167)
        Intersection(list1,list2,list3)
```

The first clause (165) becomes identical to (155) and the second (166) is logically identical to (156) but the third clause (167) is not logically indentical to (157). The program (165) – (167) will yield the same result as (155) – (157) despite the fact that the clauses are not logically identical. The interpretation of a single clause will be dependent on the entire program. In this program however, besides avoiding fruitless alternatives, we avoid carrying out the computation that an element is not a member of a list after failing to compute that it is a member of that list. If the program is used for lists with many elements the program (165) – (167) may be used. This use of cut is otherwise not preferable.

Since we can ask different types of questions about a relation by varying what arguments we instantiate, we must be careful when using "|". That the clauses express exclusive cases as regards some arguments, does not necessarily mean that they do for others.

We recall from Sect. 3.2 the program Append which expresses a relation between three lists in which the third consists of the first two lists joined together.

```
Append(∅,list,list) ←                                    (168)
Append(elem.list1,list2,elem.list3) ←                    (169)
        Append(list1,list2,list3)
```

As far as the first argument is concerned clauses (168) and (169) are mutually exclusive and we can place a "|" in the first clause.

```
Append(∅,list,list) ← |                                  (170)
Append(elem.list1,list2,elem.list3) ←                    (171)
        Append(list1,list2,list3)
```

We have removed no logical condition in clauses (170) and (171); (170) and (171) have the same logical meaning as (168) and (169) and both programs express the Append relation.

In the program Append given by the clauses (170) and (171) we control the evaluation so that it will not search for alternative evaluations if (170) is valid. If we use the program for appending two lists we get an equivalent, but slightly more efficient evaluation than in Append in (168) and (169). But, on the other hand, if we use the two programs with other combinations of arguments with given values, e.g. for dividing a list, the programs will no longer yield equivalent evaluations.

We can, for instance, ask the question

Append(list1,list2,Blue.Yellow.Red.∅)

An evaluation will only give the answer

list1 = ∅, list2 = Blue.Yellow.Red.∅

The same question evaluated by (168) and (169) yields the answers

x = ∅, y = Blue.Yellow.Red.∅,
x = Blue.∅, y = Yellow.Red.∅,
x = Blue.Yellow.∅, y = Red.∅, or
x = Blue.Yellow.Red.∅, y =∅

6.1.2 The Cases Construction

Another solution to the problem of unnecessary evaluations is the Cases construction. If we have a definition in which the conditions consist of a number of alternatives or *cases* that exclude one other, we can use a construction that shows clearly that the definition is of this type and that will exclude unnecessary evaluations. Suppose that we have a definition with a number of cases with discriminating conditions.

$$
\begin{aligned}
P \leftarrow\ & Q_1|,\ R_1 \\
& \ldots \\
P \leftarrow\ & Q_n|,\ R_n \\
P \leftarrow\ & R_{n+1}
\end{aligned}
$$

We can express the same definition with the construction Cases. Cases is written before one or several conditions in a definition to express the fact that the conditions constitute different cases. The cases consist of two parts, separated by the symbol ":". The discriminating conditions are placed before : and the expressions to be evaluated if the conditions are satisfied, are placed after it. The first discriminating condition that is satisfied guides the evaluation to the expression that is placed after :.

$$
\begin{aligned}
P \leftarrow\ & \\
& \mathsf{Cases}\ (Q_1 : R_1, \\
& \qquad \ldots \\
& \qquad Q_n : R_n, \\
& \qquad \mathsf{True} : R_{n+1})
\end{aligned}
$$

Q_1 – Q_n denote the discriminating conditions. In the last case True is the condition and this entails that if none of Q_1 – Q_n succeeds, R_{n+1} will be the continuation. Q_i and R_i may consist of several expressions and parentheses express that several expressions belong to one or other of the parts of a case construction.

We can express Intersection using the Cases construction. The different cases that exclude each other are the first list being empty or non-empty. The case of the constructed list can be divided into two more cases. The first case is when an element in the first list belongs to the second list and the second case is when the element does not belong to the second list.

```
Intersection(list1,list2,list3) ←
      Cases (list1 = ∅ : list3 = ∅,
             list1 = elem.rlist1 :
                 Cases (Member(elem,list2) : (Intersection(rlist1,list2,rlist3),
                                              list3 = elem.rlist3),
                        True : (Intersection(rlist1,list2,list3))))
```

The same goes for this solution as for Append in (170) and (171). For the first and third arguments the first case is discriminating, but not for the second argument. If we want to take this into consideration in the definition, we must express the relation in two clauses.

```
Intersection(∅,list,∅) ←
Intersection(elem.list1,list2,list3) ←
      Cases (Member(elem,list2) : (Intersection(list1,list2,rlist3),
                                   list3 = elem.rlist3),
             True : (Intersection(list1,list2,list3)))
```

6.2 Ordering of Conditions

Besides obtaining efficient computation by avoiding fruitless alternatives the order of the evaluation of the conditions can be controlled so that the computation will be efficient for different input-output patterns.

An ordering of the evaluation of conditions according to different input-output patterns may be realized by formulating a definition using *control alternatives*[1]. A clause that appears as a control alternative is not a logical alternative in a definition and consequently it does not influence the meaning of the program. Control alternatives are marked by a notation and they differ from each other only in the order of the conditions. We add a control primitive expressing the input-output pattern for the arguments in the conclusion in a clause, :Pattern(a_1, a_2, ..., a_n). The requirements that an argument is used as input or output are expressed by + or - respectively. When there are no requirements on an argument ? is used.

A definition of a predicate P with control alternatives as well as logical alternatives can be formulated as follows.

P(x,y):Pattern(+,?) ← Q(x,z), R(y,z)
P(x,y):Pattern(?,+) ← R(y,z), Q(x,z)
P(x,y):Pattern(-,-) ← Q(x,z), R(y,z)
P(x,x) ←

Let us study an example. By declaring input-output patterns the evaluation as to questions of when a construction of blocks forms an arch may be controlled in a suitable way. Let us characterize an arch as a structure consisting of two towers upon which a top block is placed, A(tower1,topblock,tower2), and a tower as a structure consisting of one or several blocks stacked on each other, T(top,rest). Assume that we can use the following blocks.

Block(A) ←
Block(B) ←
Block(C) ←
Block(D) ←

Further, assume that some blocks are placed on other blocks. The relation On(A,B) expresses the fact that the block B is placed on the block A. When a block is on a tower it is placed on the top of the tower. Furthermore, a tower consists of one block or stacked blocks.

On(A,B) ←
On(C,B) ←
On(D,C) ←

---

[1] "Annotated control alternatives" are proposed by K.L. Clark, F.G. McCabe, and S. Gregory, "IC-Prolog Language Features", in K.L. Clark and S-Å. Tärnlund (eds.), *Logic Programming*, Academic Press, New York, 1982

```
On(T(top,rest),block) ← On(top,block)

Tower(x) ← Block(x)
Tower(T(top,rest)) ← Block(top), Tower(rest), On(rest,top)
```

Finally we give the definition of arch where we prepare for different kinds of questions by using control alternatives. A structure A(t1,b,t2) is an arch if t1 and t2 are towers, b is a block and b is placed on t1 and t2. The efficiency of the evaluation of a question about arches is closely connected to the order of the conditions. We can find a set of control alternatives for the definition. We assume that questions such as "Is x an arch?", "Is there an arch with x as a tower?", and "Is there an arch with tower x and top block B?" are frequent. According to these questions the following control alternatives improve the computation.

```
Arch(A(t1,b,t2)):Pattern(+,-,-) ←
      Tower(t1), On(t1,b), Block(b), On(t2,b), Tower(t2)
Arch(A(t1,b,t2)):Pattern(+,+,-) ←
      Tower(t1), Block(b), On(t1,b), On(t2,b), Tower(t2)
Arch(A(t1,b,t2)):Pattern(-,-,+) ←
      Tower(t2), On(t2,b), Block(b), On(t1,b), Tower(t1)
Arch(A(t1,b,t2)):Pattern(-,+,-) ←
      Block(b), On(t1,b), Tower(t1), On(t2,b), Tower(t2)
Arch(A(t1,b,t2)):Pattern(?,?,?) ←
      Tower(t1), Tower(t2), Block(b), On(t1,b), On(t2,b)
```

Another way of adapting the evaluation order of the conditions to the input-output pattern is to delay the evaluation of some conditions until necessary values are computed. Reasons for variables to be bound can be that there are built-in predicates such as arithmetic operations that need values in order to perform a computation. Other reasons for delaying computation may be efficiency in computation as well as clearness in definitions.

One way of controlling the evaluation according to the *binding requirement*[2] is to use a control primitive that states that a variable shall be bound when a condition is evaluated. We express this by adding a control primitive :Bound(var) to the condition that has binding requirements. The argument is a list of variables that have to be bound. For instance, we can make definitions of two predicates as follows.

```
M(x) ← N(x), P(x,y), T(x,y)
P(x,y) ← Q(x), R(x,y):Bound(x.∅), S(x,y)
```

The logical meaning of this program is the same as without the control

---

[2] A. Colmerauer, *Prolog II, Reference Manual and Theoretical Model*, Report, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Marseille, France, 1982.
L. Naish, *Negation and Control in PROLOG*, Dept. of Computer Science, Univ. of Melbourne, Australia, 1985.

primitive but the evaluation of R(x,y) will be delayed until x is bound to a non-variable. Thus in this example the evaluation of R(x,y) when x is unbound has to wait for the evaluation of S(x,y) in the same clause or T(x,y) in the clause above.

A program type for declarative definition of problems where the solution has to satisfy many demands is a so-called generate-and-test program.

Solution(x) ← Generate(x), Test(x)

Control according to binding requirements can be used for making this type of program both efficient and clear. The addition of binding requirements still allows the program to be partitioned into a generate part and a test part while procedurally a test of the generated items can be performed as soon as an item is generated. The program evaluation will take the form of restriction-generating where the definition of the restrictions contains binding requirements, i.e. the generating part will be activated by the restriction part and produces values only when they are needed.

Solution(x) ← Restriction(x),Generation(x)

We will look at a program example where the problem is to place 4 queens on a chess board with 4 rows and 4 columns in such a way that they cannot attack each other. A queen can attack another on the same row or column or on the same diagonal. A naive solution to this problem is to generate successively possible locations of the queens and test whether they satisfy the necessary demands. The possible locations are the locations on the 4 x 4 chess board. The demands are that no queen be located on the same row, column, or diagonal as another. We let the program produce the four safe locations by means of the restriction that the queens shall all be located safely. In the definition of Restrictions we ask for a location of a queen only when we are about to check that the location is safe. The relation Member is defined in Round 3.

```
4_queens(locations) ←
        Restrictions(locations), Generate(locations)

Restrictions((queen1,queen2,queen3,queen4)) ←
        Safe(queen2,queen1):Bound(queen2.queen1.∅),
        Safe(queen3,queen1):Bound(queen3.queen1.∅),
        Safe(queen3,queen2):Bound(queen3.queen2.∅),
        Safe(queen4,queen1):Bound(queen4.queen1.∅),
        Safe(queen4,queen2):Bound(queen4.queen2.∅),
        Safe(queen4,queen3):Bound(queen4.queen3.∅)

Safe((row1,col1),(row2,col2)) ←
        row1 ≠ row2,
        col1 ≠ col2,
        row1 + col1 ≠ row2 + col2,
        row1 - col1 ≠ row2 - col2
```

```
Generate(((r1,c1),(r2,c2),(r3,c3),(r4,c4))) ←
        Generate((r1,c1)), Generate((r2,c2)),
        Generate((r3,c3)), Generate((r4,c4))
Generate((row,col)) ←
        Number(row), Number(col),
        Member(row,1.2.3.4.∅),
        Member(col,1.2.3.4.∅)
```

In this way the declarativity is preserved although the evaluation is efficient as generating and testing are interwoven. The locations for the first and the second queens will be produced and then checked that they are safe. New locations will be produced until a safe one is produced. Then locations for the third queen will be suggested such that all three have safe places. Possibly, new locations must be suggested for the first two. The fourth queen is placed analogously. A solution without binding requirements will either be defined as a generate-and-test program and will produce locations for all four queens at each suggestion or it will be defined such that the queens are generated and tested successively. In the latter case the description of the problem will be less clear.

6.3 Parallelism

Parallelism is interesting for the task of making program computation more efficient. By parallelism in computations we mean that several processors work simultaneously on different parts of the computation. In conventional programming languages it is not always easy to see where parallelism might come in and how to express the control of it. In logic programs parallelism comes naturally. It is tied to the logical connectives in the program definitions.

A conditional part of a program clause or a question consists of a number of predicates connected by a logical *and*. Parallel evaluation of a conditional part or a question would entail the predicates all of which shall hold for the conclusion or the query to be valid, being evaluated at the same time. Such an interpretation comes naturally and no special construction is needed to express this, unless we want to control the evaluation to occur in parallel sometimes and sequentially at other times. Evaluating the conditions in parallel is called *and-parallelism.*

A definition of a logic program consists of a number of clauses each of which are part of the definition of the predicate and constitute alternative or complementary parts of the definition. At least one of these clauses shall hold for the predicate to hold. The evaluation of these clauses to determine whether any of them are valid can be done in parallel. A transformation of the definition into a single clause forms a composite conditional part where a logical *or* connects the different conditional parts. Trying the clauses of a predicate in parallel is called *or-parallelism.*

Let us see how these two types of connective parallelism in logic programs work. When we have a clause with more than one condition, such as

$$P \leftarrow Q, R$$

and a question P, the conditions Q and R are evaluated in parallel. Both must be valid for P to be valid and not until both evaluations are completed is the evaluation of P completed. However, the conditions may share variables which will complicate the evaluation. If Q and R have variables in common the evaluations are not independent of each other and they must be syncronized. Let us study a program with shared variables.

$$P(x,y) \leftarrow Q(x,z), R(z,y)$$

For P to hold Q and R must hold and in this case with unifiable values for z. Without communication between the evaluations of Q and R, and without control of them, action would be taken only when the difference has been established, i.e. when both values have been computed. The issue then arises as to whether one of the values can be kept, and if so, which. The evaluation is more efficient if unnecessary evaluations are not carried out.

One way of solving the problem of shared variables and preventing unnecessary computations resulting in different values, is to declare by notation which of the predicates in a conditional part is the *producer* of the value of a common variable[3]. We use a control primitive :Producer(variable) to express which of the conditions is the producer of the value of a variable. The other conditions sharing this variable we regard as consumers.

$$P(x,y) \leftarrow Q(x,z), R(z,y):Producer(z), T(x,z)$$

The evaluations of the consumer predicates cannot be performed unless the producer predicate has produced the value; consumer predicates have to wait until producer predicates have computed the values. The advantage of a method with notations for producers is that the relevant fact that one of the conditions is actually the producer is clearly expressed. The drawback is that the evaluation will be more sequential than necessary.

Another solution to the problem of shared variables that gives less sequentiality than the producer method is *binding protection*[4], a method where a variable is not allowed to be bound in the evaluation of certain conditions. We express the fact that some variables in a condition are protected from being bound by adding the control primitive :Protected(variable) after the condition.

---

[3] "Predicates as producers" are proposed in K.L. Clark, F.G. McCabe, and S. Gregory, "IC-Prolog Language Features", in K.L. Clark and S-Å. Tärnlund (eds.), *Logic Programming*, Academic Press, New York, 1982

[4] Read-only annotation is proposed by E. Shapiro, *A Subset of Concurrent Prolog*, ICOT Technical Report, Tokyo

If there is more than one variable, a list of variables is protected. We reformulate the program above by protecting arguments of conditions from being bound.

$$P(x,y) \leftarrow Q(x,z):Protected(z),\ R(z,y),\ T(x,z):Protected(z)$$

That an argument in a condition is protected from instantiation means that in the evaluation of the condition the argument may only be unified with a variable when it is itself a variable. If the argument is a non-variable when the unification occurs it may also be unified with a non-variable. If it is a variable, but the term it is to be unified with is a non-variable, the evaluation of the condition will have to be postponed until the protected argument has a value.

We continue with or-parallelism, the other kind of parallelism in Prolog programs. Given a definition

$$\begin{aligned} P &\leftarrow Q \\ P &\leftarrow R \\ P &\leftarrow S \end{aligned}$$

and a question P, or-parallelism is when all clauses in the definition of P are evaluated at the same time. Here Q, R, and S will all begin to be evaluated at the same time. The one that is completed first will be the one to show that P holds.

To go on evaluating all clauses until one of them is fully evaluated may be inefficient. The computations may be lengthy, even infinite, and P may be partly deterministic. An ability to note that part of the conditional clause will suffice for it to be chosen as the clause to demonstrate P, shortens the evaluations. We use *guarded commands*[5], explicitly annotated conditions, to express the sufficient part of the conditions. The control primitive here is : following the conditions forming the guard in a clause.

$$\begin{aligned} P &\leftarrow Q1:\ Q2 \\ P &\leftarrow R1:\ R2 \\ P &\leftarrow S1,\ S2:S3 \end{aligned}$$

In this example, Q1, R1, and S1, S2 are guards and the guard whose evaluation has been completed first states which of the clauses of P shall be used in the evaluation of P. Thus the parallel evaluation stops at the guards; it is enough that one of the clauses holds and that the clause is chosen whose guard was first fully evaluated. The guards express determinism; no backtracking to an alternative clause occurs. This implies, for instance, that when R1 succeeds and R2 fails, P fails even though Q1 and Q2 may succeed. The difference from

---

[5] Guarded commands were introduced to logic programming by K.L. Clark, and S. Gregory, "A relational language for parallel computing", in Proceedings of the ACM Conference on Functional Languages and Computer Architecture, October 1981.
Guarded commands are used in Concurrent Prolog: E. Shapiro, *A Subset of Concurrent Prolog*, ICOT Technical Report, Tokyo.

sequential evaluation is that this cannot be controlled by manipulating the order between the clauses.

Let us look at the Quicksort program for sorting a list of numbers where the evaluation is to be performed in parallel. The ordered permutation of an empty list is an empty list. The ordered permutation of a constructed list is obtained by partitioning the list into two smaller lists, one for all elements less than a discriminating value and one for all greater than this value, ordering these lists and appending them.

```
Quicksort(list,sorted_list) ← list = ∅:
        sorted_list = ∅
Quicksort(list,sorted_list) ← list = first.rest:
        Partition(first,rest,list1,list2),
        Quicksort(list1,sorted_list1):Protected(list1),
        Quicksort(list2,sorted_list2):Protected(list2),
        Append(sorted_list1,first.sorted_list2,sorted_list):
          Protected(sorted_list1.first.sorted_list2.∅)
```

We assume that the clauses as well as the conditions are evaluated in parallel. The guards test whether the first argument is an empty list or a constructed list. The shared variables are marked with :Protected such that Partition will compute the values for list1 and list2 and the two Quicksort conditions will compute the values for sorted_list1 and sorted_list2 respectively.

The partitioning divides a list into a list with all elements smaller than a discriminator and a list with all elements greater than it. Here we use guarded commands in the condition parts of the clauses.

```
Partition(elem,list,list1,list2) ← list = ∅:
        list1 = ∅, list2 = ∅
Partition(elem,list,list1,list2) ← list=first.rest, first ≤ elem:
        list1 = first.rest1:Protected(first.rest1.∅),
        Partition(elem,rest,rest1,list2):Protected(rest)
Partition(elem,list,list1,list2) ← list=first.rest, first > elem:
        list2 = first.rest2:Protected(first.rest2.∅),
        Partition(elem,rest,list1,rest2):Protected(rest)
```

6.4 Procedural Interpretation of Logic Programs

The view that a definition expresses *what* holds when a relation is true, is called *declarative* interpretation[6] of the definition. We say that A holds if B_1 and B_2 hold.

$$A \leftarrow B_1, B_2$$

[6] See R. Kowalski, *Logic for Problem Solving*, North-Holland, Amsterdam, 1979.

We have mainly used the declarative interpretation for explanations of the programs. In this chapter, however, we deal with how the values are computed from the programs, i.e the procedural meaning of the program. We are interested in the procedure for finding a resolution proof, i.e. the sequence of steps the Prolog system performs to find an answer to a question.

The declarative explanation of programs lies near at hand when we use logic as definition language. Another type of explanation was fostered in the programming tradition: a program is seen as a description of how something is to be computed; the program expresses an algorithm for solving a problem.

When solving complex problems the problems need to be structured and divided into subproblems. Within the Algol group of languages[7] the concept *procedure* is used for parts of programs. A procedure computes a well-defined subproblem and if the problem is not trivial, its solution can be divided into procedures that compute subproblems at a lower level.

The notion of programs as descriptions of computations and as built up by procedures is adopted in logic programming. The approach that a program describes *how* a value shall be computed is called *procedural interpretation.* A logic program can be seen as a set of procedures. A procedure can be separated into a procedure *head* and a procedure *body.* A Horn clause can also be separated into head and body: the consequence of the clause corresponds to the procedure head to which a call can be matched. The conditions of the clause together express the body of the procedure and are seen as new calls to procedures. We call A the procedure head and B_1, B_2 together make up the procedure body in the following clause:

$$A \leftarrow B_1, B_2$$

The procedural interpretation includes a search order and selection principle. A procedural interpretation of the expression above reads as follows: when procedure A is called, B_1 is first called, and then B_2.

A logical statement that we want to prove or a question we ask about a relation in the procedural interpretation corresponds to a call that activates a computation. The computation yields the result true, with those values which the variables may have been unified with during the computation, or false. Viewing *derivations as computations* has promoted the development of logic programming.

A procedure computes output data, given certain input data, and a predefined distinction between input and output arguments in the procedure head is a part of the procedural interpretation. Such a distinction is not necessary in logic programs; we can use a relation to do different kinds of computations by varying the instantiation pattern. If we, however, are interested in questions with a specific input-output pattern for a predicate, we can have a more effi-

[7] See N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

cient evaluation by taking into account the fact that values are given for certain arguments.

In imperative languages a variable denotes a location that contains a value or a pointer to a structure. The contents is changed during execution and with it the state of the program. A consequence of this is that the computations must be executed in a specific order. In imperative procedural languages in which the programs are divided into smaller units it is meaningful to define the scope of a variable. Variables that can be used and manipulated by several procedures have a global scope. Variables whose scope is declared only inside a procedure are called local.

In a logic programming language a variable does not correspond to a variable in an imperative language. A *logical variable* can be instantiated to a value that will make the relation come out true or false. If an instantiation yields a non-valid relation, a new instantiation may be made by backtracking. There is no correspondence to setting and changing the value of the variable in the case of logical variables. They denote only terms related to expressions in the clause where they appear.

In procedural programming languages a procedure describes *one* algorithm. When we make a call only one procedure head can match the invocation. There are no alternative algorithms and no choice is made. Such a program is said to be *deterministic.* A definition of a logic program may have several procedure heads and more than one of them can match the same call. We can regard this as a way of expressing *non-determinism* in logic programs. In the program

$$\begin{aligned} \mathsf{P(x,y)} &\leftarrow \mathsf{Q_1(x,y)} \\ \mathsf{P(x,y)} &\leftarrow \mathsf{Q_2(x,y)} \end{aligned}$$

for instance, both clauses can match the call P(1,x) and give alternative answers on backtracking.

The procedural interpretation helps us to express *efficient programs.* One computation is more efficient than another if it consists of fewer derivation steps. We need to know Prolog's search order, its selection principle, and to consider how computations are executed to be able to express how a problem should be efficiently computed. To express an efficient program we can use the control constructions that are described in previous sections. These are outside the logic of the program, like the search order and the selection principle, and a procedural approach is applicable for an understanding of them.

We shall study the procedural interpretation of some well-known logic programs. We begin with the Member relation, defined for an element and a list. Suppose that we want to use it to determine whether an element belongs to a list.

$$\begin{aligned} &\mathsf{Member(u,v.x)} \leftarrow \\ &\qquad \mathsf{u = v\ ;\ Member(u,x)} \end{aligned}$$

The procedural interpretation would be as follows: to decide whether u

belongs to the list v.x we first have to determine whether u = v. If u = v, we are through; otherwise we go on traversing the list with a new call to Member.

Another example is Paternal_grandparent(x,y) in which we conceive of questions with at least the first argument given.

```
Paternal_grandparent(x,y) ←
        Father(x,z),
        (Mother(z,y) ; Father(z,y))
```

Calling Paternal_grandparent with the arguments x and y where x is given a value and y is unbound, we first activate a call to Father with the argument x to get a value for z. If x has a father z, the procedure Mother is called. If the invocation succeeds, y gets a value and this is the value we are looking for. If the call to Mother does not succeed, Father is called with the same arguments and on success returns the desired value for y.

If we plan to use the program to efficiently find out if a certain person is a grandparent, we have to schedule the calls in the body in another order.

```
Paternal_grandparent(x,y) ←
        (Mother(z,y) ; Father(z,y)),
        Father(x,z)
```

We then search for the person x who has z as father where z's mother or father is y. To make as few searches as possible we first determine whom y is parent of. We first call Mother(z,y). If the call is successful, we invoke Father(x,z), and, if successful, it yields a value for x which denotes the grandchild. If the call to Father(x,z) fails, Father(z,y) will be invoked. If Mother(z,y) yields no result Father(z,y) will be called and if it succeeds, Father(x,z) will be called next. If this invocation terminates successfully, the value for the grandchild is given. A declarative interpretation of these clauses shows that the second definition of paternal grandfather is a control alternative to the first. When the input-output pattern control primitive is used such facts are obvious.

Another example is the relation Append for three lists.

```
Append(∅,y,y) ←
Append(u.x,y,u.z) ←
        Append(x,y,z)
```

Let us make a procedural interpretation of Append with the first and second arguments as input data and the third as output data.

When we want to append two lists and have access to the first argument, we have to traverse the first list and successively transmit the elements to the list that is to contain all the elements. When the first list is empty all its elements have been put in the resulting list which now lacks only the elements from the second list. We can add the whole of the second list to the end of the composite list.

Let us also study a program that solves the problem of moving a farmer, a wolf, a goat, and a cabbage to the other side of a river. The farmer, the wolf,

the goat, and the cabbage are all on the north bank of a river and the problem is how to move them to the south bank. The farmer has a rowing boat and he can take one passenger at a time. The goat cannot be left with the wolf unless the farmer is with them. The cabbage also counts as a passenger and cannot be left with the goat if the farmer is not there.

We will formulate a Prolog program to solve the task. We can conceive of the problem as a transition from one state to another. We know the limitations of the transition from one state to another. A state expresses the position of the individuals. We use the relation State with one argument for each individual. State(N,N,N,N) means that the farmer, the wolf, the goat, and the cabbage are all on the north side of the river. The place of the arguments denotes who and the values denote where the individual is.

We formulate the problem in following way. The state we want to reach, State(S,S,S,S) with everyone on the south bank, is, we know, safe and thus valid. It constitutes our first clause

```
State(S,S,S,S) ←
```

A state is valid if it is related to another state by valid transitions and the state obtained is safe. A transition is a passage of the farmer and possibly one passenger to the opposite bank of the river.

```
State(f,w,g,c) ←
        Opposite(f,f1) ,
        Passenger(f1,w,g,c,w1,g1,c1) ,
        Safe(f1,w1,g1,c1) ,
        State(f1,w1,g1,c1)
```

Opposite expresses the fact that S is the opposite of N. Passenger denotes the fact that any of the creatures can be a passenger if it is on the same side as the farmer. If it is a passenger its state will change and the animal and the farmer will move to the same side. The farmer may also go alone and none of the creatures change its state.

```
Opposite(N,S) ←
Opposite(S,N) ←

Passenger(x1,x,y,z,x1,y,z) ←
        Opposite(x,x1)
Passenger(y1,x,y,z,x,y1,z) ←
        Opposite(y,y1)
Passenger(z1,x,y,z,x,y,z1) ←
        Opposite(z,z1)
Passenger(x1,x,y,z,x,y,z) ←
```

The problem specification expresses the fact that some animals must not be left without the farmer on one side of the river, i.e. the states that are unsafe.

We can express the problem in terms of safe states instead of unsafe ones. In this way we express the positive conditions, what *holds* for the relation. For a state to be safe the goat must not be left with the wolf or the cabbage unless the farmer is there, too. In other words, a situation is safe when the farmer and the goat are on the same side or if the goat is alone on one side.

```
Safe(x,y,x,z) ←
Safe(x,x,y,x) ←
        Opposite(x,y)
```

The declarative interpretation convinces us of that the above definition expresses the problem text correctly in logic. Our task was to move our four friends from the north side to the south. We ask the question can they reach the south side if they are all on the north: ← State(N,N,N,N).

Since everybody is on the north bank the farmer has to row them across to the south bank. According to our description the farmer will first bring the goat to the south side, then go back and fetch the wolf. Since the wolf and the goat cannot be left with each other, the farmer will bring the wolf back to the north bank. Then he will make a new attempt to take the wolf with him and again he will have to bring the wolf back. The farmer will get tired and not get any closer to his goal. The procedure for transportation will not be found be our program. Something has to be done.

By remembering what states we have passed and only allowing new states to be valid we solve the problem. We add a last argument to the list of states.

```
State(S,S,S,S,slist) ←
State(f,w,g,c,slist) ←
        Opposite(f,f1),
        Passenger(f1,w,g,c,w1,g1,c1).
        Safe(f1,w1,g1,c1),
        Not Member(T(f1,w1,g1,c1),slist),
        State(f1,w1,g1,c1,T(f1,w1,g1,c1).slist)
```

The question we pose is

```
← State(N,N,N,N,∅)
```

A trace of this evaluation shows how the problem of moving to the other side of the river is solved.

6.5 An Example

We will study the algorithm for unification of two expressions introduced in Round 2, in greater detail. We have two expressions which we represent by the variables a and b.

The expressions are formulated according to the following syntax:

- variable names are written with an initial lower-case letter,
- constants begin with an initial upper-case letter,
- a composite expression is formed by the constructor ".", e.g. the relation "Father(x,y)" is written "Father.x.y", i.e. the relational name with arguments is represented as a list.

The relation Unifiable(a,b,subst,re_subst) has the two expressions a and b and two substitutions as arguments. One substitution consists of the pairs we have found so far, and the other is the final substitution. We find a substitution pair in each recursive step of the definition. When we can no longer find a difference between the expressions a and b, the resulting substitution subst is equal to re_subst. In every step subst represents the substitutions found so far and applied to a and b.

When the difference between the two expressions is found the conditional part of the relation Unifiable is split in two cases using the control primitive Cases.

- We get the first case when the difference between the two expressions is empty and
- the second case when the difference consists of a list with at least one element,i.e. the expressions a and b are not equal. The differences have to be compatible. A difference set is compatible when the relation Negotiable is satisfied. When this condition is satisfied, we select the first element in the difference list as the next substitution pair. The substitution pair is applied to the substitution we have found so far and to the expressions a and b. The new expressions are next_a and next_b, respectively. These two new expressions have to be unifiable, i.e. we have one more condition requiring that the relation Unifiable be satisfied for these new expressions, the new substititution and the final substitution.

```
/* ROBINSON'S Unification algorithm from LOGIC: FORM and FUNCTION */
Unifiable(a,b,subst,re_subst) ←
        Difference(a,b,d),
        Cases (d = ∅ : subst = re_subst,
               d = df.drest : (Negotiable(d),
                               Component_subst(subst,df,next_subst),
                               Substitution(a,df,next_a),
                               Substitution(b,df,next_b),
                               Unifiable(next_a,next_b,
                                         df.next_subst,re_subst)))
```

The relation Difference has three arguments, a, b, and d. a and b represents expressions and d represent a list of differences between the expressions. The definition is divided into five cases:

- the two expressions are equal and the difference is an empty list
- both expressions are composite expressions, i.e. lists having the same length. The auxiliary relation Composite_difference checks the relation Difference for each list element.
- only the second expression is composed; we have found one difference (a,b)
- the first expression is a variable
- the second expression is a variable or a constant and the first expression is a constant. We let this expression be the first component in the difference pair:

```
Difference(a,b,d) ←
      Cases (a = b : d = ∅,
            (Composite(a), Composite(b)) :
                  (Length(a,n), Length(b,m),
                  Cases (n = m : Composite_Difference(a,b,d),
                        True : d = (a,b).∅)),
            Composite(b) : d = (a,b).∅,
            Variable(a) : d = (a,b).∅,
            True : d = (b,a).∅)
```

The relation Composite_Difference separates the two cases when the expressions are composed of at least two elements or consist of only one element. In the first case we have two difference sets, difference and diffr, if the conditions Difference and the recursive condition Composite_Difference are satisfied. We are interested in the union of these sets.

```
Composite_Difference(a,b,diff) ←
      Cases ((a = af.ar, b = bf.br) : (Composite_Difference(ar,br,diffr),
                                       Difference(af,bf,difference),
                                       Union(difference,diffr,diff))
            True : Difference(a,b,diff))
```

The relation Union is satisfied for three sets. If the first set is empty the other two are equal. If the first set has at least one element we get two cases: either the first element of the first set occurs in the second set or it does not. An element occurs only once in a set.

```
Union(∅,m,m) ←
Union(ef.er,m,nm) ←
      Cases (Member(ef,m) : Union(er,m,nm),
            True : (Union(er,m,nmr), nm = ef.nmr))
```

The relation Union makes use of the relation Member.

```
Member(u,vf.vr) ←
      Cases(u = vf : True,
            True : Member(u,vr))
```

A condition for two expressions to be unifiable is that their differences be negotiable. In each difference pair one component has to be a variable. If any of the components is a variable it is the first component, because of the construction of the pairs in the Difference relation. We also constructed the pairs in the Difference relation so that composite expressions were always placed as the second component. For a difference to be negotiable, neither component may occur in the other. Since we have placed composite expressions as second component throughout, we only need check that the first component does not occur as part of the other.

```
Negotiable((a1,a2).ar) ←
        Variable(a1), Not Occurs(a1,a2),
        (Negotiable(ar); ar = ∅)
```

An expression occurs in another if the expressions are identical or if the second expression consist of parts and the first expression occurs in any of these parts.

```
Occurs(x,y) ←
      Cases (x = y : True,
            Composite(y) : (y = yf.yr,
                            (Occurs(x,yf) ; Occurs(x,yr))))
```

Let us return to the definition of Unifiable and find out what conditions remain to be defined. The relation Substitution has three arguments, a, subst, and next_a. The variables a and next_a represent expressions and the variable subst a substitution pair. The relation is satisfied if the substitution pair subst applied to a is equal to next_a. If the pair subst consists of the components u and v, all occurrences of u are exchanged for v. If the expression a is composed, we use an auxiliary relation Composite_substitution to perform the substitution on all parts of the composition.

```
Substitution(a,subst,next_a) ←
      subst = (u,v),
      Cases(a = u : next_a = v,
            Composite(a) : Composite_substitution(a,subst,next_a)
            True : next_a = a))
```

The relation Composite_substitution differs from the relation Substitution in that the first and the third arguments are lists. We get two cases when the expression a consists of at least two elements and when a consists of only one element.

```
Composite_substitution(a,subst,next_a) ←
      Cases(a = af.ar : (next_a = next_af.next_ar,
                         Substitution(af,subst,next_af),
                         Composite_substitution(ar,subst,next_ar)),
            True : Substitution(a,subst,next_a))
```

New substitution pairs are applied to those pairs that are in the substitution found so far. The relation Component_subst traverses the substitution pair by pair and performs substitutions on the second component in the pair.

```
Component_subst(∅,sf,∅) ←
Component_subst((v,f).rest,sf,(v,next_f).next_rest) ←
        Substitution(f,sf,next_f),
        Component_subst(rest,sf,newxt_rest)
```

A composite expression is constructed with the constructor "." and consists of at least two elements.

```
Composite(a) ←
        a = af.ar
```

Variables and constants are distinguished by letting variable names begin with a lower-case letter and constants begin with an upper-case letter. The ASCII-characters 97 - 125 are lower-case and 65-93 are upper-case letters. We use the predefined relation Text_Ascii. Text_Ascii is a relation between a text and a list of ASCII characters. In this case we give the text the variable name, for instance, and get its representation in the ASCII characters. We turn to the first element in the list to decide whether it is an upper-case or a lower-case letter.

```
Variable(x) ←
        Not Composite(x),
        Text_Ascii(x,letter.list),
        97 ≤ letter, letter < 126
Constant(x) ←
        Not Composite(x),
        Text_Ascii(x,letter.list),
        65 ≤ letter, letter < 94
```

Let us try to use this program. The relation has the substitution found so far as its third argument. Let us instantiate this argument to ∅ at the call.

```
← Unifiable(Father.x.y,Father.Trud.z,∅,subst)
subst = (y,z).(x,Trud).∅

← Unifiable(P.x.(F.(G.y)).(F.x),
            P.(H.y.z).(F.z).(F.(H.u.v)),
            ∅,subst)
subst = (v,G.u).(y,u).(z,G.u).(x,H.u.(G.u)).∅

← Unifiable(Trud,Magni,∅,subst)
no
```

6.6 Exercises

Exercise 1:
Suppose that we have a definition for the relation Subset where the sets are represented by lists. A list x is a subset of another list y if all elements of x occur in y.

```
Subset(∅,y) ←
Subset(u.x,y) ←
        Member(u,y) ,
        Subset(x,y)
```

Suppose that we ask the question Subset(1.3.5.∅,1.2.3.4.∅). An unnecessary backtracking will occur after element 5 has not been found in the list. Where should a | be placed to prevent unnecessary backtracking?

Exercise 2:
In the Exercises of Round 3, there is a description of a relation Subtree. Express the program Subtree with |.

Exercise 3:
Express Subset with Cases.

Exercise 4:
Union is defined using cases. Write one definition of union without any control constructions and another definition of it using cut.

Exercise 5:
Write a program that inserts an element into an ordered list such that the result is ordered. Use cut in the definition.

Exercise 6:
Write a program that takes any clause in propositional calculus and transforms it such that A $\rightarrow$ B and A $\leftrightarrow$ B are replaced by $\neg$ A $\vee$ B and $\neg$ A $\vee$ B $\wedge$ $\neg$ B $\vee$ A respectively.

Exercise 7:
Requirements for instantiation or non-instantiation in a program for determining maternal grandmotherhood would increase the efficiency of the evaluation. Write a program for maternal grandmotherhood where control alternatives make up the definition.

Exercise 8:
A relation Same_leaves(tree1,tree2) is true if tree1 and tree2 have the same leaves. It is also required that the leaves appear in the same order when the trees are traversed in the same way.

Write the definition of Same_leaves. Assume that the evaluation is performed in parallel and discuss how the evaluation occurs. What happens when the two trees have different leaves?
Further, redefine the program using binding requirements. Find the most effective control regarding the case when the trees do not have the same leaves.

Round 7. Input and Output

7.1 Input

So far we have used the arguments in a question to give input to programs and to get output from programs. But we can also use built-in predicates for controlling the input and output of programs. In this way we can construct our own interface with the program.

All built-in predicates for input and output can succeed only once, which means that they do not have alternative evaluations on backtracking.

If we want to read the next term x from the input stream we can express this by

In_term(x)

The term x may be a number, a constant, a structure, or a variable. The term must be followed by a dot and RETURN. The input stream is guided to the terminal unless something else is specified. Usually x is not *ground*, i.e. the variable has no value. The variable x is always unified with the input term except the dot. When x already has a value this value is compared to the input term. If they are equal the predicate is true, or else it is false.

We ask the Prolog system to read a term.

```
← In_term(x)          Question to the Prolog system
*>> 3.                After *>>  the input is made
x = 3                 The answer of the Prolog system
yes
```

When the argument to In_term is ground it is compared to the input term.

```
← In_term(ABC)
*>> ACB.              The input term does not
no                    correspond to the argument
```

We want to read the first element in a list and the rest of the list.

```
← In_term(x.y)
*>> 5.                The structure of the term does not
```

no correspond to the structure of the argument

The Prolog system shall read the first element of a list.

```
← In_term(x.∅)
*>> 5.∅.
x = 5
yes
```

If we want to link our input to a file instead of to the terminal, the file must first be opened for reading. This is achieved by the predicate

Inflow(file)

where file is the file name. file has to be a sequential file since our Prolog system can only handle sequential files. It is possible to keep several files open for reading at the same time. We must then state before every reading from which file the input shall come. We do this by entering Inflow(file) again. If we wish to read from the terminal we give TTY as argument. When no file has been opened, the reading is automatically from the terminal.

If we evaluate In_term(x) and reach the end of the file, x is unified with the term End_of_file. Should we try to read once again from this file, the execution is halted.

A file that has been opened for reading or writing is closed by

Close(file)

Apart from In_term we have use of a predicate that reads characters. The syntax used by the Prolog system may not correspond to the syntax we want for communication with the user. If we can read characters, we can control the reading more than is feasible with In_term; for instance, we can read control characters. It may be easier to analyze the input character by character. The input is assembled in an input buffer. A buffer is a memory that can handle a specific number of characters, often one line. The input buffer is emptied when a return character is sent and no dot is necessary to end the input.

Reading character by character can be done using

In_char(x)

where x becomes the ASCII code of the next character from the input stream. The character may be a letter, a figure, special characters such as !, space or control characters to govern shifting of lines for instance. ASCII codes for all characters are given in Appendix E.

If the variable x has a value at an invocation of In_char(x) the Prolog system will compare x to the first input character. The result will be true or false, depending on the result of the comparison. In_char(x) reads from the input stream. If we have opened a file for reading and then want the reading to be done from the terminal, we must first direct the input stream to the terminal by Inflow(TTY).

We ask the Prolog system to read two characters from the terminal.

```
← In_char(x),In_char(y)
*>>  12
x = 32          Space
y = 49          The figure 1
yes             The figure 2 is lost
```

We now read three characters from the terminal.

```
← In_char(v1),In_char(v2),In_char(v3)
*>> ab
v1 = 97      The letter a
v2 = 98      The letter b
v3 = 31      Line shift
yes
```

Let us look at an example where we read characters until we reach a line shift and store the ASCII code of the characters in a list. Prolog internally uses the code 31 for the line shift.

```
Read_to_line_shift(list)←
        In_char(char),
        Read_ascii(char,list)
Read_ascii(31,∅) ←
Read_ascii(char,char.list) ← char ≠ 31,
        In_char(char1),
        Read_ascii(char1,list)
```

```
← Read_to_line_shift(list)
*>> PROLOG is fun!
list = 80.82.79.76.79.71.32.105.115.32.102.117.110.33.∅
yes
```

7.2 Output

We can also write terms using a built-in predicate

Out_term(x)

which writes the term x in the output buffer. If x is not ground the internal variable name is written. To get the output buffer emptied we use

TTYEmpty_buffer

which writes the contents of the output buffer on the terminal and

Line_shift

which writes the contents of the output buffer in the output stream and shifts lines after output. Both TTYEmpty_buffer and Line_shift are used for writing on the terminal. But if we write a file only Line_shift can be used.

If the output is to be directed to a file, the file must first be opened for writing which is is done by

Outflow(file)

where file is the file name. Unless we already have a file with this name, a new file is created. If there already exists a file of this name, the old version is deleted. The file becomes a sequential file. When the output is finished an end mark is made on the file and it is closed by

Close(file)

The built-in predicate Close can thus be used both for closing files that have been opened for reading and for files that have been opened for writing.

We can direct the output stream to different files, just as we can the input stream. The files are then open for output at the same time. Before every change of output file, we must state on what file the output is to be made by using Output(file). If we close a file after output and then open it for writing again in the same program, we get a new file. Hence we cannot continue writing in the old file after it has been closed and opened again.

If we wish to control where on the line the writing is to be placed, we use

Tab(n)

which writes n spaces. The output is thus not directed to any particular position.

Output character by character is made by

Out_char(x)

where x is the ASCII code for the written character. The character is written in the output stream. The output is directed to the terminal unless a file has been opened for output. If an output file is opened and we want to write on the terminal, the output is directed there by Outflow(TTY). Character by character output is used, for instance, for sending control characters to the terminal to handle the graphics.

Let us write "Hi!" character by character on the terminal, one character per line and with a successive indentation by one character on each line.

```
← Out_char(72), Out_char(31),
Tab(1),Out_char(105),Out_char(31),
Tab(2),Out_char(33)
H
 i
  !
yes
```

7.3 Examples

Let us study an example in which we read information from a file and divide this information into two files. The numbers, names, and addresses of a firm's clientele are stored in the file Clients. Every fact is ended by a dot. In the file Name we store the client's number and the name, and in the file Address we write the client's number and the address. Moreover, all the clients' numbers are written on the terminal as they are dealt with.

```
/* Main program that opens the file we read from, reads the first client's
      number, executes a procedure to continue the reading and then write
      the information as well as to see to the closing of the files. */
Divide_personal_info ←
      Open_inflow,
      Read_info(client_no),
      Read_Write(client_no),
      Close_files
/* When we read a term and reach the end of the file, the term is unified
      with End_of_file. If the file is not ended we read the name
      and address and execute a procedure that writes the information
      on the output files. Then a new client's number is written. */
Read_Write(End_of_file) ← |
Read_Write(client_no) ←
      Read_info(name), Read_info(address),
      Write_client(Name,client_no,name),
      Write_client(Address,client_no,address),
      Write_on_terminal(client_no),
      Read_info(next_client_no),
      Read_Write(next_client_no)
/* We read a term from the input file. */
Read_info(x) ← In_term(x)
/* The output is directed to the file denoted by the variable file
      and we write the client's number and info in the file. */
Write_client(file,client_no,info) ← Outflow(file),
      Transcription(client_no),
      Transcription(info)
/* The client's number is printed on the terminal. */
Write_on_terminal(client_no) ←
      Outflow(TTY), Out_term(client_no), Lineshift
/* We write a term in the current output file. Only one term per line
      and the term should be finished by dot otherwise we
      cannot read the information with "In_term" in a Prolog program. */
Transcription(x) ←
      Out_term(x), Out_term('.'), Lineshift
```

```
Open_inflow ← Inflow(Client)

Close_files ←
        Close(Client),
        Close(Name),
        Close(Address)
```

We shall take another example. It is convenient to have a program that can find the birthdays within a particular month, especially if one has many friends. Let us write a program that keeps track of all the birth dates of our friends and then writes the names of all the people whose birthdays fall in a particular month on a new file. The output shall contain date, name, and the age the person reaches, and shall be sorted in date order. If a person reaches a round number of years, a mark shall be made after the number of years.

We store the birthday information in a tree in the predicate Birth_tree. The nodes in the tree consist of the structure Born(year,month,day,name) and the tree is sorted by years. The tree is at the end of the program. We write our list of persons in the file Birthdays.

We use a few predicates in our program that we have defined before. They are Read_to_line_shift above, Append in Round 3, and Length in Round 4.

```
/* The main program reads in the month which we are interested in and
        the year in question. Then it finds all the people who were born
        in this month and collects the information about them in a list and
        writes the desired list. */
Birthdays ←
        Read_month_year(month,year),
        Search_persons(month,birthday_list),
        Write_birthdays(month,year,birthday_list)

/* The input of month and year is made character by character. The line is
        divided into two parts, one for the month and one for the year,
        after which there is a check that the dates are reasonable. If they
        are not, a fault message is written and a new reading is done. */
Read_month_year(month_no,year) ←
        Out_term('Give month and year in the form: February 1987'),Line_shift,
        Read_to_line_shift(line),
        Divide_line(line,month,year1),
        (Check_input(month,month_no,year1),year = year1;
                Out_term('Wrong input'),
                Read_month_year(month_no,year))

/* All persons in b_tree are checked and those who were born
        in the right month are assembled in a list sorted by date. */
Search_persons(month,list) ←
        Birth_tree(b_tree),
        Sorted_answers((day,year,name),
                Right_month(month,b_tree,year,day,name),list)
```

```
/* The output file is opened, headings and the birthday list are written.
        Then the output file is closed. */
Write_birthdays(month,year,birthday_list) ←
        Outflow(Birthdays),
        Write_headings(month),
        Write_list(year,birthday_list),
        Close(Birthdays)


/* The line is divided into month and year. All spaces before the month,
        between the month and the year, and after the year are removed.*/
Divide_line(line,month,current_year) ←
        Remove_initial_spaces(line,new_line),
        Append(month_ascii,32.year_line,new_line),
        Text_Ascii(month,month_ascii),
        Remove_initial_spaces(year_line,year_ascii),
        (Append(year,32.list,year_ascii) ; year = year_ascii),
        Text_Ascii(current_year,year)


/* Spaces (ASCII code 32) before the text are removed. */
Remove_initial_spaces(32.list1,list2) ←
        Remove_initial_spaces(list1,list2)
Remove_initial_spaces(x.list,x.list) ← x ≠ 32


Check_input(month,month_no,year) ←
        Month(month_no,month),
        Integer(year),
        year > 1986


/* Searching for persons who are born in the right month.*/
Right_month(month,T(_,Born(year,month,day,name),_),year,day,name) ←
Right_month(month,T(left_subtree,_,right_subtree),year,day,name) ←
        Right_month(month,left_subtree,year,day,name)
Right_month(month,T(left_subtree,_,right_subtree),year,day,name) ←
        Right_month(month,right_subtree,year,day,name)


Write_headings(month_no) ←
        Out_term('Birthdays in '),
        Month(month_no,month),
        Out_term(month),Line_shift,
        Out_term('Date'),Tab(6),
        Out_term('Name'),Tab(16),
        Out_term('Year'),Line_shift,Line_shift
```

```
/* The transcription is finished when the list is empty. If the list is not
      yet emptied we write the date when a person has birthday. We write the
      information in columns, each column having a straight left side. Write_name
      puts the name in the right position in the line. Afterwards we calculate
      the age of the person and whether he reaches a round number of years.
      Is this the case we print three stars at the end of the line.*/
Write_list(current_year,∅) ←
Write_list(current_year,(day,year,name).list) ←
      Write_day(day),
      Write_name(name),
      Write_year(current_year,year),
      Write_list(current_year,list)

Write_day(day) ←
      (day < 10,Tab(1); True),
      Out_term(day),Tab(8)

/* The name column should be 20 characters long. If the name is shorter,
      the column should be filled with spaces. When the name is written
      we translate the characters in the name to a list with corresponding
      ASCII code. Then we can examine the length of the list with Length
      and calculate the number of spaces that should be written. */
Write_name(name) ←
      Out_term(name),
      Text_Ascii(name,name_ascii),
      Length(name_ascii,n),
      Value(20 − n,blanks),Tab(blanks)

Write_year(current_year,year) ←
      Value(current_year − year,age),
      Out_term(age),
      Value(age Mod 10,a),
      (a = 0, Out_term(' ***'), Line_shift; Line_shift)

Month(1,January) ←
Month(2,February) ←
Month(3,March) ←
Month(4,April) ←
Month(5,May) ←
Month(6,June) ←
Month(7,July) ←
Month(8,August) ←
Month(9,September) ←
Month(10,October) ←
Month(11,November) ←
Month(12,December) ←
```

```
Birth_tree(T(T(T(0,Born(1908,2,23,Grace),0),
        Born(1930,1,23,'Anne'),
        T(0,Born(1937,1,17,Cindy),0)),
        Born(1948,1,21,Eric),
        T(0,Born(1954,11,13,Jill),
        T(0,Born(1967,1,2,Jeff),0),Born(1969,1,2,'Dave'),
        T(0,Born(1975,8,7,Cathy),0))))) ←

← Birthdays
Give month and year in the form:  February 1987
January 1987
yes
```

The output in the file Birthdays in January:

```
Birthdays in January
Date        Name              Year

2           Jeff              20 ***
2           Dave              18
17          Cindy             50 ***
21          Eric              39
23          Anne              57
```

7.4 Exercises

Exercise 1:

Write a program that reads information from two files and assembles it into a single file. A firm's clientele with the clients' numbers and their names is stored in the file Names. Their numbers and addresses are stored in the Address file. Each entry shall end with a ".". The same persons shall be in both files and the order shall be the same. If there is anything wrong in the files so that the clients' numbers do not agree, a commentary will be written on the screen and the execution will be halted. The name of our output file will be Person. It shall contain all information from the files, i.e. client numbers, names and addresses.

Exercise 2:

In our birthday example we had all information assembled in a tree. If one has many friends it may be more convenient to store the information in a file instead. Suppose that we have a file Born that contains name, year, month and day of birth for a set of persons. Each entry is ended by a full stop. It may be handy to have the address of every person in the file if we would like to send a birthday greeting card. Extend the file to include the address

of every person. The address of each person shall be read via the terminal after the name of the person has been written on the terminal.

Round 8. Prolog Implementations

8.1 DECsystem-10 Prolog

D.H.D. Warren, L.M. Pereira, and F.C.N. Pereira developed an implementation of Prolog[1] at the Department of Artificial Intelligence at the University of Edinburgh in 1977. This implementation is for the DEC-10 under the Tops-10 and Tops-20 operating systems. The Prolog system consists of an interpreter and a compiler, both written in Prolog to a large extent.

Syntax

- Constants in DECsystem-10 Prolog consist of lower case letters and the symbol "_", starting wiht a lower case letter or enclosed by ' '.
- Variables begin with upper case letters or "_", and may also contain "_" in the variable name. A single "_" denotes an anonymous variable which we cannot refer to.
- Numbers are integers only, in the range -131072 – 131071.
- Predicate names begin with lower case letters and may contain "_".
- Lists are enclosed within "[" and "]". Elements in a list are separated by ",". The empty list is written [ ]. [a] is an example of a list with an element a. [a,b] is a list with two elements. In [X|L], X is the first element and L is the rest of the list. The constructor | can be read as "followed by". It is also possible to define the list structure "." as an operator in the system.

":-" corresponds to "←" in our Prolog system. "," and ";" have the same meaning, i.e. ∧ and ∨, respectively. All Prolog clauses and questions in the system must be ended by a dot.

Examples of clauses in DECsystem-10 Prolog:

```
ancestor(X,Y) :-                                    (172)
        parent(X,Y).
```

[1] D.L. Bowen, *DECsystem-10 Prolog User's Manual*, Department of Artificial Intelligence, University of Edinburgh, UK, 1981

```
ancestor(X,Y) :-                                    (173)
        parent(X,Z), ancestor(Z,Y).
parent('Andrew','Margaret').                        (174)
parent('Bridget','Margaret').                       (175)
```

Clause (172) is read as ancestor being true if the parent of X is Y. Clause (173) is true if the parent of X is Z and the ancestor of Z is Y. Clause (174) means that parent of Andrew is Margaret. Clauses (174) and (175) are examples of facts in the Prolog system.

The predicate ancestor can also be written as one clause:

```
ancestor(X,Y) :-                                    (176)
        parent(X,Y); parent(X,Z), ancestor(Z,Y).
```

"?-" is written by the Prolog system and may be followed by a question or a user command. Just as in our Prolog system a question may consist of several subquestions. We may request alternative evaluations by typing ";".

An example of a question to the system is

```
?- parent(X,'Margaret').
X = Andrew ;
X = Bridget
yes
```

Logical predicates

The arithmetic relations +, −, *, / and mod work in the same way as in our Prolog system. The correspondence to our Value(num_expr,res) is

Res is Num_expr

where Res is the value of the numerical expression Num_expr. The values of all the variables in the numerical expression must be known, i.e. they are ground. If the value of the first argument is also known the two values are compared.

Numbers and terms are unified by =. If X and Y can be unified the following relation will be true, else false.

X = Y

Comparisons between numbers and between terms are handled in a slightly different manner in DECsystem-10 Prolog.

Comparisons Between Numbers	
X < Y	X is less than Y
X > Y	X is greater than Y
X =< Y	X is less than or equal to Y
X >= Y	X is greater than or equal to Y
X =:= Y	X and Y have the same value
X =\= Y	X and Y do not have the same value

The arguments to the relations have to be ground.

Comparisons Between Terms

T1 @> T2	T1 comes after T2 in standard order
T1 @=< T2	T1 does not come after T2 in standard order
T1 @>= T2	T1 does not precede T2 in standard order
T1 == T2	T1 and T2 are identical (not a logical predicate)
T1 \== T2	T1 and T2 are not identical (not a logical predicate)

Standard order is an order between terms stipulated by the system in which constants are alphabetically ordered while numbers are ordered in numerical order.

There is also a predefined predicate that compares two arguments and unifies the first argument with the result of the comparison.

```
compare(R,T1,T2)
```

R is unified by = if T1 and T2 are identical. If T1 is greater than T2 in standard order, R is unified with > and if T1 is less than T2 in standard order, R is unified with <.

There is a built-in relation in the Prolog system that computes the length of a list. L is a list and N is the number of elements in the list.

```
length(L,N)
```

Sorting of a list in standard order is also predefined. L2 is the sorted permutation of L1 where duplicates of L1 have been removed.

```
sort(L1,L2)
```

True in our Prolog system corresponds to

```
true
```

in DECsystem-10 Prolog. Our False which is always false and starts backtracking corresponds to

```
fail
```

Metalogic

We will first take a look at the constructions we can use to control the execution of a program. Our cut, |, corresponds to

```
!
```

in DECsystem-10 Prolog and it stands as a condition on its own.

"Negation as failure" in the system is

```
\+ (P)
```

It differs in one point from our Not. We get an execution error if P can succeed only when a variable is bound during the derivation. In DECsystem-10 Prolog P

would succeed and consequently \+ (P) fail. \+ can be defined in the following way:

```
\+(P):- P, !, fail.
\+(P).
```

There are several metalevel constructions for controlling evaluation.

var(X)	is true if X is not ground
nonvar(X)	is true if X is not a variable
atom(X)	is true if X is a constant (not a number)
integer(X)	is true if X is an integer
atomic(X)	is true if the argument X is a constant or a number

```
functor(T,N,A)
```

is a relation between a constructed term T, the name of the constructor N, and the arity of the term A. Either T or both N and A must have a value.

```
name(X,L)
```

corresponds to our Text_Ascii. X is a constant or a number and L is the list of the ASCII codes for the characters in X.

If we want to change a database during execution we use

```
assert(P)
```

to add clause P to the database. The placement of the clause in the memory depends on the implementation. It is possible to state that we would like the clause put first among the clauses of the definition by

```
asserta(P)
```

or last by

```
assertz(P)
```

assertz corresponds to our Assert. We remove a clause from the database by

```
retract(P)
```

The first clause to match P is removed, corresponding to our Retract.

We will consider a few other metalogical constructions.

```
clause(C,A)
```

C is a consequence in a clause in the program and A is unified with the antecedent in the clause or with true if the clause does not have any conditions.

Should we want to list on the terminal the statements we have in our Prolog world, we do it by

```
listing
```

If we want to list a particular definition only, we give the name of the definition as argument, i.e.

```
listing(P)
```

where P is the name of the definition. To get several specified definitions listed at the same time we assemble the names of the definitions in a list and give the list as argument.

All_answers in our system corresponds to the construction

```
bagof(X,P,L)
```

in DECsystem-10 Prolog. All substitutions for X which will make the question P come out true are assembled in the list L. There is a variant in the Prolog system where the list L is sorted and all duplicates have been removed. The name of this variant is

```
setof(X,P,L)
```

setof is far more time and space consuming than bagof. As in our system we can use the construction

```
X^P
```

which means that there exists an X such that P is true. X^P is significant only in bagof and setof.

Communication

If we want to read the next term X which is ended by "." and RETURN we use

```
read(X)
```

To read the next term in the input stream we write

```
get0(X)
```

X is the ASCII code for the character to be read. If we want to read the next character in the input stream which is not a space or control character we use

```
get(X)
```

instead. Shall the input stream be directed to a file we open the file F for reading by

```
see(F)
```

We can also direct the input stream to the terminal by

```
see(user)
```

Before a file has been opened the input stream is automatically directed to the terminal. Character by character reading is made from the terminal even if we have directed the input stream to a file if, in place of get0 and get, we use

```
ttyget0(X)
ttyget(X)
```

Several files may be open for reading at the same time. Before every change of current input file see(F) must be given again. The file we read last is closed using

```
seen
```

A file F that has been opened for reading or writing can be closed using

```
close(F)
```

We write the term X in current output using

```
write(X)
```

The term X is written in prefix form, if we use

```
display(X)
```

Character by character writing is possible using

```
put(X)
```

where X is the ASCII code for the character to be written. To direct the output to a file we use

```
tell(F)
```

File F is opened for writing. We can direct the output to the terminal by

```
tell(user)
```

If the output has not been directed to a file the writing is automatically done on the terminal. If we have directed the output to a file we can still output character by character on the screen by using

```
ttyput(X)
```

Just as in the case of input, several files can be opened for output at the same time. Before every change of the current output stream we declare tell(F). To empty the output buffer and get the output on the terminal we type

```
ttyflush
```

If we want a line shift after output we use

```
ttynl
```

To empty the output buffer of the current output stream and to start a new line after output we type

```
nl
```

To write N space characters in the current output stream we declare

```
tab(N)
```

The file we wrote last is closed by

```
told
```

We can also close file F by close(F).

Trace

We are able to follow the execution of a program in DECsystem-10 Prolog if we first declare

```
trace
```

before asking the system a question. We can then follow all calls to predicates on the screen and see if they have succeeded. If a predicate does not succeed we see the backtracking and the attempts of the Prolog system to make an alternative evaluation. There are also possibilities to control what to trace of the execution.

Editing programs

We write and edit our Prolog programs outside the Prolog system. But when we are inside the Prolog system, we have to interpret our program file. This is done by

```
consult(F)
```

F may be a program file or a list of several files. If we make an exit from the Prolog system to change a file and then enter the same Prolog world again, we want to update the world with the changes we have made. We then type

```
reconsult(F)
```

If we had used consult again, instead of reconsult, the result would have been two simultaneous versions of the program in our world. If we want to save our entire Prolog world we can do it by typing

```
save(S)
```

The interpreted code is saved in the file S. The file is read in again by

```
restore(S)
```

Compilation

Apart from the interpreter there is a compiler in the Prolog system. The compiler is called from the interpreter. We can compile a file with

```
compile(F)
```

The compiled code for the predicates in file F is then in our Prolog world. To be able to ask a question about a predicate in compiled code the predicate has to be declared public before compilation.

```
:- public P.
```

":-" denotes that a command to the Prolog system will follow. P may be one or several predicate names with or without the number of arguments of the predicate. Suppose that we want to declare a predicate member with two arguments, and a predicate ordered without specifying the number of arguments this predicate has. In this case the public declaration will be

```
:- public member/2, ordered.
```

The Prolog system always reaches all interpreted clauses in the Prolog world from a compiled program. But it is not possible, however, to ask a question to a compiled clause, directly, or from interpreted code, without its being declared public. A compiled clause, though, can always reach another compiled clause.

The compiler can save memory space and the execution will be faster if we declare what arguments will be known and what arguments will not be known in a query to a relation. We can achieve this by mode declaring our relations. + in a mode declaration means that the argument will be known, – means that the argument will not be known, and ? means that we set no restrictions on the argument.

```
:- mode partition(+,?,-,-).
```

In the relation partition the first argument must have a value at invocation. The third and fourth arguments must be unknown. We know nothing about the second argument.

An example

To wind up we will look at a program for quick-sorting numbers in a list in DECsystem-10 Prolog.

```
quick_sort([ ],[ ]).
quick_sort([X | L],Sorted):-
        partition(L,X,L1,L2),
        quick_sort(L1,Sorted1), quick_sort(L2,Sorted2),
        append(Sorted1,[X | Sorted2],Sorted).

partition([ ],X,[ ],[ ]).
partition([Y | L],X,[Y | L1],L2):- Y =< X,
        partition(L,X,L1,L2).
partition([Y | L],X,L1,[Y | L2]):- Y > X,
        partition(L,X,L1,L2).

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

8.2 Tricia

The Tricia Prolog system[2] has been developed at UPMAIL, Uppsala University, by J. Barklund, M. Carlsson and H. Millroth, among others. The system is an extension of DECsystem-10 Prolog and Quintus Prolog. The extension consists mainly in an increased number of data types and more control possibilites for execution. Tricia has been developed for DEC-20 under the Tops-20 operating system.

One incentive for developing the system was to use the larger address space available in DEC-20 as compared to DEC-10, the machine for which DECsystem-10 Prolog was originally developed. Another incentive was to use and test some ideas from logic programming research.

The system is implemented in Prolog and assembler but will soon be implemented in Prolog and C.

Syntax

The Tricia syntax is the same as the DECsystem-10 Prolog syntax which means that constants, variables, and lists are denoted in the same way. The Tricia system contains some extensions, though.

- Numbers are integers in the interval $-2^{35} - 2^{35} - 1$ but there are also floating point numbers.
- Characters may be letters, figures or special characters like + or *, spaces and carriage return.
- Arrays are implemented in Tricia. An array is a structured term whose components have been ordered in a rectilinear coordinate system with a variable number of dimensions. A one-dimensional array is called a vector. There are two kinds of arrays, a Prolog array and a byte array. In a Prolog array every element is an arbitrary Prolog term. In a byte array the elements consist of numbers which can be contained in one byte. The byte size may vary between 1 bit and 36 bits. Important is that an array in Tricia is a logical term.
- Strings are sequences of characters surrounded by double-quotes. " " is an empty string and "This is a string" is a string of 16 characters. Strings are a special case of arrays. In DECsystem-10 Prolog a string is a set of integers but in Tricia the components are characters.
- Hash tables are a form of tables in which every line consists of a key and a value. A line is reached by a function computing the index of the line in the table. Using hash tables instead of, for instance, lists or trees gives more efficient execution. The hash tables are implemented as logical terms too.

---

[2] J. Barklund et al., *Tricia User's Guide*, UPMAIL, Uppsala University, Sweden, 1987

Logical Predicates

Most built-in relations in DECsystem-10 Prolog are also to be found in Tricia. Since the Tricia system contains both integers and floating point numbers, there are built-in predicates for computations with both kinds of numbers. Integer expressions are computed by

```
Res is Integer_expr
```

where Res is the value of the numerical integer expression Integer_expr. Floating point numbers is computed by

```
Res equals Float_expr
```

The numbers in the numerical expression Float_expr should all be floating point numbers. We can obtain the absolute value of both integers and floating point numbers using

```
abs(X)
```

and an integer X may be converted to a floating point number by

```
float(X)
```

Metalogic

The character

```
!
```

corresponds to our cut. There is also a garbage cut

```
!!
```

The only difference between the two is that garbage cut collects any storage wasted since the last non-deterministic clause.

In Tricia the character

```
\+
```

and not(P) described below are used for negation as failure.

The execution of one or several predicates may be delayed.

```
freeze(P)
```

delays the execution of the question P until there are no variables as arguments in the question. The question P can consist of one or several predicates.

```
freeze(X,P)
```

delays execution until the variable X is ground.

```
not(P)
```

corresponds to the negation \+ with the difference that the execution of P is delayed until all the arguments of P are ground.

dif(X,Y)

delays execution of X \ == Y until the variables X and Y have got values or been unified with each other.

The metalevel constructions for checking the type of a term are

is_float(X)	is true if X is a floating point number
is_list(X)	is true if X is a list
is_nil(X)	is true if X is an empty list
is_character(X)	is true if X is a character
is_array(X)	is true if X is an array
is_hash_table(X)	is true if X is a hash table
is_string(X)	is true if X is a string

There are several built-in constructions in Tricia for manipulating strings. We will look at some of them.

string_trim(S1,Char,S2)

is satisfied if S2 is a substring of S1 where the characters Char have been removed both at the beginning and at the end of S1. There is the corresponding construction for deleting the characters in Char only at the beginning of the string

string_left_trim(S1,Char,S2)

or at the end of the string

string_right_trim(S1,Char,S2)

The construction

substring(S1,Start,S2)

is satisfied when S2 is a substring of string S1 which begins at Start and ends at the end. A string is a vector character array. Hence all constructions for arrays can also be used for strings.

There are a number of built-in predicates for handling arrays. To create an array we declare

array(D,A)

where D is the dimension and A the name of the array. After creating an array we can update it by

aset(Old,S,Elem,New)

where Old is our original array, S is the subscript of the element to be updated, Elem is the element that the old array is to be updated with and New is the updated array. The array Old is not affected. The subscripts are specified in a list or as a number if the array has only one dimension.

Examples of some built-in predicates for using hash tables are

hash_table(H)

that creates the hash table H. A value is inserted into the table by

puthash(Key,Old,V,New)

where New is a hash table in which the value V has been placed in the position of the key Key in the hash table Old, without affecting Old. We reach a value in a table by

gethash(Key,H,V)

where the value V is associated to key Key in hash table H. A line in the hash table is removed by

remhash(Key,Old,New)

where hash table New corresponds to the table Old with the exception that the entry Key has been removed.

Our built-in construction All_answers corresponds to

bagof(X,P,L)

and our Sorted_answers is called

setof(X,P,L)

where L is a list of the instances of X which will satisfy the predicate P. With setof the list L is sorted and without duplicates. Another construction for finding all solutions is

findall(X,P,L)

The difference in this construction is that all variables in P are regarded as local. In the first two constructions the variables are implicitly universally quantified, which means that there may be variables in P that have values before bagof and setof. findall always succeeds but if we do not get any values of X the list L will be an empty list.

Communication

The built-in predicates we treated in DECsystem-10 Prolog hold for Tricia, too.

Trace

There is a debugging facility in the Tricia system that works almost as in DECsystem-10 Prolog. A difference is that it is possible to trace a predicate after a special pattern of argument values.

Editing Programs

We write and edit our programs outside the Prolog system. The same built-in constructions we took up in the section on DECsystem-10 Prolog are valid for Tricia. There is one difference, however; in Tricia we have access to an editor

which allows us to edit our questions to the system. This editor contains, among other things, a subpart of the commands the text editor EMACS[3] has.

Compilation

Tricia contains a compiler. Compilation of files occurs at operating system level and not inside the Prolog system. We can define a number of switches in the program for guiding the compiler so as to make the compiled code as efficient as possible.

8.3 Quintus Prolog

Quintus Prolog[4] has been developed at Quintus Computer Systems Inc. in Palo Alto, California, USA. The system is available under the UNIX operating system on Sun workstations and VAX machines as well as under VMS on VAX and DEC machines.

Quintus Prolog has much in common with DECsystem-10 Prolog, e.g. the syntax and the built-in predicates, but the programming environment is richer. It is possible, for instance, to have a text editor as interface to Prolog. The system also supplies a help system as support for the user. The compatibility between DECsystem-10 Prolog and Quintus Prolog is great.

Syntax

The syntax is the same as in DECsystem-10 Prolog. The only difference is that we can use floating point numbers in Quintus Prolog. An example of correct syntax for numbers is -1.225, 1.0E6, 12.34567e-12. The numbers have an approximate accuracy of six decimal digits. The magnitude on VAX computers is 0.29E-38 to 1.7E38 and on Sun computers the range is slightly larger.

Logical Predicates

Most of the built-in logical predicates are the same as in DECsystem-10 Prolog. However, in Quintus Prolog / denotes division between floating point numbers and // division between integers.

Metalogic

At metalevel there are, apart from all the built-in constructions in DECsystem-10 Prolog, the two constructions

```
integer(X)
```

which succeeds if the argument X is an integer and the corresponding construction if the argument is a floating point number is

---

[3] R.M. Stallman, *EMACS Manual for TWENEX Users*, Artificial Intelligence Laboratory of the Massachusetts Institute of Techology, USA, 1981

[4] *Quintus Prolog Users Guide*, Quintus Computer Systems Inc., Palo Alto, USA.*Quintus Prolog Reference Manual*,Quintus Computer Systems, Palo Alto, USA

```
float(X)
```

All predicates in Quintus Prolog are either static or dynamic. Only the dynamic predicates can be changed during execution. The change may consist of retracting a clause from, or asserting a clause to, a definition. The static predicates can be changed by the use of **consult** or in connection with compilation. The predicates are static unless we declare them to be dynamic by the command

```
:- dynamic predicate/arity
```

where **predicate** is the predicate name and **arity** is the number of arguments of the predicate. When a clause P is removed by **retract(P)**, it disappears at once from the database. As a comparison we might mention that in DECsystem-10 Prolog the clause still exists in the memory but cannot be referred to.

Communication

All reading and writing is made via streams. A stream can refer to a file or the terminal. Streams may be opened for reading or printing, not for both. Up to twenty streams may be open at the same time. A stream can be opened by

```
open(File,Mode,Stream)
```

where Mode may be **read**, **write**, or **append**. Using the mode **append** opens an already existing file and adds printout to the end of the file.

All built-in constructions for reading and printing in DECsystem-10 Prolog can also be found in Quintus Prolog. There is, moreover, a version of the predicates where an extra argument specifies the stream wanted. As an example we can take

```
read(S,T)
```

which controls the reading of a term T to the stream S.

Trace

The execution can be followed by a trace. It resembles DECsystem-10 Prolog's but it has been extended by further facilities, the main addition being the possibility to trace compiled predicates.

Editing

The EMACS[5] text editor can be used as interface to the Prolog system. It is an essential simplification to be able to go directly from the editor to Prolog and back again to make changes in a program. The Prolog world will then only have to be updated with these changes. The Prolog program is in a "text window" on the upper half of the terminal screen while the Prolog environment is on a "Prolog window" on the lower half of the screen.

---

[5] R.M. Stallman, *EMACS Manual for TWENEX Users*, Artificial Intelligence Laboratory of the Massachusetts Institute of Technology, USA, 1981.

At loading of code via EMACS to Prolog the whole file may be loaded or a particular part of the file or the definition of a particular predicate.

While we are in the Prolog world we have access to all EMACS commands all the time. This means that the queries and commands we give to the Prolog system can be edited after they have been given. It is also possible to let the editor search for a particular predicate. The "Prolog window" is thus a buffer to the editor.

If we do not want to enter Prolog via the editor, we state

```
Prolog
```

at operating system level. We can then load files in the same way as in DEC-system-10 Prolog.

Beside the EMACS editor it is also possible to use any editor to save the text in a file and then load the file into the Prolog system by consult(file), where file is the file name. A difference is that consult works as reconsult in DECsystem-10 Prolog, i.e. the Prolog world is updated with only those changes made since the last loading. At the first consult to an empty Prolog world the whole file is loaded, of course.

Compilation

```
compile(F)
```

compiles file F, or the files, if F should be a list. Predicates declared to be dynamic are not compiled but are stored in the same way as predicates loaded by consult. Compiled predicates do not have to be declared public to be reached from interpreted predicates which is a feature that differs from DECsystem-10 Prolog.

Via the EMACS editor it is possible to compile everything in the buffer, part of it, or a single predicate.

In compiled predicates the first argument is used as an index. This means that when a question is asked to a predicate whose first argument is known, a hash table is used to quickly reach the first clause of the predicate that matches the question.

Program Check and Error Messages

Apart from the syntax checker, Quintus Prolog also includes a style checker. The system gives an error message if certain stylistic conventions are not observed. These conventions are

- all clauses of a predicate should be in one and the same file
- all clauses in the definition of a predicate should be sequentially ordered in the program
- when a variable occurs only once in a clause, it shall be referred to using the symbol "_" as an anonymous variable, or else, the variable name shall begin with "_"

The error messages are more specific in Quintus Prolog than in DECsystem-10 Prolog. One facility, not normally found in Prolog systems, is that if we ask a question of a predicate which is not defined, we get an error message and the debugging facility is started and tracing is begun. In most Prolog systems the question fails and backtracking is started.

Help System

Quintus Prolog has an on-line help system containing the User's Guide and the Reference Manual to the Prolog system. They can both be entered under EMACS and in Prolog. The help system is menu-based with menus corresponding to the chapters in the manual, chapter subsections, etc. There are two ways to get information; one corresponds to looking up a subject in an index, the other to looking at the contents.

Operating System Commands

In Prolog it is possible to execute operating system commands. This means that we can reach the operating system we are running under and can make changes in our directory.

Other Programming Languages

In Quintus Prolog we can make use of programs written in other programming languages. Under the VMS operating system we may call or load programs written in Basic, C, Cobol, Fortran, Pascal, and PL1. The C programming language can be used under UNIX.

8.4 MProlog

The Prolog system MProlog[6] was developed at the Institute for Co-ordination of Computer Techniques in Budapest, Hungary. MProlog is available for the IBM 370 under VM/CMS/SP, for the Siemens 7000 under BS2000 and for the VAX-11 under VMS and UNIX.

MProlog is divided into two systems, a base system and a production system. The production system is called PDSS and is a subsystem for program development. PDSS contains an editor and possibilities for executing MProlog programs interactively. The production system transforms modules in MProlog to effectively executable code called binary modules. These binary modules are then connected to executable programs. An optimizer takes care of the binary modules and transforms them on the basis of the program structure to shorten the execution time. The base system consists of an interpreter for execution of PDSS and the user's program.

---

[6] P. Koves, *The MProlog Programming Environment: Today and Tomorrow*, Proceedings of the Workshop on Prolog Programming Environments pp. 81-89, Linköping, Sweden, 1982. Z. Farkas and P. Szeredi, *Getting Started with MProlog*

In MProlog the programs consist of one or several *modules*[7]. Programs written in DECsystem-10 Prolog can also be executed in MProlog. There are facilities for integrating user-defined standard procedures written in Prolog or in some conventional language like Fortran.

Syntax

The syntax is the same as in DECsystem-10 Prolog.

Modules

Modules consist of a number of definitions of predicates and module information. The module information contains, among other things, information on what predicates may be used outside the module and what predicates outside the module may be used inside it.

```
module modulename.
<moduleinfo>
body.
<predicate>
endmod.
```

When we define a module we begin by naming it in modulename. Then the Prolog system gets information on the module in modulinfo. After body we give definitions of the predicates that are to be included in the module, predicate. The module is ended by endmod.

Logical Predicates

The arithmetic operators are the same as in DECsystem-10 Prolog and the evaluation is made using is. Moreover, there are built-in predicates for the four rules of arithmetic.

plus(X,Y,Res)	Res is the sum of X and Y
minus(X,Y,Res)	Res is the difference between X and Y
times(X,Y,Res)	Res is the product of X and Y
div(X,Y,Res)	Res is the quotient between X and Y

Comparisons between numbers and between terms are the same in MProlog and DECsystem-10 Prolog.

Metalogic

The built-in constructions at metalevel that DECsystem-10 Prolog has exist in MProlog, too. But the system also has additional facilities and we will mention some of them.

MProlog has more constructions for database handling. We can add a clause to a definition and control where it is placed in the definition. Depending on

---

[7] P. Szeredi, *Module Concepts for Prolog*, Proceedings of the Workshop on Prolog Programming Environments, pp. 69-79, Linköping, Sweden, 1982

what construction we use when adding the clause, it is removed or not on backtracking. The same possibilities exist when we delete a clause. The clauses in a definition may be sorted after a particular argument and a metaconstruction can tell us how many clauses a certain definition has.

There is a larger set of constructions for reading, writing, and file handling in MProlog than in DECsystem-10 Prolog. In MProlog there is input from terminal and from file. The output may go to the terminal, to a file, or to a printer. When the reading fails or the predicate where the reading occurs, fails, the system backtracks in the input stream to re-read the previous characters or terms. We have several possibilities for controlling the output. If we use the printer, for example, we can specify where on the page we want to write, how the margins are to be, if we would like to start on a new page, etc.

There are constructions for string handling in the system, a feature that DECsystem-10 Prolog lacks. Among these may be mentioned

substring(S,N1,N2,X)

X is unified with a substring in S from position N1 to N2.

remove_blanks(S,X)

X is unified with string S without initial spaces.

concat(S1,S2,X)

X is the concatenation of strings S1 and S2.

raised(S1,S2)

S1 is a variable or a string. If S1 is a string S2 is unified to a string with corresponding upper case characters. If S1 is a variable S2 must be a string and S1 is unified with corresponding lower case characters.

string_length(S,L)

is the relation between a string S and its length L.

Various kinds of errors may arise during execution. The user can describe to the system what measures are to be taken depending on the nature of the error. Most errors are attended to by a predicate, a local error handler that replaces the predicate that caused the error. We can specify a predicate that takes care of all errors within a particular part of a program. This error handler handles the errors occurring in that part and which the local error handler cannot manage.

Editing Programs

In PDSS there are means for reading clauses easily from the terminal or from file, listing clauses, focusing a clause, editing the clause, executing programs, or parts of programs, tracing the execution of clauses, handling errors, and saving modules on file.

type P N_arg

lists clause P, or, if P is not specified, the clause in focus. N_arg is the arity of P. N_arg can be omitted if P has only one definition.

list

gives a list of all predicates that have been defined in the current module.

There is always a clause in focus. This clause can for instance be exchanged or deleted.

focus P N_arg

focuses the definition of the predicate P with arity N_arg. We can then move the focus forwards and backwards in the current definition.

delete P

deletes the whole definition of P. If P is not specified we delete the current clause, i.e. the clause in focus.

insert S

inserts the clause S in the current definition after the focused clause.

enter S

enters clause S last in the current definition. If S does not belong to the definition it is placed after the current definition.

replace S

replaces the current clause by clause S.

Should we want to edit a single line only, we have access to a line editor. The program clauses are assigned line numbers and we can refer to a line number for listing, deleting, or inserting the line. We also have the possibility of modifying a line number.

consult File

reads in the file File which may be an entire module or single clauses.

read File

treats the contents of File as commands, which means that the file may contain a whole module, commands to PDSS, or a mixture of both.

savemod File

writes the current module in File.

Trace

trace P

lets us trace the execution of the predicate P.

untrace P

stops the trace.

PDSS Control Commands

```
module M
```

makes module M the current module. If we omit the name we get the name of the current module.

```
body
```

reads clauses until the system finds **endmod** or **end**. The clauses are added to the module.

Execution Commands

```
execute P
```

starts the execution of P. After execution we get information on the CPU time we have used up.

```
solutions P
```

finds a solution to the predicate P and writes out the values the variables have been unified to and if we want we can request more solutions.

Help Commands

```
help PDSS-COMMAND NAME
```

gives information on PDSS Commands.

8.5 Turbo Prolog

Turbo Prolog[8] was developed at Borland International, Inc., Scotts Valley, California. This system is developed for the IBM PC and compatibles and can be run under the PC-DOS and MS-DOS operating systems. A Prolog system usually consists of an interpreter. Turbo Prolog, though, is a *typed* compiler. This means that we have to compile our programs and furthermore declare to the compiler the type of the arguments.

Syntax

The syntax for variables, constants, predicate names, and lists is the same as in DECsystem-10 Prolog. There is one difference, however, as Turbo Prolog can represent both integers and floating point numbers. The integers are in the range -32768 – 32767 and the floating point numbers in the range 1E-307 – 1E308 for both positive and negative numbers.

---

[8] *Turbo Prolog, The Natural Language of Artificial Intelligence*, Borland International Inc., Scotts Valley, USA, 1986.

Both ":-" and "if" can be used and they correspond to "← " in our Prolog system. There is also double notation for conjunction, i.e. "," and "and". Disjunction is denoted ";" and "or".

A difference from other Prolog systems is that infix operators may not be defined.

A Turbo Prolog program consists of a number of program sections.

domains	zero or more domain declarations
global domains	zero or more domain declarations
database	zero or more database predicate declarations
predicates	one or more predicate declarations
global predicates	zero or more predicate declarations
goal	goal
clauses	zero or more clauses

At least the predicates and clauses sections must be included in a program. The domains section contains domain declarations. We have to specify the domains to which objects in a relation may belong. Five formats are allowed.

char	character enclosed between two single quotation marks
integer	integers
real	floating point numbers
string	any sequence of characters written between a pair of double quotation marks
symbol	a sequence of letters, numbers and underscores, provided the first character is lower case or a character sequence surrounded by a pair of double quotation marks

Composite terms can also be defined in the domains section.

The sections global domains and global predicates are treated below in the modules section and the section on database is treated in connection with metalogic. predicates defines the predicates in the program by predicate names and arguments, the arguments we have typed in domains. If questions to the program use only one predicate, we can define goal. The Prolog program is placed after clauses.

Let us look at an example. In a relation likes(Name,X), the variable X may be various kinds of objects a person of the name Name likes, for instance, a particular book, a car, money. We can call these things objects. Furthermore, we want a relation person(Name,Sex,Age) for information on the sex and age of the person. In another relation we want to know the names of a person's children, children(Name,Children). Children is a list of names.

```
domains
        objects = book(title,author) ; car(make) ; money
        title, author, make, money, name, sex = symbol
        age = integer
        children = symbol*
```

```
predicates
        likes(name,objects)
        person(name,sex,age)
        children(name,children)
clauses
        likes(paul,dollar).
        likes(paul,car("Volvo")).
        likes(paul,book(The_Time_Machine, Wells)).

        person(paul,man,35).
        person(elizabeth,woman,23)

        children(paul,[anne,eric]).
        children(elizabeth,[ ])
```

";" in the domains section means "or". children is a list of zero or more elements of the symbol type, which is denoted by the asterisk, *.

Modules

Modules consists of definitions of predicates. A Prolog program can be divided into several modules. A module may be written, edited, and compiled separately. The modules are linked together to form an executable program. When a program has been divided into several modules the modules the program consists of should be specifed in a project. A special file must be generated for storing module names for the project.

All predicates in a module are local. If the modules in a program are to communicate with each other they have to be defined as global in the section global predicates. The domains that are used in the global predicates must be defined as global domains or be standard type domains. All modules in a project need to know exactly the same global predicate and global domains.

Logical predicates

The built-in construction = works both as predicate and operator. The result of a division is dependent on whether the arguments are integers or floating point numbers. Apart from the common arithmetic relations we have access to the most common trigonometric functions and mathematical functions such as absolute value and square root. They are represented as system functions.

An example of their use is

```
test(X_real,Answer_real):-
        Answer_real = ln(exp(sin(sqrt(X_real*X_real))))
```

Comparisons between both numbers and terms is made by the same built-in relations.

Comparisons between Numbers and Terms

X < Y	X is less than Y

X <= Y	X is less than or equal to Y
X > Y	X is greater than Y
X >= Y	X is greater than or equal to Y
X <> Y, X >< Y	X and Y are not equal

Metalogic

Turbo Prolog has built-in constructions for string handling. Some examples of these are

concat(S1,S2,S3)

that concatenates string S1 and S2 to string S3. At least two of the arguments must be ground.

frontstr(N,S1,S2,S3)

divides string S1 into two parts. String S2 is the first N characters in string S1 and S3 is the remainder.

str_len(S,L)

L is the number of characters string S consists of.

If we want to change a database during execution the predicates in the database must be declared dynamic in the section database. The database may be in the Prolog world or in a file. Updates are made by assert and retract in the same manner as in DECsystem-10 Prolog.

The construction in Turbo Prolog for finding all solutions is called

findall(X,P,L)

All substitutions on X which will validate the query P are gathered in the list L.

Communication

In most of the Prolog systems we have encountered so far, all file handling has dealt with sequential files. In Turbo Prolog, however, there are also random access files. A file can be opened for reading, writing, appending to the file, and for modification.

Several files may be open at the same time and we can control in the program what file is the current one. All files must have a symbolic file name and this is the name we use when referring to the file. Predefined symbolic file names are printer, screen, keyboard, and com1. com1 refers to the serial communication port. Symbolic file names must be declared in the section domains as file.

The following is an example of the file handling of the system. We want to open file Adress.one, with the symbolic name output, for printing. A name is read from the terminal. There is a relation person(Name,Address) in a database in the Prolog program. The address which belongs to the person we have read in is to be written in a specific position in the file.

```
domains
        file = output
        name, address = symbol
predicates
        updates
        person(name,address)
goal
        updating.
clauses
        updating:-
                write("Which name"),
                readln(Name),
                person(Name,Address),
                openmodify(output,"Address.one"),
                writedevice(output),
                filepos(output,100,0),
                write(Address),
                closefile(output)
        person(paul,"110 Park Lane, Boston")
```

readln(Name) unifies the variable Name with a line from the current input file, in this case the terminal. openmodify(output) opens the file with the symbolic file name output for modification and writedevice(output) states what file is the current output file. The construction filepos(output,100,0) states that Name shall be written in a file with the symbolic name output in position 100 reckoned from the beginning of the file. The third argument in filepos can specify that the file position shall be counted from the beginning of the file which is denoted by 0, as above. Another alternative is to have the position determined by the present position, denoted by 1. The final alternative is to have the position determined in relation to the end of file which is denoted by 2.

Trace

Before a program is executed in the Prolog system a syntax control is made of the program. If any errors are detected the control is transmitted to the editor and the cursor is placed where the error was detected. When a program is syntactically correct it is possible to follow the program execution step by step if we declare

```
trace(on)
```

The variables keep their original names in Turbo Prolog, i.e. we see no internal variable names during the interaction with the system.

Editing Programs

Turbo Prolog contains an editor. The Prolog system is menu-based and if we declare, for instance,

edit(In,Out)

we have access to the editor in the active window in the menu. In is the text we want to edit and Out the result after editing.

Compilation

Turbo Prolog is a compiling system. This means that the execution time for a program will usually be less than for the corresponding interpreted program.

Other Programming Languages

Turbo Prolog allows interfaces to other languages. The languages supported are Pascal, C, Fortran, and assembler.

8.6 micro-Prolog

micro-Prolog, also called LPA-Prolog, was developed at Imperial College, London, by K.L. Clark and F.G. McCabe[9]. micro-Prolog can be run on all computers with Z80 processor under the CP/M80 operating system and on 8088/86 machines under MS-DOS. It is also available for computers which have the UNIX operating system. The Prolog system consists of an interpreter in micro-Prolog and a number of extensions written in micro-Prolog. The system is implemented in assembler. Above all we will use a program called SIMPLE, which is implemented in micro-Prolog and uses the micro-Prolog system to answer our queries. SIMPLE is an extension to standard micro-Prolog aimed at making it more user-friendly.

Syntax

- Constants in the system begin with an upper-case letter.
- Variables are recognizable by always beginning with X, Y, Z, x, y, or z, and may be followed by a number.
- Numbers occur as integers in the range -32767 – 32767 and floating point numbers with at least 8 digits of accuracy and exponents in the range 10^{-127} – 10^{127}. Examples of how floating point numbers are written are 23.45, 2.345E2 ($2.345*10^2$=234.5) and -0.07E-5.
- Predicate names for predicates that we define ourselves consist of small letters which we can join by a hyphen.
- Lists are enclosed within brackets. The elements in a list are separated by spaces. ( ) is the empty list. (A B) is an example of a list of two elements. In (X1|X), X1 is the first element in the list and X is the rest of the list.

---

[9] K.L. Clark and F.G. McCabe, *micro-Prolog: Programming in Logic*, Prentice-Hall International, Englewood Cliffs, NJ, 1984

There are various ways of defining facts. Let us look at a few examples.

Anna mother-of Eve (177)
Anna woman (178)
(Anna Eric) parents-of (Eve Veronica Michael) (179)

Clause (177) is of the form "individual relation individual". In (178) the syntax is "individual property". In (179) it is "argumentlist relation argumentlist".

if corresponds to our "← ". and is the equivalent of our "," and either or to our ";".

Examples of clauses with conditions are

X grandmother-of Y if X mother-of Z and Z mother-of Y (180)
X parent-of Y if X father-of Y (181)
X parent-ofY if X mother-of Y (182)

The predicate parent-of can also be written in one clause:

X parent-of Y if (either X father-of Y or X mother-of Y) (183)

Clause (183) holds if X is father of Y or X is mother of Y.

There are several types of queries in SIMPLE.

is(p)

yields only the answer "yes" or "no", depending on whether micro-Prolog succeeded in proving clause p or not.

which(X : p)

gives all substitutions on X so that p succeeds. The query p can be a composite question and contain one or several variables. We get one answer a line. If we find which less appropriate from a linguistic point of view, we can use

all(X : p)

instead. which and all answer questions in the same manner. If we want only one answer we use

one(X : p)

After the system has supplied an answer it asks us if we are satisfied or would like to see an alternative answer. Thus, one can give all answers just like which and all.

Let us consider a few examples. We have the following statements:

Victoria plays Guitar
Jacob plays Soccer
Kate plays Piano
Kate plays Soccer

We ask if Victoria plays the guitar.

```
&.  is(Victoria plays Guitar)
yes
```

"&." is the command prompt from the Prolog system. The answer to the query is yes.

Let us ask if there is anyone who plays both the guitar and the piano.

```
&.   is(X plays Guitar and X plays Piano)
no
```

The answer is negative since X cannot be unified with any individual.

Which plays soccer?

```
&.   which(X : X plays Soccer)
Jacob
Kate
no (more) answers
```

We get the same answer if we ask

```
&.   all(X : X plays Soccer)
```

What does Kate play?

```
&.   one(X : Kate plays X)
Piano
more?  (y/n)  y
Soccer
more?  (y/n)  y
&.
```

Modules

Modules are a named set of relations. An import and an export list are linked to each module. In the import list are all names of relations which are defined outside our module and which we want to use within the module. In the export list shall be found names of all relations which will be able to be used outside the module. The module division of programs is handy if we have programs consisting of many relations and we fear that the same relational name might have been used more than once but with different meanings. The MODULES program in micro-Prolog enables us to use division into modules.

Logical Predicates

We unify two arguments by

X EQ Y

The relation holds if X and Y are identical or can be made identical by unification.

Two expressions can also be compared by =.

```
X = Y
```

X and Y can be numerical expressions. The difference between EQ and = is that when we use =, X and Y are each of them evaluated and then compared to EQ. If we have a variable on one side and a numerical expression on the other, the variable is unified with the value of the numerical expression.

A comparison between two numbers can be made by

```
X LESS Y
```

which holds if X is less than Y. LESS can also be used for constants and the relation holds if X precedes Y in standard order.

For addition and subtraction there is the built-in predicate

```
SUM(X Y Z)
```

which means that Z = X + Y. A query condition for the relation can have at most one argument unknown. The arithmetic relation is decided depending on what variable is unknown. If Z is unknown we get addition. If X is not ground the result is a subtraction between Z and Y. The same occurs when Y is not ground. If all three arguments have a value a comparison is made.

Should we want to multiply or divide, we use the relation

```
TIMES(X Y Z)
```

where Z = X * Y. Not more than one argument can be unknown. If either the first or second argument is unknown we get division; if the third is unknown we get multiplication.

Numerical expressions may contain the arithmetical relations * for multiplication, / for division, + for addition and - for subtraction. We can use = to unify X to a numerical value, i.e. X = (Y * 3 - Z / 2).

Metalogic

Our cut, |, in micro-Prolog corresponds to

```
/
```

and stands as a condition on its own.

"Negation as failure" is called

```
not p
```

and functions like Not in our Prolog system. The only difference is that p does not have to be an atomic clause. If p is composed of several atomic formulas they must be assembled within brackets.

There is a built-in construction that checks whether a number X is an integer.

```
X INT
```

(45 INT) is true. (4.67E2 INT) is also true since $4.67*10^2 = 467$. INT can also be used to separate the integer part from a floating point number. The first argument of the relation must have a value, but the second argument must not.

```
(X INT Y)
```

In (3.14 INT Y) Y is unified with 3.

We can check if a number of arguments satisfy a particular relation by

```
p true-of (X Y)
```

It holds if p succeeds with the arguments X and Y.

If we want to assemble all solutions to a question in a list, we evaluate

```
X isall(Y : p)
```

X is the list of all substitutions for Y that satisfy the relation p. The list is reversed, i.e. the first substitution that the systems makes for Y occurs last in the list.

There are also built-in constructions for database handling. We add a clause p by

```
add (p)
```

Then the clause will be placed last among the clauses in the current definition. It is possible to determine where in the definition the clause is placed by using

```
add n (p)
```

p will be the nth clause in the definition. It is also possible to use add in a question to save the answer as a clause in the program.

We remove a clause by

```
delete (p)
```

Should we want to delete the nth clause of a definition we specify this by

```
delete (p) n
```

And if we would like to delete the entire definition of a predicate we do it by

```
kill p
```

kill all will strip our whole Prolog world of information.

We will have listing of a specific relation p if we declare

```
list p
```

All the relations we have defined can be listed by list all.

Communication

We read the next term X from the terminal with

```
R(x)
```

A term X is written on the terminal using

```
P(X)
```

If we would like to shift lines after printing we use

```
PP(X)
```

Trace

If we read in the program SIMTRACE which is written in micro-Prolog it is possible to follow the execution of the queries is and all. We then use the queries

```
is-trace
all-trace
```

Editing Programs

We can edit our programs in the Prolog system. But as the editor is specific to different computers we will not discuss it here.

We save our Prolog world in interpreted code in the file FILE with

```
save FILE
```

To read in the Prolog world again we use

```
load FILE
```

An Example

Let us see what a program for quicksorting a list of numbers might look like in SIMPLE.

```
() quick-sort ()
(X) quick-sort (y1 | Y) if
        (Y1 Y2) difference (Y y1) and
        Z1 quick-sort Y1 and Z2 quick-sort Y2 and
        append(Z1 (y1 | Z2) X)
(() ()) difference (() y1)
((x1 | Y1) Y2) difference ((x1 | X) y1) if
        (either x1 LESS y1 or x1 EQ y1)
        and (Y1 Y2) difference (X y1)
(Y1 (x1 | Y2)) difference ((x1 | X) y1) if
        y1 LESS x1 and (Y1 Y2) difference (X y1)
append(() X X)
append((x1 | X) Y (x1 | Z)) if append(X Y Z)
```

8.7 LM-Prolog

LM-Prolog was developed by M. Carlsson and K.M. Kahn at UPMAIL, Uppsala University. LM-Prolog[10] is an implementation of Prolog on personal computers of the MIT Lisp Machine[11] type. The machine has been adapted to the desired programming environment and not the other way around. Lisp is used as system programming language throughout, with the exception of a micro-code kernel. The machines are personal work stations with large address spaces, large amounts of memory and disk storage, high resolution graphics, and high-speed network interfaces. The idea behind LM-Prolog is to provide the rich environments and unique features of the Lisp Machine to Prolog programmers.

The implementation consists of an interpreter, a compiler, and a program that translates from Decsystem-10 Prolog into LM-Prolog. The compiler optimizes relations and translates them into Lisp functions.

Syntax

The syntax is influenced by Lisp.

- Constants are called symbols in LM-Prolog. The symbols consist of lower and upper case letters and may contain "-".
- Variable names begin with "?" followed by upper or lower case letters. A single question mark denotes an anonymous variable that may occur only once in a given clause.
- Numbers include all kinds of numbers, i.e. integers, floating point numbers, and complex numbers.
- Predicate names consist of upper or lower case letters and may contain "-".
- Lists are enclosed in brackets. The elements in a list are separated by space characters. ( ) and nil is the empty list. The (A B) construction is an example of a list of two elements. The (A . B) construction denotes a list where A is the first element and B is the rest of the list.

The conjunction between a list of relations denoted by p is written as

```
(AND . p)
```

and disjunction is written as

```
(OR . p)
```

A query in LM-Prolog consists of a list where the first element in the list is the predicate name and the rest of the list contains the arguments to the

---

[10] M. Carlsson and K. Kahn, *LM-Prolog User Manual*, UPMAIL Technical Report No. 24, Uppsala University, Sweden, 1983

[11] D. Weinreb and D. Moon, *Lisp Machine Manual*, Massachusetts Institute of Technology, Cambridge, USA, 1981

question. If a clause consists of a conclusion and some conditions, the entire clause is a list consisting of the conclusion as the first element followed by the conditions.

```
((grandmother ?x ?y) (mother ?x ?z) (mother ?z ?y))
((mother Andrew Margaret))
```

A definition of a relation consists of an *option* part and the relation. The option part may contain information to the compiler on whether the arguments to the relation shall be ground or not when a question is asked. This declaration is voluntary but gives slightly more efficiently compiled code. Other declarations that may be given control, among other things, whether it is permissible to assert or retract the relation, which world it is to belong to, what will happen if there are synonymous definitions, whether the relation shall be compiled or interpreted, etc.

The option part of the following definition declares that the relation has exactly two arguments. The first argument must be ground when the relation is about to be evaluated and the second should not have a value.

```
(define-predicate grandmother
    (:options (:argument-list (+grandchild -name)))
    (grandmother ?x ?y) (mother ?x ?z) (mother ?z ?y)))
```

There are *multiple worlds* in LM-Prolog. A world or a database is denoted by a symbol and contains a large number of relations. A world may be *active* or *passive*. The active worlds form a list called universe.

Logical Predicates

x and y unifies with

```
(= x y)
```

The generalization to an arbitrary number of elements is called SAME.

Relations for the four rules of arithmetics are

(SUM result . numbers)	result is the sum of the numbers numbers
(DIFFERENCE result x y)	result is the difference between x and y
(PRODUCT result . numbers)	result is the product of the numbers numbers
(QUOTIENT result x y)	result is the quotient of x and y

The arguments to the arithmetical relations, except result, must be ground, which also goes for comparisons between numbers. Note that numbers denotes a variable number of elements. The same holds for comparisons between numbers.

(< . numbers)	The numbers numbers must be in strictly ascending order
(> . numbers)	The numbers numbers must be in strictly descending order
(≤ . numbers)	The numbers numbers must be in ascending order
(≥ . numbers)	The numbers numbers must be in descending order

LM-Prolog contains many built-in predicates. Among these may be mentioned

(APPEND list . lists)

which takes a variable number of arguments and succeeds if list is formed by appending all lists. Compare the Append program in Round 3.

(REVERSE rev list)

succeeds if rev is the reverse of list.

(LENGTH length list)

succeeds if list is a list which is length long.

(NTH nth n list)

unifies nth with the nth element in list.

(MEMBER element list)

is true if element is in list.

(REMOVE left element list)

is true if left is what is left of list after all occurrences of element have been deleted.

(INTERSECTION intersection . sets)

is the intersection of the sets sets represented as lists.

(UNION union . sets)

is the union of the sets sets represented as lists.

(SORT sort-list unsort-list P)

which succeeds if sort-list is a permutation of unsort-list and sort-list is ordered in the order P.

Truth is denoted

(TRUE)

Our False corresponds to

(FALSE)

Metalogic

What we call cut, |, in our Prolog system corresponds in LM-Prolog to

```
(CUT)
```

An alternative to using CUT is to declare to the system that we want only the first solution to a question by

```
(PROVE-ONCE p)
```

If we have a *deterministic* relation that may potentially give several solutions, but we know that we will never want more than one solution from it, the declaration

```
(:DETERMINISTIC :ALWAYS)
```

can be included in the option part of its definition. This is an alternative to PROVE-ONCE or to CUT after invocation of the definition.

For provability and unprovability there are the constructs

```
(CAN-PROVE p)
(CANNOT-PROVE p)
```

that evaluate whether p can be proved or not. No bindings are made.

The current values of variables can be tested by the constructs below.

(IDENTICAL . terms)	tests that terms are equal
(NUMBER term)	tests that term is a number
(VARIABLE term)	tests that term is not ground

The built-in construct Cases in our system corresponds to by

```
(CASES alt)
```

where alt consists of lists of questions. If we ask the question

```
(CASES (A B C) (D E))
```

A, B, and C are evaluated if A is provable. If A is not provable but D is, D and E are evaluated. If neither A nor D are provable, CASES fails.

LM-Prolog has metalevel facilities for asserting, retracting or listing clauses, adding and deleting worlds from the universe, assembling all solutions of questions in lists, testing the values of the variables, interpreting constructed terms, etc.

```
(ASSERT clause w)
(ASSERT clause)
```

adds a clause during execution to world w, or, alternatively, to the current world.

```
(ASSUME clause)
```

works like ASSERT but in addition it guarantees that clause is deleted upon backtracking past ASSUME.

(RETRACT clause w)
(RETRACT clause)

deletes a clause during execution.

(RETRACT-ALL p w)
(REMOVE-DEFINITION p w)

deletes all clauses from a definition by the name of p, that is, deletes the definition.

(ADD-WORLD w)

activates a world w.

(REMOVE-WORLD w)

makes a world w inactive.

(UNIVERSE universe)

gives the current universe.

(WITH-WORLD w . p)

evaluates questions p with world w temporarily added to the current universe.

(WITHOUT-WORLD w . p)

evaluates questions p with world w temporarily removed from the current universe.

(PREDICATOR p w)

finds out what predicate symbols p are defined in the world w.

If we want to print out a definition of a relation p we use

(PRINT-DEFINITION p)

If we want to print out a definition of a relation p in a particular world w we use

(PRINT-DEFINITION p w)

To assemble all solutions of a definition in a list we have

(BAG-OF x term . p)

which unifies x with all values of term to make the questions p come out true.

(SET-OF x term . p)

differs from BAG-OF only in that duplicates in the list are removed. All variables without values in question p that do not occur in term are assumed to be

existentially quantified and remain without value when BAG-OF or SET-OF terminate.

There is a means to declare all-quantified variables by

```
(QUANTIFIED-BAG-OF x allquant term . p)
(QUANTIFIED-SET-OF x allquant term . p)
```

which finds values of the variable allquant so that x will be the list of all values of term so that the questions p hold. Furthermore, in QUANTIFIED-SET-OF the solutions x is ordered in a normal order.

Communication

Among others, there are the following input and output predicates:

```
(LOAD file)
```

which loads a file, and

```
(OPEN-FILE stream file . op)
```

which opens the file stream as a stream to a file. op are the various options we can give as information to the system. The file is always closed when LM-Prolog backtracks past the call to OPEN-FILE. Such a stream can then be used as arguments in the input and output predicates that follow:

```
(TYI x stream)
```

reads an ASCII character x from the current stream stream.

```
(TYO x stream)
```

prints an ASCII character x in the current stream stream.

```
(READ term stream)
```

reads a term from the current stream stream.

```
(FORMAT stream string . terms)
```

prints a number of terms in the current stream, formatted according to codes given in string.

Trace

For debugging purposes, etc., we can ask to be shown each time a relation p is executed by specifying

```
(TRACE p)
```

This facility is switched off again by means of

```
(UNTRACE p)
```

Editing Programs

In LM-Prolog we use the Lisp machine editor. It is a very powerful editor which allows us to

- edit text
- edit structures; the editor checks, for instance, that parentheses match
- get an argument list and documentation for a relation if we specify the relational name
- get our relations printed in several lines so they become easy to read
- read in files
- compile and define relations
- compile files

Compilation

There is a compiler in LM-Prolog that optimizes relations and translates them into Lisp functions. These are translated by the ordinary Lisp compiler into an extended Lisp machine code. The extensions consist of new machine instructions made to support unification, among other things.

The compiler normally translates a question into invocation of relations. We can, however, tell the compiler to replace the question by a variant of the definition of the relation instead. This possibility is unique to LM-Prolog and is important if we want to define our own relations at metalevel.

For a relation to be compiled we can specify this in the option part of the definition of the relation or give a command in the editor that the relation shall be compiled.

Evaluation Order

The normal search order in Prolog is, given a question, to pick up those clauses whose consequence can match the question, and try them one by one. The LM-Prolog search order differs from standard in two respects.

First, LM-Prolog has multiple worlds that may be active or passive. Only active worlds are concerned when Prolog searches for matching clauses. The active worlds form a list called universe. The universe is searched world by world until we have a definition of a relation that matches the question and this relation is used.

In the implementation, however, such a search through the universe is only rarely made. Searching is avoided because each relational symbol remembers what definition was used last and what universe was then current. If the same universe is still current, the latest definition is used. Otherwise a search is made.

Secondly, we can declare that a relation shall be *indexed.* This means that we use some of its argument as a search key to get a smaller subset of those clauses that may match the question. For unindexed relations all clauses have to be tried.

Lazy and Eager Evaluation

In LM-Prolog there is a facility for *lazy evaluation* of the built-in constructions BAG-OF and SET-OF. Lazy evaluation means that a computation is not done until its value is asked for.

```
(LAZY-BAG-OF x term . p)
(LAZY-SET-OF x term . p)
```

The declarative semantics of these are exactly the same as for BAG-OF and SET-OF. The difference is when the computation of x occurs. Instead of doing the computation immediately LAZY-BAG-OF and LAZY-SET-OF postpone the computation until we need the value of x. This need may arise in either of two ways. First, if x is unified with a value, at least a partial result of x must be computed for us to know if the unification was successful or not. Second, if x will be sent to a Lisp function (e.g. to be printed), we will normally be interested in its whole value. In this second case there is another facility for the user to control the desired action.

Another facility in LM-Prolog is *eager evaluation* of BAG-OF and SET-OF. Again the difference lies in when the computation of x is carried out. It creates a separate process to compute the value of x in parallel and thus avoids having the rest of the program wait for the result of the computation. The need may arise for having part of the computation completed, and only then does the rest of the program wait. The need arises in the same way as for lazy evaluation.

Constraints

In LM-Prolog we can use constrained variables. This means that we impose a condition or *constraints* on what values a variable is permitted to assume. The condition is executed only when the variable has received a value. Another means of delaying the execution of a variable than lazy evaluation, is to link it to the value of a variable using

```
(CONSTRAIN variable p)
```

p is a question. Declaratively, this is equivalent to p. But the execution of p will be delayed until variable is ground. A limitation is that p cannot backtrack and give more than one solution.

Cyclic Structures

It is possible to use *cyclic structures* in LM-Prolog. The handling of them is user controlled. There are three different modes for handling cyclic structures.

- :prevent entails that unification can never create cyclic structures, i.e. none of the expressions in the unification may be a component of the other expression. This calls for a special occur check which will cause some programs to run twice as slowly.
- :ignore does away with the occur check. The unification may give rise to cyclic structures. It is possible to write programs so as not to make the

unification terminate. The Lisp interface, for instance, cannot handle cyclic structures. Most Prolog implementations neglect the occur check in the interests of efficiency.

- :handle does away with the occur check and thus the unification may generate cyclic structures. The unification is guaranteed to terminate, however. The Lisp interface and the rest of the system handles cyclic structures correctly. :handle makes the program execution slightly slower than :ignore and it is the default circularity mode.

Lisp Interface

If we want to use Lisp functions in our Prolog program we can do this very easily. Lisp functions are called in one of the following manners:

(LISP-VALUE term form)

lets Lisp evaluate form and the value is unified with term.

(LISP-PREDICATE form)

lets Lisp evaluate form. LISP-PREDICATE succeeds if the value is different from nil (the empty list).

(LISP-COMMAND form)

lets Lisp evaluate form but ignores the value.

Concurrent Prolog

There is an interpreter for another logic programming language, Concurrent Prolog[12] as a subsystem under LM-Prolog. We will describe this language very briefly.

Programs are built up by Horn clauses in Concurrent Prolog. The clauses have two kinds of control mechanisms. Clauses consist of three parts, conclusion, the *guard* control mechanism, and conditions. The second part, the guard, consists of a conjunction like the conditions. Instead of an evaluation of the clauses one by one and backtracking, all clauses are evaluated at the same time in Concurrent Prolog. The first clause whose conclusion matches and whose guard succeeds, wins. The other clauses are beaten and the conditions of the winner are executed.

The execution of a conjunction, i.e. a guard or a condition, is done in parallel, that is, all questions are evaluated at the same time. When we have parallel conditions with dependent common variables we sometimes need to synchronize them. For this synchronization there is a second control mechanism, *read-only-variables*. A variable A which is marked read-only, can always be unified with an unknown variable and it then makes this variable read-only, too. A can be unified with a non-variable only if A is ground. If A is not ground unification is delayed until some other condition has made the variable A ground.

---

[12] E. Shapiro, *A Subset of Concurrent Prolog and Its Interpreter*, ICOT Technical Report, TR-003, ICOT, Tokyo, 1983

An example

The following example of Quicksort is a specimen of LM-Prolog syntax:

```
(define-predicate quicksort
        ((quicksort () ()))
        ((quicksort (?u . ?x) ?y)
         (partition ?x ?u ?x1 ?x2)
         (quicksort ?x1 ?y1) (quicksort ?x2 ?y2)
         (append ?y ?y1 (?u . ?y2)))
(define-predicate partition
        ((partition () ? () ()))
        ((partition (?v . ?x) ?u (?v . ?x1) ?x2) (≤ ?v ?u)
         (partition ?x ?u ?x1 ?x2))
        ((partition (?v . ?x) ?u ?x1 (?v . ?x2)) (> ?v ?u)
         (partition ?x ?u ?x1 ?x2)))
```

The append relation is predefined in LM-Prolog.

Round 9. Sparringpartner

9.1 Simple Exercises

Problem 1:

Given the following definition of the relation Studies(x,y), where x is a student and y the studied subject,

Studies(Rhonda,CS) ←
Studies(Mark,CS) ←
Studies(Nathan,History) ←
Studies(Rhonda,Math) ←

formulate the following questions in Horn form:

a) Who studies history?

b) What does Rhonda study?

c) Does anyone take the same class as Mark?

d) What two students read the same subject?

e) Who studies?

f) Is there anyone taking two subjects at the same time?

Problem 2:

A kinship relation is given in Figure 9.1.

a) Define relations Parent and Grandchild and, using them, formulate the queries below.

   i) Who is parent of Harmonia?

   ii) What two persons are parents of Ares?

   iii) Who is a grandchild of Zeus?

   iv) Whom is Semele grandchild of?

   v) Who are parents of whom?

   vi) Is Zeus grandchild of Aphrodite or is Harmonia grandchild of Hera?

b) Describe the search space for the solutions bottom-up and top-down.

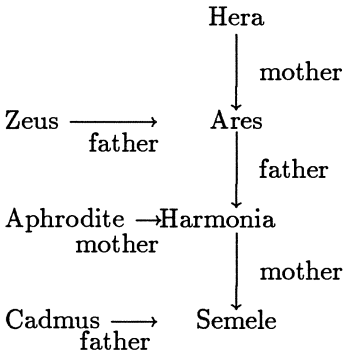

Figure 9.1: A kinship relation.

c) Define the relation Ancestor(x,y) in the sense that y is an individual from whom x is descended.

Problem 3:

We know that Adrian likes math, Deirdre likes logic while Christopher is fond of Deirdre and statistics.

We also know that Bill likes what Christopher and Adrian like, that Deirdre does not like herself but she cares for anyone who likes her, and that Adrian likes those who like math.

Express the above in Prolog and formulate the following queries:

a) Is there anyone who likes Adrian?

b) What does Deirdre like?

c) What is Bill fond of?

d) What two people like the same things?

e) What two people care for each other?

9.2 Structures

Problem 4:

Define a relation Greatest(Number1,Number2,GNumber) between three numbers in Horn form, when the numbers are expressed by the constructor S and GNumber consists of the greatest of the numbers Number1 and Number2.

Problem 5:

Define a relation that for a term and a binary structure with the constructor F, is true if the term is included as an argument somewhere in the structure.

```
← Occurs_in_F(B,F(A,F(B,C)))
yes
```

Lists

Problem 6:
Define a relation between a list and the number of occurrences of the element B in the list.

Problem 7:
Define a relation between a list containing numbers and the sum of the elements in the list.

Problem 8:
Define a relation between a list of numbers and the mean value of the elements in the list.

Problem 9:
Define a function from a list to another list in which all occurrences of a specific element have been removed.

```
← Delete_all(A.B.R.A.K.A.D.A.B.R.A.∅,A,list)
list = B.R.K.D.B.R.∅
```

Problem 10:
Define a relation between three lists where the third list shall be the intersection between the first two lists, i.e. all elements which are mutual to the first two lists.

Problem 11:
Define a relation between three lists where the third list shall be the union of the first two lists, i.e. all elements to be found in either or both of the first two lists.

Problem 12:
Defina a function from a list to another list containing all elements of the first list where all repeated elements have been removed.

```
← Set(A.B.B.A.∅,m)
m = A.B.∅
```

Problem 13:
Define a relation between a list and two sublists. All elements in the first sublist occur in the list before a certain "delimiter". All elements after this mark occur in the second sublist.
If, for instance, ; is given as the delimiter

```
← Divide(A.B.C.D.';'.E.F.∅,';',list1,list2)
list1 = A.B.C.D.∅
list2 = E.F.∅
← Divide(list,'#',1.2.3.∅,3.2.1.∅)
list = 1.2.3.'#'.3.2.1.∅
```

Problem 14:
Define a relation between two lists and a number n; the second list shall contain the first n elements from the first list.

Problem 15:
Define a relation between two lists where the second list shall contain all elements in the first list which start with a certain given combination of letters.

```
← Prolog(BC.ABC.A.ABDE.AB.∅,AB,list)
list = ABC.ABDE.AB.∅
```

Problem 16:
Define a function from a text to a compressed text in which all spaces have been removed.

```
← Compact('remove all spaces',text)
text = 'removeallspaces'
```

Problem 17:
Define the property Palindrome. A palindrome is a word or sentence that reads the same backward or forward, as, for example "Able was I ere I saw Elba".

```
← Palindrome(ROTOR)
yes

← Palindrome(PROLOG)
no
```

Problem 18:
Define a relation Less_all between a list and an element. The relation is satisfied when the element is less than all elements in the list. Assume that we will only try to derive ground formulas from the program.

```
← Less_all(3.2.100.-4.∅,2)
no
```

Problem 19:
Define a relation Greatest which shall hold between a list and an element if the element is the greatest element in the list.

Problem 20:
Define a function from a list to a triple of numbers. The first number in the triple shall be the number of positive numbers in the list, the second the number of negative numbers, and the third the number of zeros in the list.

Problem 21:

Define a relation between a number n, a word and a list. The list shall contain all words of length n which can be made up from the word and keep the internal order of the letters in the word.

```
← WordGen(3,FEAST,wordlist)
wordlist = FEA.FES.FET.FAS.FAT.FST.EAS.EAT.EST.AST.∅
```

Problem 22:

Define a relation Partition between three lists and a discriminating element. The second of the lists shall contain all elements in the first list that are less than the discriminating element, and the third list all that are greater than or equal to the element.

```
← Partition(7.4.5.9.2.3.0.6.1.∅,5,list1,list2)
list1 = 4.2.3.0.1.∅
list2 = 7.5.9.6.∅
```

Problem 23:

Define a relation between three sorted lists. The third list shall contain elements in the first and second lists and no more.

```
← Merge(1.5.6.10.∅,-1.3.4.7.∅,list)
list = -1.1.3.4.5.6.7.10.∅
```

Problem 24:

Define a relation between two lists. The second list shall be an ordered permutation of the first, i.e. a sorted list.

Problem 25:

Define a relation between two sorted lists and an element. The second list shall contain, apart from the elements in the first list, only the given element.

Problem 26:

Define a relation between a list of numbers and another list containing only those numbers before and including the greatest of the numbers of the list.

```
← Before_greatest(2.7.1.9.3.∅,2.7.1.9.∅)
yes
```

Problem 27:

Define a function Flatten(x,y) from the list x where the elements may be lists with an arbitrary depth to a list y with depth 1 which contains each element to be found in any level in x and in the same order as the elements in x.

```
← Flatten(((A.B.∅).∅).(C.((D.∅).∅).∅).E.∅,list)
list = A.B.C.D.E.∅
```

Trees

Problem 28:
Define a Prolog program that removes a node from an ordered binary tree producing a new ordered binary tree.

Problem 29:
Define a relation between two ordered binary trees and an element. The second tree shall only contain the given element apart from the elements in the first tree.

Problem 30:
Define a Prolog program that inserts a node in an ordered binary tree so that it will become the root of the tree.

Problem 31:
Define a Prolog program that selects all nodes between the root and a specific node in the tree, together with all nodes under the specific node.

Problem 32:
Define a relation between a tree and a list such that all elements in the tree are also in the list and vice versa.

Problem 33:
Define a relation Number_of_nodes(x,t) where x shall correspond to the number of nodes in the binary tree t.

Problem 34:
Define a relation Branch_depth between a number and a binary tree where the number denotes the depth of the tree.

Problem 35:
Define a Prolog program that investigates whether two nodes are connected, i.e. define a relation between a binary tree and two values.

Problem 36:
Define a relation between two binary trees where each subtree is the mirrorimage of the corresponding subtrees in the other tree.

Problem 37:
Define a program that counts those nodes in a binary tree which are not leaves.

Problem 38:
Define a program that replaces all occurrences of a given element in a binary tree by another. For instance, all occurrences of A are replaced by B.

Problem 39:
Define a program that produces a sorted list of all elements in a binary tree that begin with certain characters.

Problem 40:

Define a relation between a representation of a propositional formula, as for example, a tree (∧,(¬,a),b) representing the formula $\neg a \wedge b$ and

a) the number of connectives in the formula,

b) the number of atomic formulas,

c) the rank of the formula.

D-lists

Problem 41:

Define the property Palindrome for a difference list. A list is a palindrome if it reads the same backwards and forwards. For example, if D(list1,list2) is used to denote a d-list,

```
← Palindrome(D(E.V.E.∅,∅))
yes

← Palindrome(D(A.D.A.M.∅,∅))
no
```

Problem 42:

Define a relation between a tree and a difference list such that all elements in the tree are also in the list and vice versa.

Problem 43:

Define a predicate for printing propositional expressions. The propositional expressions can be represented in the form
({binary connective},{expr},{expr}) and
({unary connective},{expr}).
It should be possible to give the required line width as a parameter. If the line width is 25 the expression (↔,gloomy_weather,(or,rain,cold)) shall be written as

```
gloomy_weather ↔
rain or cold
```

instead of

```
gloomy_weather ↔ rain or
cold
```

Problem 44:

Define a program that makes a circular right hand shift of the elements in a list. A shift operation means that every character is moved one step to the right except the last character which will become the first. The number of shifts to be made shall be stated each time the program is executed.

```
← Rightshift(S.U.M.M.E.R.∅,1,word)
word = R.S.U.M.M.E.∅
```

9.3 Miscellaneous Exercises

Problem 45:
Define a relation between a number and the absolute value of the number.

Problem 46:
Define a relation between two numbers and the maximum value of the numbers.

Problem 47:
Define the factorial function.

Problem 48:
Define a program that writes the first ten numbers of Fibonacci series. The definition of Fibonacci numbers is to be found among the exercises in Round 2.

Problem 49:
Define a relation between two numbers and their sum. The relation shall hold between arbitrarily large numbers.

Problem 50:
Two predicates Life and Country are given. Life has three arguments, a person's name, the year of his birth and year of his death. Country has two arguments, the person's name and his country of activity.
Define the following predicates in Prolog:

a) Countryman(person1,person2) which holds if the two persons have been active in the same country

b) Age_at_death(person,age_at_death) which is the age a person had reached at his decease.

c) Oldest(person1,person2,oldest), where oldest shall be the name of that person who lived longest of person1 and person2

d) Active(person,year1,year2), where year1 denotes the beginning of a person's active period in life and year2 the end of it. Suppose that a person is active from the age of twenty until his death.

e) Contemporary(person1,person2) holds if the active years of two persons coincide wholly or in part.

Information about the span of life, time and country of various personages:

```
Life(Cervantes,1547,1616) ←      %Miguel de Cervantes
Life(Calderon,1600,1681) ←       %Pedro Calderon de la Barca
Life(Marlowe,1564,1593) ←        %Christopher Marlowe
Life(Shakespeare,1564,1616) ←    %William Shakespeare
Life(Jonson,1573,1637) ←         %Ben Jonson
Life(Corneille,1606,1684) ←      %Pierre Corneille
Life(Racine,1639,1699) ←         %Jean Racine
```

```
Life(Moliere,1622,1673) ←          %Moliere
Life(Monteverdi,1567,1643) ←       %Claudio Monteverdi
Life(Handel,1685,1759) ←           %George Frederick Handel
Life(Gay,1685,1732) ←              %John Gay
Life(Voltaire,1694,1778) ←         %Francois M A de Voltaire
Life(Beaumarchais,1732,1799) ←     %P A C de Beaumarchais
Life(Goethe,1749,1832) ←           %Johann Wolfgang Goethe
Life(Schiller,1759,1805) ←         %Friedrich Schiller
Life(Holberg,1684,1754) ←          %Ludvig Holberg

Country(Cervantes,Spain) ←         Country(Calderon,Spain) ←
Country(Marlowe,England) ←         Country(Shakespeare,England) ←
Country(Jonson,England) ←          Country(Corneille,France) ←
Country(Racine,France) ←           Country(Moliere,France) ←
Country(Monteverdi,Italy) ←        Country(Handel,England) ←
Country(Gay,England) ←             Country(Voltaire,France) ←
Country(Beaumarchais,France) ←     Country(Goethe,Germany)←
Country(Schiller,Germany) ←        Country(Holberg,Denmark) ←
```

Problem 51:

Two types of canoes are used at canoeing: kayaks and Canadians. The maximum length permitted for both types is 5 meters and 20 centimeters. Minimum width for kayaks is 51 centimeters and for Canadian canoes 75 centimeters. The minimum weight for a kayak is 12 kilos and for Canadians the minimum weight is 16 kilos. Define a relation between a length, a width, a weight and a classification. If the canoe does not meet the requirements, the classification will be E. If the length, width, and weight are valid for both types the classification shall be KC, but if they hold only for kayaks the classification shall be K.

Problem 52:

Three numbers represent the lengths of the sides of a triangle. The three sides can form a triangle if they are positive and always greater in pairs than the third side. Find out if three given values form the sides of a triangle. Investigate also whether the triangle is ordinary, equilateral or equiangular.

Problem 53:

According to the Julian calendar a solar year has 365 and 1/4 days. For the chronology to agree an extra day is added every fourth year. Such a year is called a leapyear. A further adjustment is necessary and is achieved by letting the first year of a century be a leapyear only if it is exactly divisible by 400. Define the property Leap_year.

Problem 54:

Use the built-in Prolog predicate Read to write a summation program. The program shall read positive numbers from the terminal and sum them. A number less than or equal to zero is the end mark.

Problem 55:

Given two sums of money (the sum to be paid and the sum actually handed over), expressed in dollars and cents, find the difference and declare in what denominations it shall be returned. The denominations are 100, 50, 10, and 5 dollars, 1 dollar, and 50, 25, 10, and 5 cents. The change shall always be given in as large denominations as possible, for example, 30 cents shall be returned as a quarter and a nickel, not as three dimes.

Problem 56:

Use the following little database and All_answers to answer the questions below.

```
Person('Abigail Adams',440416) ←
Person('Bill Blake',510818) ←
Person('Carrie Catt',560126) ←
Person('Doug Dillon',591212) ←
Person('Emily Eden',600710) ←

Phone(591212,123412) ←
Phone(600710,64201) ←
Phone(440416,76543) ←
Phone(510818,87903) ←

Membership(440416,'Student Club') ←
Membership(510818,'Student Club') ←
Membership(560126,'Student Club') ←
Membership(560126,'Athletic Club') ←
Membership(591212,'Athletic Club') ←
Membership(510818,'Athletic Club') ←
Membership(600710,'Athletic Club') ←
```

a) Compute the mean age of the members of the Student Club.

b) Write a list of the birth dates of the members of the Athletic Club.

c) Write a list of names of people who are members of both the Student Club and the Athletic Club.

Problem 57:

A publishing company stores information about its authors in trees where the root is the author's name and the nodes consist of lists. Each list has the title of a book as its first element. The following elements in the list are the years the book was published by the company. The tree is sorted on first year of publication (see Figure 9.2).

Write a program for the publishers that generates a list of all books of a particular author that were reprinted more than twice.

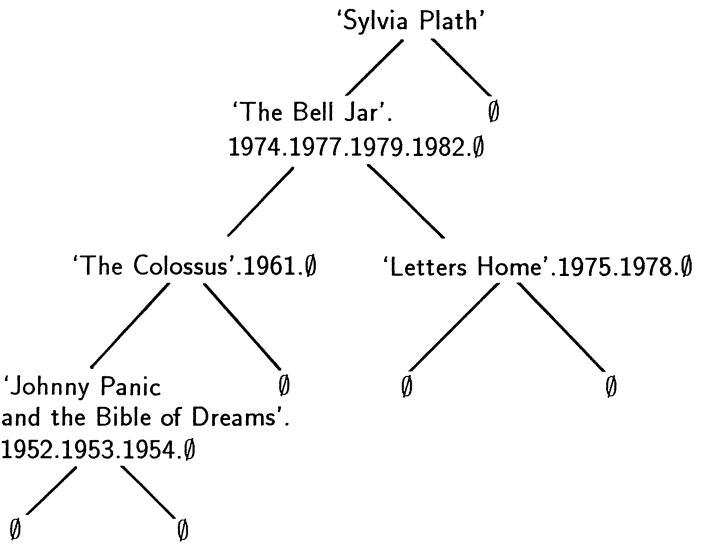

Figure 9.2: Author tree for Sylvia Plath

Problem 58:

Define a relation between a sentence written in small letters and another sentence in capital letters only.

```
← Letters('the sun shines and the sky is blue',capitals)
capitals = 'THE SUN SHINES AND THE SKY IS BLUE'
← Letters('IT IS SNOWING OUTSIDE',small)
small = it is snowing outside
```

Problem 59:

A routine for computing the number of nights between two given dates is needed for a hotel room booking system. Write a program that checks that dates are valid and performs the above routine.

Problem 60:

Write a program that computes the parking fee to be paid, given the time when parking began and the current time. The parking fee is 30 cents an hour. The fee shall be stated in round cents. All parking is assumed to begin and end within the same twenty-four-hour period.

Problem 61:

Write a program that reads a three-digit integer and writes the value of the number in English.

```
← Number_text(182,text)
text = ONEHUNDREDANDEIGHTYTWO
```

Problem 62:

A date consists of six digits: the first two digits are the month, and the following two are the day of the month, and the last are the year. The digits for month, day and year shall be separated by one or several spaces. For example, '10 10 84'. Change the format of the date into year, the day and the name of the month.

```
← Date('10 10 84',d)
d = 1984, 10 OCTOBER
```

Problem 63:

Write a program that solves the n-queens problem when $n = 4$. Four queens shall be placed on a chess board with 4*4 squares in such a way that no queen can attack another. Queens may attack horizontally, vertically, and diagonally according to the rules of chess.

Problem 64:

Solve the Towers-of-Hanoi problem in Prolog. Given are three pegs A, B, and C and three discs of different sizes. The discs have holes in the middle so they can be placed over the pegs. At the start all discs shall be placed on the first peg A with the largest disc at the bottom and the smallest disc at the top.

The problem consists in moving all discs from peg A to peg C. You are only allowed to move one disc at a time and only the topmost disc on a peg may be moved. A disc must never be placed on a smaller disc. The problem can be solved with more than three discs, of course.

Problem 65:

Three missionaries and three cannibals shall cross a river. A boat is available and holds two people. The boat can be navigated by one or two persons and it does not matter if it is done by missionaries or cannibals. If the cannibals should happen to outnumber the missionaries on any occasion, the cannibals will succumb to their habits and eat the missionaries. Try to find the simplest way for all six persons to cross the river safe and sound.

Problem 66:

At a Monte Carlo rally there were three starting places: London, Paris, and Lisbon.

i) From London John set off in his BMW, Tommy in his Ford, Fred in a BMW, Anne in a Ford, and Teddy in a BMC.

ii) From Paris Guy started in a Fiat, Claude in a Ford, Jean in a Fiat, Brigitte in a BMC.

iii) In Lisbon there were Luis in a Fiat, Carlos in a BMW, Lucas in a BMC, and Pedro in a Ford, and Nuno in a Fiat.

iv) Only four drivers arrived at Monte Carlo, Tommy, Fred, Pedro, and Nuno.

Write a program that answers the following questions:

a) Which cars arrived at Monte Carlo?

b) From where had those drivers set off who eventually reached the goal?

Problem 67:

Construct a simple proof checker that contains the following inference rules:

($\wedge$I)

$$\frac{p \quad q}{p \wedge q}$$

($\wedge$E)

$$\frac{p \wedge q}{p} \qquad \frac{p \wedge q}{q}$$

($\rightarrow$I)

$$\frac{\begin{array}{c}(p)\\ q\end{array}}{p \rightarrow q}$$

($\rightarrow$E)

$$\frac{p \quad p \rightarrow q}{q}$$

($\bot$)

$$\frac{\bot}{p}$$

(RAA)

$$\frac{\begin{array}{c}\neg p\\ \bot\end{array}}{p}$$

Decide yourself how the rules shall be declared to the proof checker.

Problem 68:

Construct a deduction system that, for a set of known facts about animals in a zoo, is able to draw new conclusions from this information.
The deduction system shall contain the following fifteen rules for the recognition of an animal:

1. If the animal has a pelt then it is a mammal.
2. If the animal suckles then it is a mammal.
3. If the animal has feathers then it is a bird.
4. If the animal flies and lays eggs then it is a bird.

5. If the animal is a mammal and eats meat **then** it is a beast of prey.
6. If the animal is a mammal, has sharp teeth, claws, and eyes directed forwards, **then** it is beast of prey.
7. If the animal is a mammal and has hoofs, **then** it is a hoofed animal.
8. If the animal is a mammal and it ruminates, and it has an even number of toes, **then** it is a hoofed animal.
9. If the animal is a beast of prey, tawny with dark spots, **then** it is a cheetah.
10. If the animal is a beast of prey, tawny with black stripes, **then** it is a tiger.
11. If the animal is a hoofed animal, has a long neck, long legs, is tawny with dark spots, **then** it is a giraffe.
12. If the animal is a hoofed animal, white with black stripes, **then** it is a zebra.
13. If the animal is a bird, does not fly, has long legs and a long neck, and is white and black, **then** it is an ostrich.
14. If the animal is a bird, does not fly, but swims, is black and white, **then** it is a penguin.
15. If the animal is a bird and an excellent flyer, **then** it is an albatross.

Using the above rules the deduction system should be able to classify an animal by asking for the distinguishing features of the specific animal. It should also be able to describe the properties of the specified animals.

Problem 69:

Write a relation that gives the computer a word to think of and lets the user try to guess the word. If the user gives the same word, the text "Correct guess!" should appear, or else the program will say how many letters agree in the two words.

Extend the program to tell also what letters are placed in the corresponding locations.

For instance, `SUMMER` and `WINTER` each have a letter placed in the corresponding place (`R`). `SUMMER` and `SUNSHINE` have two letters in corresponding locations.

Problem 70:

The Albatross boat company manufactures pleasure boats made to order. A buyer may choose her particular boat by specifying the desired length, width, number of berths, and motor strength.

The length may vary between 4 and 24 meters.

The width may vary between 1 and 8 meters.

The number of berths may vary between 0 and 14.

Possible values for motor power are 10, 50, 100, 500, or 1000 horsepower.
If the prospective buyer specifies some parameter not included in the value areas given above or requires a forbidden value (a 20-horsepower-engine, for instance) she will be told so.
Some combinations result in an unusable boat and certain restrictions must therefore be introduced. In particular, the length of the boat must not be less than 3 meters and not more than five times the width of the boat. The horsepower cannot exceed 6 times the product of the length and width of the boat. The number of berths cannot be more than 10.
Should any of these conditions not be satisfied the buyer will receive one of the following messages:

```
MODEL TOO WIDE IN RELATION TO DESIRED LENGTH
MODEL TOO NARROW IN RELATION TO DESIRED LENGTH
ENGINE POWER OVERDIMENSIONED
TOO MANY BERTHS IN RELATION TO SIZE OF BOAT
```

Write a program that the Albatross boat company can use for checking their customers' demands and which, moreover, can compute the costs of the model provided that the model can be manufactured at all.
The cost of an Albatross boat is \$150 multiplied by the square of the length of the boat, plus \$750 for each berth plus \$ 60 per horsepower.

Problem 71:
Make an addition to the exercise above so that the length may vary between 4.5 and 24 meters and assume all values which are exact half-meters inbetween. Furthermore, the width may vary between 1.5 and 8 meters and shall also be specified half-meter.

Problem 72:
Among 8 coins there is one counterfeit. The counterfeit coin has a different weight from the others. Your task is to determine which coin is faked by weighing the coins on a balance. As few weighings as possible should be made; it is possible to decide with only three weighings. Write a relation that can be used for deciding which coin is counterfeited.

Problem 73:
This and the next exercise deal with examples of fiscal niceties in Sweden. Define a relation Property_tax with suitable arguments. "Taxable property" is equal to that part of a person's property that is liable to taxation, with the sum rounded off to the next lower whole number of 100 Sw. kronor. The property tax can be calculated from the table in Figure 9.3.

Problem 74:
Write a program to calculate what basic tax deduction the user is entitled to on taxation.

Taxable property	Tax at the lowest limit	Tax on that part of the property which is above the lowest limit
200 000-274 900	0	1 %
275 000-399 900	750	1.5 %
400 000-999 900	2 625	2 %
1 000 000-	14 625	2.5 %

Figure 9.3: Table for calculating property tax.

The basic deduction is calculated for the period of the fiscal year during which a person has been residing in Sweden and is 375 Sw.kr a month or part of a month. If one has been residing in Sweden during the whole year the basic deduction will be 4500 Sw.kr.

Problem 75:

Write a program that produces a list of random numbers with a variable amount of numbers. How many numbers the list shall contain shall be stated in the relation. Random numbers are chosen in the interval 1 to 100 without repetition

The program shall check whether or not the amount of numbers is reasonable and if not write 'Only 1 – 100' and demand a new input.

Problem 76:

Write a program that reads an English text comprising one or several lines. The lines shall be written the way they looked when fed in with the exception that the following changes shall be made:

'he'	is exchanged for	'he/she'
'she'		same as above
'him'		'him/her'
'her'		same as above
'his'		'his/hers'
'hers'		same as above
'man'		'person'
'woman'		'person'
'daughter'		'child'
'son'		'child'

The exchange shall be effected also when 'man', 'woman','daughter', or 'son' are part of a word. 'Chairperson' shall thus be exchanged for 'chairman'.

Problem 77:

Read a text with lines of different length and reorder them into lines of 50 characters, for instance, and with straight right-hand and left-hand margins.

Problem 78:

Sweden is a parliamentary democracy with five political parties seated in the "Riksdag", the Swedish Parliament. The parties are the Conservatives (M), the Liberals (FP), the Agrarians (CP), the Social Democrats (SAP), and the Communists (VPK). The abbreviations are established usage and are the initials of the full party name in Swedish.

Seats are distributed according to a modified version of the Sainte-Laguë system of successive division by odd integers.

9 fixed seats are to be distributed in a constituency and the distribution shall be done according to the following rules:

The polls of the different parties are first divided by 1.4, after which operation a comparative number is obtained. The seats are then distributed one by one, always to that party which has the highest comparative number at the moment. After a certain party has received a seat, a new comparative number is computed for that party by dividing the original poll of the party by 3. If the party gets another seat the next comparative number for that party shall be computed by dividing the party's original poll by 5. The comparative numbers of a party are thus to be had by first dividing the total poll by 1.4 and then by the odd numbers 3, 5, 7, 9, etc.

Construct an algorithm and write a program that distributes the seats among the different parties according to the method described.

Problem 79:

A magic square of order n is a layout of the numbers 1, 2, ..., n^2 in a quadratic table so that the sum of each line, column, and diagonal is equal. Write a program that generates and writes out magic squares.

Problem 80:

Solve 'Josephus's problem' which consists in deciding who will stay alive the longest in the following macabre game.

n persons forms a ring. Starting from a given position, every mth person is executed. The circle closes afterwards. If, for example, the number of persons n is eight and we execute every fourth person, the execution order will be 4 8 5 2 1 3 7 6, i.e. the fourth person from the starting point will be the one to be executed first. Write a program that writes out the execution order for an arbitrary n and m.

Problem 81:

This is a variant of "Game of Life" (see "Mathematical Games", *Scientific American*, October 1979). "Game of Life" is a "process of birth and death" – a model that describes rules for how individuals in a population are born, live, and die. These models can be used in biology, nuclear physics, etc. This particular game has simple rules but will still lead to interesting results.

Imagine a two-dimensional field divided into squares like a chess board. At a particular moment a square is either empty or inhabited by a single

individual in the population. The state of the field at a specific moment is called a generation.
The individuals in a generation, R, are born, die, and survive into the next generation, S, according to the following rules:

1. If a square is inhabited in R and the individual has 2 or 3 neighbors, the square will be populated also in generation S. (A neighbor is an individual who lives in any of the eight adjacent squares).
2. If a square is inhabited in R and the individual has either less than 2, or more than 3 neighbors, the individual will die (from loneliness or overpopulation) and the square will be empty in S.
3. If a square is empty in R and there are exactly 3 neighbors, a new individual will be born in the square of generation S.

Write a program that creates a starting generation consisting of M individuals from given input values, writes out the starting generation, and generates and writes out the succeeding generations N of the population.
The field is a quadratic island, 14 squares long and 14 squares wide. The outer edges of all 4 sides represent a beach where nobody can live, i.e. they are always empty.
The arguments given to the program shall be the number of generations to be produced as well as the number of individuals in the starting generation and their locations.
Clue: If you let your solution consist of the following parts which are not given in order, your task will be easier.

1. Generate a new generation S from generation R.
2. Initialize a generation to be empty.
3. Create a generation from the positions.
4. Write out one generation.

Problem 82:

Given one generation, see the preceding problem, demonstrate what appearance the generation before must have, according to the rules above.

Problem 83:

Write a program that will produce a picture according to Figure 9.4.
The frame shall consist of B's and D's as in Figure 9.4; the size is for you to decide. Then imagine that a turtle is walking about leaving a trace behind. The turtle may move in 8 different directions. The turtle leaves a letter as trace in every empty square it comes to. What letter it leaves behind depends on what direction it moves in, see Figure 9.5.
When the turtle arrives in a square where there is already a letter, it will change its direction. The new direction is dependent on the present direc-

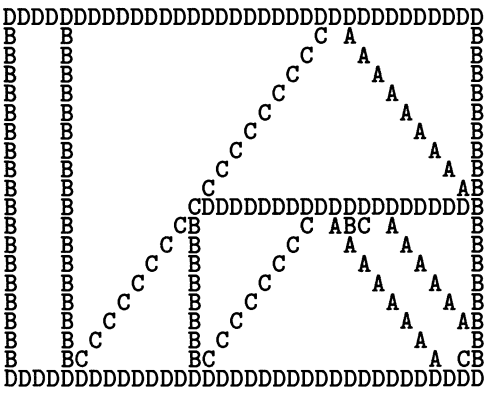

Figure 9.4: Turtle walk

Direction	Number	Letter
↑	1	B
↗	2	C
→	3	D
↘	4	A
↓	5	B
↙	6	C
←	7	D
↖	8	A

Figure 9.5: Table of directions and letter traces.

	1	2	3	4	5	6	7	8
A	7	7	5	0	2	1	2	0
B	0	8	6	7	0	3	4	3
C	8	0	1	7	8	0	5	3
D	6	4	0	2	2	1	0	6

Figure 9.6: Table to determine the turtles new direction.

tion, on one hand, and on the other hand, on what letter is to be found in the new square the turtle arrives in.

Construct a table which will help you to determine the new direction. It may look like the table in Figure 9.6, for instance. If the turtle moves in direction 2 and meets a square with a B the new direction will be 8.

The turtle moves in the new direction starting from the square it has just arrived in (look closely at the example in figure 9.4.). If the first square the turtle comes to contains a letter, the walk is over and the program shall write out the result.

To begin the turtle's walk you should read in its position and direction at start. Do not mark the starting square.
In the example figure 9.4 the starting line was 1, the starting column 5, and the starting direction was 5.

Problem 84:

Write a program that handles a list of names. The names are in the form: given name family name, where the given names and the family names shall be separated by one or several spaces. About 30% of the names occur more than once in the list.
The program shall read the names and write them out in alphabetical order by given name or family name, depending on your choice.

Problem 85:

Fanny believes that her life would be a lot easier if she could store her recipe collection on a computer.
Each recipe is to be stored under the name of the dish. Ingredients, cooking time and preparation method shall be stored for every dish. For each ingredient shall be stated an amount, i.e. 1 tablespoon, or 100 kilos (for a particular ingredient the amount shall always be given in the same unit of measure).
To support her recipe collection Fanny needs a number of programs.

a) A program for feeding in new recipes.

b) A program that takes out all information on the dish she wants to cook.

c) A program that writes a shopping list for her if she states what dish she wants to cook. If the same ingredient is to be used in more than one dish, the whole quantity needed shall be on the shopping list.

d) A program that suggests to her what dishes she can prepare from the ingredients she has available.

e) A program that informs her what dishes can be cooked in a certain time.

Problem 86:

Write a relation between two numbers where one number is represented by arabic figures and the other by roman numerals.
The roman numerals are:

roman	arabic correspondence
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

There are never more that 3 occurrences of a numeral in a row. The number 4 is not written IIII but IV (can be read as 5-1). That is, a lesser figure is put after a greater on addition but before on subtraction.

I	1	XXXV	35
II	2	XXXVI	36
III	3	XXXVII	37
IV	4	XXXVIII	38
V	5	XXXIX	39
VI	6	XL	40
VII	7		
VIII	8	MCMLXXX	1980
IX	9		
X	10		
XI	11		
XII	12		
XIII	13		
XIV	14		
XV	15		
XVI	16		

Problem 87:

Write a program that transforms roman numerals to arabic numerals.

Problem 88:

Information on telephone subscribers is stored in a file. Each subscriber has a subscription number and information on name and address.

a) Write a program that can be used for updating the database on telephone subscribers. Before the update the program shall inquire which number the update concerns and then ask for the subscriber entries name and address.

b) Write a program that changes the information for a given number or removes all information on the subscriber if the reply to the question for name and address is only a press on the RETURN key. Update can occur only for subscribers to be found in the database.

c) Change the program above so that new subscribers can be entered. That is, if a subscription number not included in the database is written, a new subscriber will be entered.

9.4 Games

Problem 89:

Write an interactive program that plays a simplified version of roulette. The player (the user at the terminal) plays against the house (the program).

The game consists of a sequence of rounds till the player ends the game or is forced to stop (because he has run out of money and is not granted credit). When the game begins the player owns 10 counters. In every round the player may stake on either ROUGE or NOIR. The smallest stake is 1 counter. The stake must not exceed the number of remaining counters the player has in his possession. The probability for NOIR to come up is 0.45, and for ROUGE 0.45 while for 00 the probability is 0.1. Stakes on NOIR or ROUGE will return the double. When 00 comes up all counters revert to the house.

For each round the program shall report how many counters the player has left, receive his stake and then inform him of the result. The program shall ask for the user's stakes (preferably in French but English will do). Appropriate comments (congratulations or regrets depending on the outcome of the game) shall be written before the game is finished.

Problem 90:

Define a program for the so-called game of Nim or the Number War, according to the following description.

Two players and three piles of counters. Each player in turn removes a number of counters from any of the piles. The number of counters which can be removed is unlimited but a player must only take from one pile at a time. The player who takes the last counter wins.

The number of possible moves is very large in every situation. There is a mathematical formula, however, using which an optimal move can be made (a move that will lead to victory whatever the opponent may do). Because of the very large number of possible combinations of moves it is difficult for one who does not know the optimal game strategy to win against a player who knows this strategy.

The game strategy aims at representing the number of counters in every pile as a binary number. If we have piles containing 5, 4, and 3 counters we get

	Decimal	Binary
Pile 1	5	101
Pile 2	4	100
Pile 3	3	011

We will look at the parity of the columns, i.e. whether the number of ones in each column is odd or even. The first column contains 2 ones and thus has even parity. The middle column has odd parity because it contains only 1 one. The third column has even parity. The optimal strategy consists in removing counters from a pile so that the parity in each column will remain even. If we take away 2 counters from the third pile we get

	Decimal	Binary
Pile 1	5	101
Pile 2	4	100
Pile 3	1	001

The player who takes away the last counter and wins the game, leaves all columns with even parity (the number of ones in each column is zero).
When one or several columns have odd parity it is always possible to make a move to give all columns even parity and when all columns have even parity each move will make one or more columns get odd parity. Hence a player can be sure of winning if he has achieved a position when all columns have even parity and he makes only optimal moves after this point. There may be several optimal moves. If we have 7, 6, and 5 counters in the piles, removing 4 will give even parity no matter which pile the 4 are taken from.
The method for deciding on an optimal move is to find a line that has a one in the first column with odd parity. Counters are taken from that line so that each figure in an odd parity column is changed from 0 to 1 or from 1 to 0.
The task consists in writing a program that plays the number war according to the optimal game strategy against a player who sits at the terminal. The player at the terminal shall always make the first move and the number of counters in each pile shall be randomized before the game begins. The program shall make appropriate output, check the opponent's moves and give the user of the program sufficient information to be able to play.

Hint:
Algorithm for computing the optimal move:

optimal_move	denotes the number of counters that should be removed in an optimal move
x	x denotes the power of 2 that adjusts the optimal move

OM1. Find the first column from the left with odd parity.

OM2. Choose a line with a one in the odd column.

OM3. Initialize optimal_move to the decimal number equal to the binary number which has a one in the current odd column and zeros in the other columns.

OM4. Until all columns have been checked repeat:

OM41. Take the next column.

OM42. If the column has odd parity:

OM421. Find the number x that in the binary representation has a one in the column and zeros in the other columns.

OM422. If the line has a zero in the column, subtract x from optimal_move, otherwise add x to optimal_move.

OM5. Subtract optimal_move number of counters from the pile which corresponds to the line.

Problem 91:

This task consists in writing a program that plays the game of Kalah against one opponent, according to the following rules.

Rules for Kalah.

The game of Kalah that has fascinated the world for 7000 years is one of the oldest games in the world. Its history can be traced back to around 5000 BC. The 14 holes in the Kalah board have been found in Egyptian temples. The game is depicted on grave paintings in the Nile valley. Holes are dug out in rocks along the old caravan trails.

The maharajas in ancient India played Kalah with diamonds and rubies instead of stones. The game was popular in Egyptian coffee houses where the loser had to pay the check for the winner. African tribal chiefs amused themselves by playing their own version of the game using young slave girls as counters.

Kalah is a strategic game and the aim is to win as many stones as possible. The game is as entertaining in our time as it has been for past millenia.

Terms:

to sow - to put a stone in each hole

to pick - to pick up all the stones from a hole and then sow them

move - one or more pickings

kalah - a larger hole for keeping won stones

ambos - the six small holes on each side of the board, facing each other

The initial set-up and the basic moves:

Kalah is played by two persons. Each player has a row of six small round holes in front of him with six stones in each hole. Children and beginners should start with only three stones in each ambo until they are clever enough to play with six stones.

On his right hand, in front of the small holes, is the player's kalah, a large empty hole. The idea of the game is to collect as many stones as possible in one's own kalah. The players take turns in making moves and draw lots for the first move.

How to play:

A player begins by picking up all the stones form one of his ambos. He places one stone in each hole, starting with the next hole to the right and proceeding counterclockwise around the board until there are no stones left to be distributed. If the last stone lands in one's own kalah, one is allowed another move.

Sometimes the last stone does not land in one's own ambo or kalah. Then the player may cross the board and distribute his stones in the opponent's holes, as well. But a player may never start his move on the opponent's side. All the stones of an ambo are always moved at the same time and the game proceeds counterclockwise.

A player may not sow his own stones in the opponent's kalah. If he has to proceed to the opponent's side of the board he sows in the opponent's ambos, skipping the opponent's kalah and continues over to his own side. Sometimes the last stone of a move is sown in an empty ambo on the player's own side of the board. The player is then allowed to pick that stone as well as all the stones of the opponent which are in the hole directly across the board. All these stones are put in the player's kalah. The player is not allowed another move, however; it is now the opponent's turn. (If the last stone lands in an empty ambo on the opponent's side, the player is not allowed to perform this trick.)

The game is over when all the ambos of a player are empty. Remaining stones are put in the opponent's kalah and are included in his final score. Hence, a player should take care not to empty his side of stones as long as the opponent still has several stones left on his side.

The player with the largest number of stones in his kalah has won the game.

Appendix A. Answers to Exercises

A.1 Round 1

Exercise 1:

$\forall x \forall y(Playmate(x, y) \leftrightarrow \exists z(Lives(x, z) \wedge Lives(y, z)))$

```
Playmate(x,y) ←
        Lives(x,z), Lives(y,z)
```

Exercise 2:

See Round 3.

Exercise 3:

$\forall x \forall w(Part(x, w) \leftrightarrow x = w \vee \exists y \exists z(w = Or(y, z) \wedge (x = y \vee Part(x, z))))$

```
Part(x,x) ←
Part(x,Or(y,z)) ←  x = y
Part(x,Or(y,z)) ←
        Part(x,z)
```

Exercise 4:

```
Game(x) ← Big_game(x)
Game(x) ← Small_game(x)

Big_game(x) ←  Moose(x)
Big_game(x) ←  Deer(x)
Big_game(x) ←  Wild_boar(x)
Big_game(x) ←  Lion(x)

Small_game(x) ←  Fox(x)
Small_game(x) ←  Rabbit(x)
Small_game(x) ←  Bird(x)
```

Exercise 5:

```
Judo(x,Lightweight) ←  x ≤ 63
Judo(x,Welterweight) ←  x > 63, x ≤ 70
```

Judo(x,Middleweight) ← x > 70, x ≤ 80
Judo(x,'Light heavyweight') ← x > 80, x ≤ 93
Judo(x,Heavyweight) ← x > 93

A.2 Round 2

Exercise 1:

- {⟨x,z⟩,⟨y,Frey⟩}
- {⟨x,u⟩,⟨w,Or(y,z)⟩}
- not possible to unify since x occurs in S(x)

Exercise 2:

```
Fibonacci(1,0) ←
Fibonacci(2,1) ←
Fibonacci(n0,t0) ← n0 > 2,
        Value(n0-1,n1),
        Value(n0-2,n2),
        Fibonacci(n1,t1),
        Fibonacci(n2,t2),
        Value(t1+t2,t0)
```

Bottom-up search space

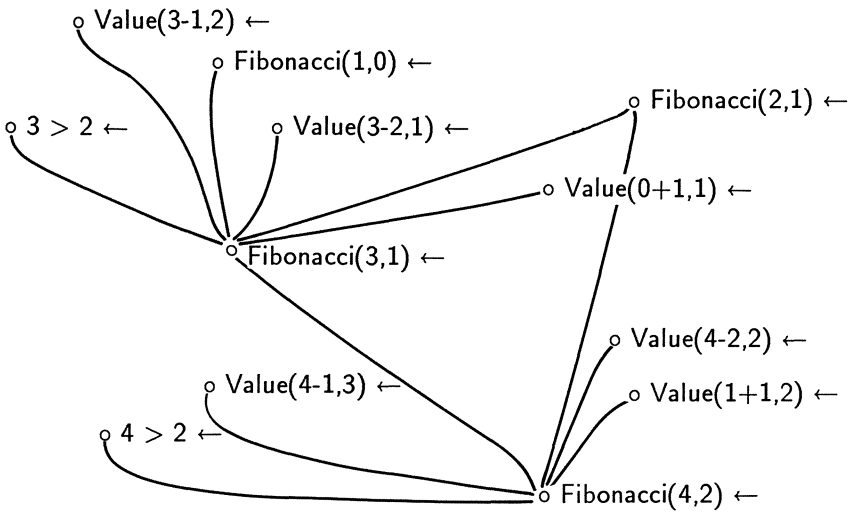

Figure A.1. Bottom-up search space for Fibonaccid(4,2)

Top-down search space for Fibonacci(4,2)

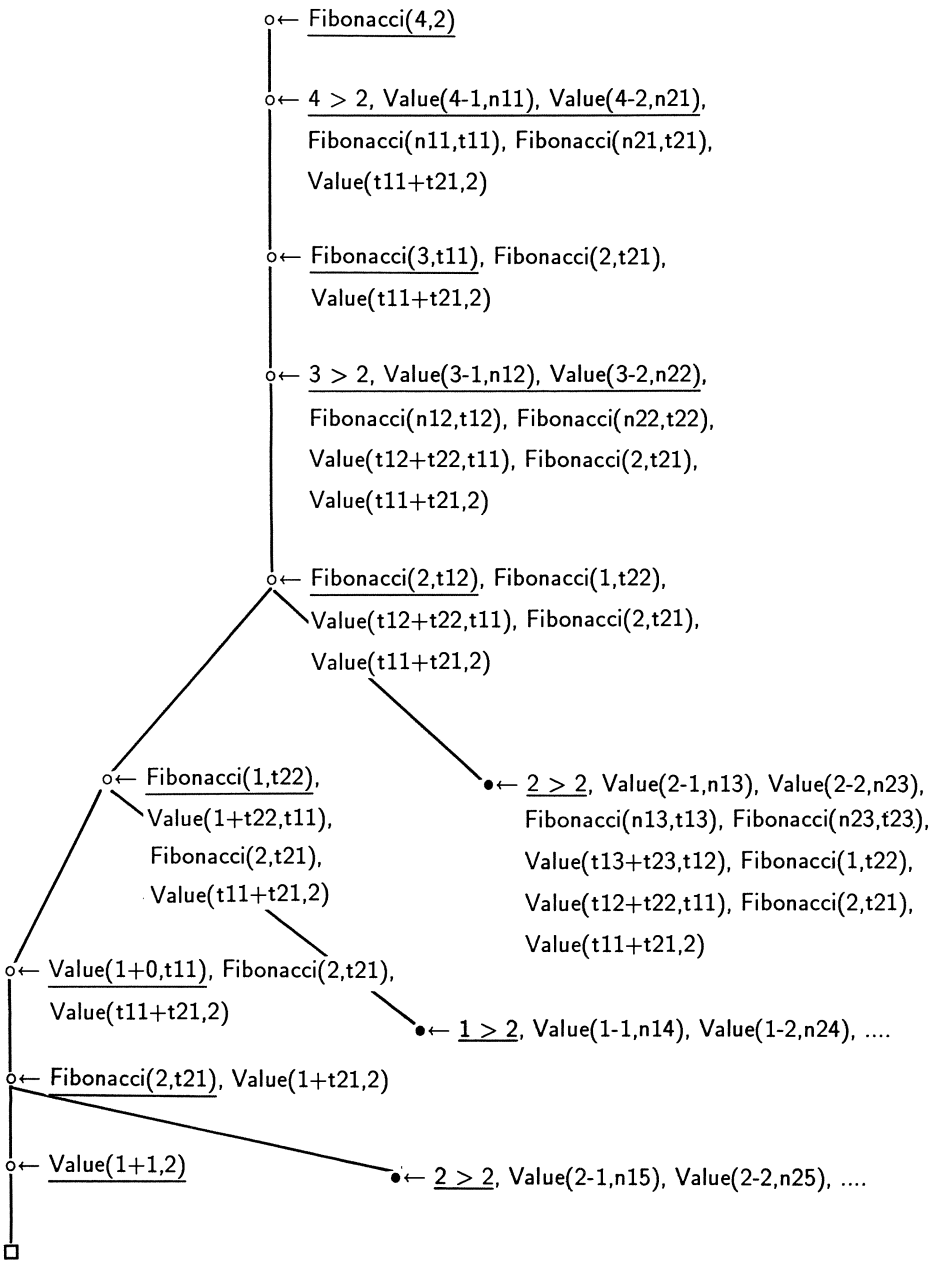

Figure A.2. Top-down search space for Fibonaccid(4,2)

Exercise 3:

$\leftarrow$ Celestial_body('Halleys comet')
Celestial_body(x1) $\leftarrow$ Comet(x1)

$\theta_1 = \{\langle$x1,'Halleys comet'$\rangle\}$

$\leftarrow$ Comet('Halleys comet')
Comet(x2) $\leftarrow$ Tail(x2),Close_to_sun(x2)

$\theta_2 = \{\langle$x2,'Halleys comet'$\rangle\}$

$\leftarrow$ Tail('Halleys comet'),Close_to_sun('Halleys comet')
Tail('Halleys comet') $\leftarrow$

$\theta_3 = \{\}$

$\leftarrow$ Close_to_sun('Halleys comet')
Close_to_sun('Halleys comet') $\leftarrow$

$\leftarrow$

Exercise 4:

The conclusion is negated.

$$\neg(\neg Equilateral(T) \leftarrow Has\_obtuse\_angle(T))$$
$$\Leftrightarrow$$
$$\neg(\neg Equilateral(T) \vee \neg Has\_obtuse\_angle(T))$$
$$\Leftrightarrow$$
$$Equilateral(T) \wedge Has\_obtuse\_angle(T)$$

In Horn form

(3) Equilateral(T) $\leftarrow$
(4) Has_obtuse_angle(T) $\leftarrow$

Equilateral(T) $\leftarrow$ (3)
Equiangular(x1) $\leftarrow$ Equilateral(x1) (2)

$\theta_1 = \{\langle$x1,T$\rangle\}$

Equiangular(T) $\leftarrow$
$\leftarrow$ Equiangular(x2), Has_obtuse_angle(x2) (1)

$\theta_2 = \{\langle$x2,T$\rangle\}$

$\leftarrow$ Has_obtuse_angle(T)
Has_obtuse_angle(T) $\leftarrow$

$\leftarrow$

A.3 Round 3

Structures

Exercise 1:

Even(0) ←
Even(S(S(x))) ←
    Even(x)

Exercise 2:

Less_than(0,S(x)) ←
Less_than(S(x),S(y)) ←
    Less_than(x,y)

Lists

Exercise 1:

Remove(elem,elem.list,list) ←
Remove(elem,x.list1,x.list2) ← elem ≠ x,
    Remove(elem,list1,list2)

← Remove(Gamma,Alfa.Beta.Gamma.Delta.∅,newlist)
← Remove(Gamma,list,Alfa.Beta.Delta.∅)

Exercise 2:

Number_of_elements(0,∅) ←
Number_of_elements(S(n),elem.restlist) ←
    Number_of_elements(n,restlist)

Exercise 3:

Insert(S(0),elem,list,elem.list) ←
Insert(S(n),element,x.list1,x.list2) ←
    Insert(n,elem,list1,list2)

← Insert(S(S(S(0))),Mi,Do.Re.Fa.Sol.∅,list)
← Insert(S(S(S(0))),element,list,Do.Re.Mi.Fa.Sol.∅)

Exercise 4:

Quicksort(∅,∅) ←
Quicksort(elem.list,slist) ←
    Partition(list,elem,list1,list2) ,
    Quicksort(list1,slist1),
    Quicksort(list2,slist2),
    Append(slist1,elem.slist2,slist)

Partition(∅,x,∅,∅) ←
Partition(elem.list,x,elem.list1,list2) ← elem ≤ x ,
    Partition(list,x,list1,list2)
Partition(elem.list,x,list1,elem.list2) ← elem > x ,
    Partition(list,x,list1,list2)

Trees

Exercise 1:

```
Subtrees(node,T(l,node,r),l,r) ←
Subtrees(node,T(l,root,r),l1,r1) ← node ≠ root,
        Subtrees(node,l,l1,r1)
Subtrees(node,T(l,root,r),l1,r1) ← node ≠ root,
        Subtrees(node,r,l1,r1)
```

Exercise 2:

```
Motherhood(root,node,T(T(l1,node,r1),root,r)) ←
Motherhood(root,node,T(l,root,T(l2,node,r2))) ←
Motherhood(root,node,T(l,root1,r)) ←
        Motherhood(root,node,l)
Motherhood(root,node,T(l,root1,r)) ←
        Motherhood(root,node,r)
```

Exercise 3:

```
Append(∅,t,t) ←
Append(T(l,root,r),t,T(l,root,r1)) ←
        Append(r,t,r1)
```

Exercise 4:

The program Remove is defined at the end of Sect. 4.2 on databases.

Difference lists

Exercise 1:

```
Quicksort(∅,D(x,x)) ←
Quicksort(x.y,D(u,w)) ←
        Partition(y,x,y1,y2),
        Quicksort(y1,D(u1,w1)), Quicksort(y2,D(u2,w2)),
        Conc(D(u1,w1),D(x.u2,w2),D(u,w))
Quicksort(∅,D(x,x)) ←
Quicksort(x.y,D(u,w)) ←
        Partition(y,x,y1,y2),
        Quicksort(y1,D(u,x.l)),
        Quicksort(y2,D(l,w))
```

Exercise 2:

```
Pre_order(0,D(x,x)) ←
Pre_order(T(x,r,y),D(r.u,w)) ←
        Pre_order(x,D(u,z)),
        Pre_order(y,D(z,w))
```

Exercise 3:

$\{\langle x,A.B.C.D.E.F.w\rangle,\langle y,D.E.F.w\rangle,\langle v,\ D.E.F.w\rangle,\langle z,w\rangle,\langle l,A.B.C.D.E.F.w\rangle,\langle ls,w\rangle\}$

Arrays

Exercise 1:

```
Knights_tour(1,(Route(1,1),itinerary),n) ←
        Start_tour(board,n),
        Visit(r(1,1),board,itinerary)
Knights_tour(2,(square,itinerary),n) ←
        Knights_tour(1,(prev_square,prev_route),n),
        Possible_step(prev_square,square,n),
        Visit(square,prev_route,itinerary)
Knights_tour(visited,(square,itinerary),n) ←
        Value(visited - 1,prev_visited),
        Knights_tour(prev_visited,(prev_square,prev_route),n)
        Possible_step(prev_square,square,n),
        NotVisited(square,prev_route),
        Visit(square,prev_route,itinerary)

    Possible_step(prev_square,square,n) ←
            Step(prev_square,square),
            Within(square,n)

/* It is possible to advance to a square
        if it is a permitted advance for the knight
        and the square is within the limits of the board.*/
Possible_step(prev_square,square,n) ←
        Step(prev_square,square),
        Within(square,n)

/* The valid advances of the knight */
Step(S(row,col),S(row1,col1)) ←
        Value(row-2,row1), Value(col-1,col1);
        Value(row-2,row1), Value(col+1,col1);
        Value(row-1,row1), Value(col+2,col1);
        Value(row+1,row1), Value(col+2,col1);
        Value(row+2,row1), Value(col+1,col1);
        Value(row+2,row1), Value(col-1,col1);
        Value(row+1,row1), Value(col-2,col1);
        Value(row-1,row1), Value(col-2,col1)

/* The board has squares from 1 up to n */
Within(S(row,col),n) ←
        row > 0, col > 0, row ≤ n, col ≤ n

NotVisited(S(r,c),itinerary) ←
        Not Array_Value(itinerary,[r,c],1)

Visit(S(r,c),prev_route,itinerary) ←
        New_Array_Value(prev_route,[r,c],1,itinerary)
```

```
Start_tour(itenary,n)←
        Array([n,n],itenary)
```

A.4 Round 4

Exercise 1:

```
Odd_Numbers(∅,∅) ←
Odd_Numbers(x.list1,x.list2) ← Odd(x),
        Odd_Numbers(list1,list2)
Odd_Numbers(x.list1,list2) ← Not Odd(x),
        Odd_Numbers(list1,list2)

Odd(x) ←
        Value(x Mod 2,remainder),
        remainder=1
```

```
← Odd_Numbers(1.2.3.4.5.∅,list)
list = 1.3.5.∅ ;
no
```

Exercise 2:

```
Leopard(x) ← Color(x,Tawny), Spotted(x)
Leopard(x) ← Color(x,Black), Not Spotted(x)
```

a)
```
← All_answers(animal,Leopard(animal),leopards)
leopards = Blackie.Jarvis.Leo.∅
animal = _
```

b)
```
← All_answers((animal,color),
              (Leopard(animal),Color(animal,color)),
              name_color)
name_color = (Blackie,Black).(Jarvis,Tawny).(Leo,Tawny).∅
animal = _
color = _
```

c)
```
← All_answers(animal,(Feline(animal),Not Spotted(animal)),
              not_spotted)
not_spotted = Tom.Leona.Shere-Khan.Amur.Juba.Blackie.∅
animal = _
```

d)
```
← Sorted_answers(zoo,
                 animal^price^(Zoo(zoo,animal,price)),zoo_list)
zoo_list = Bronx_Zoo.Central_Park,Childrens_Zoo.∅
zoo = _
animal = _
price = _
```

e) ← Sorted_answers(zoo,
cat^price^(Big_cat(cat), Zoo(zoo,cat,price)),
big_cats)
big_cats = Bronx_Zoo.Central_Park.∅
zoo = _
cat = _
price = _

f) ← Not Leopard(Louis)
yes

g) ← Not Leopard(Blackie)
no

h) ← Not Spotted(x)

We get an execution error since this question can only be true if x get instantiated during the execution.

Exercise 3:

Shortest_path(town1,town2,itinerary) ←
Sorted_answers((dist,path),Path(town1,town2,town1.∅,path,dist),list),
list = (shortest_dist,shortest_path).restlist,
Reverse(shortest_path,itinerary)

Path(town,town,path,path,0) ←
Path(town1,town2,path,itinerary,dist) ←
(Distance(town1,town3,addition) ; Distance(town3,town1,addition)),
Not Member(town3,itinerary),
Path(town3,town2,town3.itinerary,earlier_dist),
Value(earlier_dist + addition,dist)

A.5 Round 5

Exercise 1:

$\forall x\,(\mathit{Computing\_science\_town}(x) \leftrightarrow x = \mathit{Uppsala} \vee x = \mathit{Linkoping})$

Exercise 2:

(1) $\mathit{Computing\_science\_town}(\mathit{Uppsala}) \leftarrow$ $\mathit{Uppsala} = \mathit{Uppsala} \vee \mathit{Uppsala} = \mathit{Linkoping}$	$\forall$E , $\leftrightarrow$E $(e1)$
(2) $\mathit{Uppsala} = \mathit{Uppsala}$	$\forall$E $(d1)$
(3) $\mathit{Uppsala} = \mathit{Uppsala} \vee \mathit{Uppsala} = \mathit{Linkoping}$	$\vee$I (2)
(4) $\mathit{Computing\_science\_town}(\mathit{Uppsala})$	$\leftarrow$E (1)(3)

(1)	$Computing\_science\_town(Linkoping) \leftarrow$ $Linkoping = Uppsala \vee Linkoping = Linkoping$	$\forall$E , $\leftrightarrow$E $(e1)$
(2)	$Linkoping = Linkoping$	$\forall$E $(d1)$
(3)	$Linkoping = Uppsala \vee Linkoping = Linkoping$	$\vee$I (2)
(4)	$Computing\_science\_town(Linkoping)$	$\leftarrow$E (1)(3)

Exercise 3:

$\forall x \forall m\,(Max(x, m) \leftrightarrow Member(m, x) \wedge \forall u\,(Member(u, x) \rightarrow u \leq m))$

Exercise 4:

The relation shall hold for all lists with one or more elements. Consequently, the base case is a list with one element.
We need the relation $\leq$.

$$\forall x \forall y\,(x \leq y \leftrightarrow x = y \vee x < y) \qquad (a1)$$

We also need transitivity of $\leq$.

$$\forall x \forall y \forall z\,(x \leq y \wedge y \leq z \rightarrow x \leq z) \qquad (a2)$$

Derivation of the base case:

(1)	$Max(e.\emptyset, m) \leftarrow$ $Member(m, e.\emptyset) \wedge \forall u\,(Member(u, e.\emptyset) \rightarrow u \leq m))$	$\forall$E , $\leftrightarrow$E $(e13)$
(2)	$List(e.\emptyset) \rightarrow Member(m, e.\emptyset) \leftrightarrow m = e$	$\forall$E Lemma1
(3)	$m = e$	hyp
(4)	$List(e.\emptyset)$	hyp
(5)	$Member(m, e.\emptyset) \leftrightarrow m = e$	$\rightarrow$E (2)(4)
(6)	$Member(m, e.\emptyset) \leftarrow m = e$	$\leftrightarrow$E (5)
(7)	$Member(m, e.\emptyset)$	$\leftarrow$E (3)(6)
(8)	$Member(e1, e.\emptyset)$	hyp
(9)	$List(e.\emptyset) \rightarrow (Member(e1, e.\emptyset) \leftrightarrow e1 = e)$	$\forall$E Lemma1
(10)	$Member(e1, e.\emptyset) \rightarrow e1 = e$	$\rightarrow$E , $\leftrightarrow$E (4)(9)
(11)	$e1 = e$	$\rightarrow$E (8)(10)
(12)	$e1 = m$	id (3)(11)
(13)	$e1 = m \vee e1 < m$	$\vee$I (12)
(14)	$e1 \leq m \leftarrow e1 = m \vee e1 < m$	$\forall$E $(a1)$
(15)	$e1 \leq m$	$\leftarrow$E (13)(14)
(16)	$Member(e1, e.\emptyset) \rightarrow e1 \leq m$	$\rightarrow$I (8)(15)
(17)	$\forall u\,(Member(u, e.\emptyset) \rightarrow u \leq m)$	$\forall$I (16)
(18)	$Member(m, e.\emptyset) \wedge \forall u\,(Member(u, e.\emptyset) \rightarrow u \leq m)$	$\wedge$I (7)(17)
(19)	$Max(e.\emptyset, m)$	$\leftarrow$E (1)(18)
(20)	$Max(e.\emptyset, m) \leftarrow m = e \wedge List(e.\emptyset)$	$\leftarrow$I (3)(4)(19)
(21)	$\forall x \forall m\,(Max(x.\emptyset, m) \leftarrow m = x \wedge List(x.\emptyset))$	$\forall$I (20)

Derivation of Lemma1

$$\forall x \forall u\,(List(x.\emptyset) \rightarrow (Member(u, x.\emptyset) \leftrightarrow u = x))$$

from the definitions ($d9$) and ($d10$).

(1)	$List(x.\emptyset) \rightarrow (Member(u, x.\emptyset) \leftrightarrow u = x \vee Member(u, \emptyset))$	$\forall$E ($d10$)
(2)	$List(x.\emptyset)$	hyp
(3)	$Member(u, x.\emptyset) \leftrightarrow u = x \vee Member(u, \emptyset)$	$\rightarrow$E (1)(2)
(4)	$Member(u, x.\emptyset) \rightarrow u = x \vee Member(u, \emptyset)$	$\leftrightarrow$E (3)
(5)	$Member(u, x.\emptyset) \leftarrow u = x \vee Member(u, \emptyset)$	$\leftrightarrow$E (3)
(6)	$Member(u, x.\emptyset)$	hyp
(7)	$u = x \vee Member(u, \emptyset)$	$\rightarrow$E (4)(6)
(8)	$u = x$	hyp
(9)	$u = x \rightarrow u = x$	$\rightarrow$I (8)
(10)	$Member(u, \emptyset)$	hyp
(11)	$\neg\, Member(u, \emptyset)$	$\forall$E ($d9$)
(12)	$\bot$	$\neg$E (10)(11)
(13)	$u = x$	$\bot_{int}$ (12)
(14)	$Member(u, \emptyset) \rightarrow u = x$	$\rightarrow$I (10)(13)
(15)	$u = x$	$\vee$E (7)(9)(14)
(16)	$Member(u, x.\emptyset) \rightarrow u = x$	$\rightarrow$I (6)(15)
(17)	$u = x$	hyp
(18)	$u = x \vee Member(u, \emptyset)$	$\vee$I (17)
(19)	$Member(u, x.\emptyset)$	$\leftarrow$E (5)(18)
(20)	$Member(u, x.\emptyset) \leftarrow u = x$	$\leftarrow$I (17)(19)
(21)	$Member(u, x.\emptyset) \leftrightarrow u = x$	$\leftrightarrow$I (16)(20)
(22)	$List(x.\emptyset) \rightarrow Member(u, x.\emptyset) \leftrightarrow u = x$	$\rightarrow$I (2)(21)
(23)	$\forall x \forall u\,(List(x.\emptyset) \rightarrow (Member(u, x.\emptyset) \leftrightarrow u = x))$	$\forall$I (22)

Let us take the recursive case.

(1)	$Max(e.l, m1) \leftarrow$ $Member(m1, e.l) \wedge \forall u\,(Member(u, e.l) \rightarrow u \leq m1))$	$\forall$E , $\leftrightarrow$E ($e13$)
(2)	$Max(l, m2)$	hyp
(3)	$Max(l, m2) \rightarrow$ $Member(m2, l) \wedge \forall u\,(Member(u, l) \rightarrow u \leq m2))$	$\forall$E , $\leftrightarrow$E ($e13$)
(4)	$Member(m2, l) \wedge \forall u\,(Member(u, l) \rightarrow u \leq m2))$	$\rightarrow$E (2)(3)
(5)	$Member(m2, l)$	$\wedge$E (4)
(6)	$\forall u\,(Member(u, l) \rightarrow u \leq m2)$	$\wedge$E (4)
(7)	$e \leq m2 \wedge m1 = m2$	hyp
(8)	$e \leq m2$	$\wedge$E (7)
(9)	$m1 = m2$	$\wedge$E (7)
(10)	$List(e.l)$	hyp

(11) $Member(m1, l)$ id (5)(9)
(12) $List(e.l) \rightarrow (Member(m1, e.l) \leftarrow Member(m1, l))$ $\forall$E Lemma2
(13) $Member(m1, e.l) \leftarrow Member(m1, l)$ $\rightarrow$E (10)(12)
(14) $Member(m1, e.l)$ $\leftarrow$E (11)(13)
(15) $\forall u\,(Member(u, l) \rightarrow u \leq m2) \wedge e \leq m2 \wedge m1 = m2 \wedge List(e, l)$ $\wedge$I (6)(7)(10)
(16) $\forall u\,(Member(u, l) \rightarrow u \leq m2) \wedge e \leq m2 \wedge m1 = m2 \wedge List(e, l) \rightarrow \forall u\,(Member(u, e.l) \rightarrow u \leq m1)$ $\forall$E Lemma3
(17) $\forall u\,(Member(u, e.l) \rightarrow u \leq m1)$ $\rightarrow$E (15)(16)
(18) $Member(m1, e.l) \wedge \forall u\,(Member(u, e.l) \rightarrow u \leq m1)$ $\wedge$I (14)(17)
(19) $Max(e.l, m1)$ $\leftarrow$E (1)(18)
(20) $Max(e.l, m1) \leftarrow Max(l, m2) \wedge e \leq m2 \wedge m1 = m2 \wedge List(e.l)$ $\leftarrow$I (2)(7)(10)(19)
(21) $\forall x \forall y \forall m \forall w\,(Max(x.y, m) \leftarrow Max(y, w) \wedge x \leq w \wedge m = w \wedge List(x.y))$ $\forall$I (20)

This is a recursive case. We have assumed that the first element on the list $x.y$ is less than or equal to the maximum number found in the list y so far, i.e. $x \leq w$. The maximum number on the list $x.y$ is denoted by m and is consequently equal to w.
We also need to study the case when the first number on the list $x.y$ is greater than w, i.e. $w < x$. The maximum number on the list is then x, i.e. $m = x$.

(1) $Max(e.l, m1) \leftarrow Member(m1, e.l) \wedge \forall u\,(Member(u, e.l) \rightarrow u \leq m1))$ $\forall$E , $\leftrightarrow$E (e13)
(2) $Max(l, m2)$ hyp
(3) $Max(l, m2) \rightarrow Member(m2, l) \wedge \forall u\,(Member(u, l) \rightarrow u \leq m2))$ $\forall$E , $\leftrightarrow$E (e13)
(4) $Member(m2, l) \wedge \forall u\,(Member(u, l) \rightarrow u \leq m2))$ $\rightarrow$E (2)(5)
(5) $Member(m2, l)$ $\wedge$E (4)
(6) $\forall u\,(Member(u, l) \rightarrow u \leq m2)$ $\wedge$E (4)
(7) $m2 < e \wedge m1 = e$ hyp
(8) $m2 < e$ $\wedge$E (7)
(9) $m1 = e$ $\wedge$E (7)
(10) $List(e.l)$ hyp
(11) $Member(m1, l)$ id (5)(9)
(12) $List(e.l) \rightarrow (Member(m1, e.l) \leftarrow Member(m1, l))$ $\forall$E Lemma2
(13) $Member(m1, e.l) \leftarrow Member(m1, l)$ $\rightarrow$E (10)(12)
(14) $Member(m1, e.l)$ $\leftarrow$E (11)(13)
(15) $\forall u\,(Member(u, l) \rightarrow u \leq m2) \wedge m2 < e \wedge m1 = e \wedge List(e.l)$ $\wedge$I (6)(7)(10)

(16) $\forall u\,(Member(u,l) \rightarrow u \leq m2) \wedge m2 < e \wedge m1 = e \wedge$ $\forall$E Lemma4
$List(e.l) \rightarrow$
$\forall u\,(Member(u,e.l) \rightarrow u \leq m1)$
(17) $\forall u\,(Member(u,e.l) \rightarrow u \leq m1)$ $\rightarrow$E (15)(16)
(18) $Member(m1,e.l) \wedge \forall u\,(Member(u,e.l) \rightarrow u \leq m1)$ $\wedge$I (14)(17)
(19) $Max(e.l,m1)$ $\leftarrow$E (1)(18)
(20) $Max(e.l,m1) \leftarrow$ $\leftarrow$I (2)(7)(10)(19)
$Max(l,m2) \wedge m2 < e \wedge m1 = e \wedge List(e.l)$
(21) $\forall x \forall y \forall m \forall w\,(Max(x.y,m) \leftarrow$ $\forall$I (20)
$Max(y,w) \wedge w < x \wedge m = x \wedge List(x.y)$

Derivation of Lemma2

$$\forall x \forall y \forall u\,(List(x.y) \rightarrow (Member(u,x.y) \leftarrow Member(u,y)))$$

from the definition $(d10)$.

(1) $List(x.y) \rightarrow (Member(u,x.y) \leftrightarrow u = x \vee Member(u,y))$ $\forall$E $(d10)$
(2) $List(x.y)$ hyp
(3) $Member(u,x.y) \leftarrow u = x \vee Member(u,y)$ $\rightarrow$E , $\leftrightarrow$E (1)(2)
(4) $Member(u,y)$ hyp
(5) $u = x \vee Member(u,y)$ $\vee$I (4)
(6) $Member(u,x.y)$ $\leftarrow$E (3)(5)
(7) $Member(u,x.y) \leftarrow Member(u,y)$ $\leftarrow$I (4)(6)
(8) $List(x.y) \rightarrow (Member(u,x.y) \leftarrow Member(u,y))$ $\rightarrow$I (2)(7)
(9) $\forall x \forall y \forall u\,(List(x.y) \rightarrow (Member(u,x.y) \leftarrow Member(u,y)))$ $\forall$I (8)

Derivation of Lemma3

$$\forall x \forall y \forall w \forall m\,(\forall u\,(Member(u,y) \rightarrow u \leq w) \wedge x \leq w \wedge m = w \wedge List(x,y) \rightarrow \forall u\,(Member(u,x.y) \rightarrow u \leq w))$$

(1) $\forall u\,(Member(u,y) \rightarrow u \leq w) \wedge x \leq w \wedge m = w \wedge List(x.y)$ hyp
(2) $\forall u\,(Member(u,y) \rightarrow u \leq w)$ $\wedge$E (1)
(3) $x \leq w$ $\wedge$E (1)
(4) $m = w$ $\wedge$E (1)
(5) $List(x.y)$ $\wedge$E (1)
(6) $Member(u,x.y)$ hyp
(7) $List(x.y) \rightarrow (Member(u,x.y) \leftrightarrow u = x \vee Member(u,y))$ $\forall$E $(d10)$
(8) $Member(u,x.y) \leftrightarrow u = x \vee Member(u,y)$ $\rightarrow$E (5)(7)
(9) $Member(u,x.y) \rightarrow u = x \vee Member(u,y)$ $\leftrightarrow$E (8)
(10) $u = x \vee Member(u,y)$ $\rightarrow$E (6)(9)
(11) $u = x$ hyp

(12)	$u \leq m$	id (3)(4)(11)
(13)	$u = x \rightarrow u \leq m$	→I (11)(12)
(14)	$Member(u, y) \rightarrow u \leq w$	∀E (2)
(15)	$Member(u, y) \rightarrow u \leq m$	id (4)(14)
(16)	$u \leq m$	∨E (10)(13)(15)
(17)	$Member(u, x.y) \rightarrow u \leq m$	→I (6)(16)
(18)	$\forall u\,(Member(u, y) \rightarrow u \leq w) \wedge x \leq w \wedge m = w \wedge List(x, y) \rightarrow \forall u\,(Member(u, x.y) \rightarrow u \leq w)$	→I (1)(17)
(19)	$\forall x \forall y \forall w \forall m\,(\forall u\,(Member(u, y) \rightarrow u \leq w) \wedge x \leq w \wedge m = w \wedge List(x, y) \rightarrow \forall u\,(Member(u, x.y) \rightarrow u \leq w))$	∀I (18)

Derivation of Lemma4

$$\forall x \forall y \forall w \forall m\,(\forall u\,(Member(u, y) \rightarrow u \leq w) \wedge w < x \wedge m = x \wedge List(x, y) \rightarrow \forall u\,(Member(u, x.y) \rightarrow u \leq m))$$

(1)	$\forall u\,(Member(u, y) \rightarrow u \leq w) \wedge w < x \wedge m = x \wedge List(x.y)$	hyp
(2)	$\forall u\,(Member(u, y) \rightarrow u \leq w)$	∧E (1)
(3)	$w < x$	∧E (1)
(4)	$m = x$	∧E (1)
(5)	$List(x.y)$	∧E (1)
(6)	$Member(u, x.y)$	hyp
(7)	$List(x.y) \rightarrow (Member(u, x.y) \leftrightarrow u = x \vee Member(u, y))$	∀E ($d10$)
(8)	$Member(u, x.y) \leftrightarrow u = x \vee Member(u, y)$	→E (5)(7)
(9)	$Member(u, x.y) \rightarrow u = x \vee Member(u, y)$	↔E (8)
(10)	$u = x \vee Member(u, y)$	→E (6)(9)
(11)	$u = x$	hyp
(12)	$u = m$	id (4)(11)
(13)	$u = m \vee u < m$	∨I (12)
(14)	$u \leq m \leftrightarrow u = m \vee u < m$	∀E ($a1$)
(15)	$u \leq m \leftarrow u = m \vee u < m$	↔E (14)
(16)	$u \leq m$	←E (13)(15)
(17)	$u = x \rightarrow u \leq m$	→I (11)(16)
(18)	$Member(u, y) \rightarrow u \leq w$	∀E (2)
(19)	$Member(u, y)$	hyp
(20)	$u \leq w$	→E (18)(19)
(21)	$w = x \vee w < x$	∨I (3)
(22)	$w \leq x \leftrightarrow w = x \vee w < x$	∀E ($a1$)
(23)	$w \leq x \leftarrow w = x \vee w < x$	↔E (22)
(24)	$w \leq x$	←E (21)(23)
(25)	$u \leq w \wedge w \leq x$	∧I (20)(24)

(26) $u \leq w \wedge w \leq x \rightarrow u \leq x$	$\forall$E $(a2)$
(27) $u \leq x$	$\rightarrow$E (25)(26)
(28) $u \leq m$	id (4)(27)
(29) $Member(u, y) \rightarrow u \leq m$	$\rightarrow$I (19)(28)
(30) $u \leq m$	$\vee$E (10)(17)(29)
(31) $Member(u, x.y) \rightarrow u \leq m$	$\rightarrow$I (6)(30)
(32) $\forall u\,(Member(u, x.y) \rightarrow u \leq m)$	$\forall$I (31)
(33) $\forall u\,(Member(u, y) \rightarrow u \leq w) \wedge w < x \wedge$ $m = x \wedge List(x.y) \rightarrow$ $\forall u\,(Member(u, x.y) \rightarrow u \leq m)$	$\rightarrow$I (1)(32)
(34) $\forall x \forall y \forall w \forall m\,(\forall u\,(Member(u, y) \rightarrow u \leq w) \wedge w < x \wedge$ $m = x \wedge List(x.y) \rightarrow$ $\forall u\,(Member(u, x.y) \rightarrow u \leq m))$	$\forall$I (33)

Exercise 5:

$\forall x \forall y \forall z\,(Maxnumber(x, y, z) \leftrightarrow (x \leq y \wedge z = y) \vee (y < x \wedge z = x))$

Exercise 6:

The derivation of the base case is not changed.

From the specification of *Maxnumber* we can deduce the following lemma

$$\forall x \forall y \forall z\,(Maxnumber(x, y, z) \rightarrow x \leq z \wedge y \leq z)$$

Using this lemma we can easily derive a program clause

$$\forall x \forall y \forall w \forall m\,(Max(x.y, m) \leftarrow Max(y, w) \wedge Maxnumber(x, w, m))$$

Exercise 7:

$$\forall x \forall y\,(\exists z\,(Father(x, z) \wedge (Father(z, y) \vee Mother(z, y))) \rightarrow Paternal\_grandparent(x, y))$$

Exercise 8:

a) $\forall x \forall y (List(x) \wedge List(y) \rightarrow \exists z\, Intersection(x, y, z))$

b) The derivation is performed by induction over the first argument of the relation *Intersection*. The induction schema for lists is used. The program clauses for Intersection and Member are the premises for the derivation.

Base case: From the program Intersection we derive the theorem with the first argument instantiated to an empty list.

$$\forall y \exists z\, Intersection(\emptyset, y, z)$$

Outline of the derivation of the induction step:

(1) $\forall y \exists z\ Intersection(x, y, z)$	hyp
...	
(n-2) $\forall y \exists z\ Intersection(u.x, y, z)$	
(n-1) $\forall y \exists z\ Intersection(x, y, z) \rightarrow$ $\forall y \exists z\ Intersection(u.x, y, z)$	$\rightarrow$I (1)(n-2)
(n) $\forall u \forall x\ (\forall y \exists z\ Intersection(x, y, z) \rightarrow$ $\forall y \exists z\ Intersection(u.x, y, z))$	$\forall$I (n-1)

c) The program for Intersection is not complete according to the specification. In the program the elements in the list used as third argument are always in the same order as in the first argument. According to the specification the elements in the list can be given in any order.

Exercise 9:

$$\forall x \forall u \forall x1\ (\forall v\ (Member(v, x) \vee v = u \leftrightarrow Member(v, x1)) \wedge Ord\_binary\_tree(x) \wedge Ord\_binary\_tree(x1) \leftrightarrow Insert(u, x, x1))$$

Exercise 10:

a) $\forall x\ Append(x, \emptyset, x)$

b) The theorem is proved by induction

Base case: derive $Append(\emptyset, \emptyset, x)$ from the program.

Induction step:

(1) $Append(x, \emptyset, x)$	hyp
...	
(n-2) $Append(u.x, \emptyset, u.x)$	
(n-1) $Append(x, \emptyset, x) \rightarrow Append(u.x, \emptyset, u.x)$	$\rightarrow$I (1)(n-2)
(n) $\forall u \forall x (Append(x, \emptyset, x) \rightarrow Append(u.x, \emptyset, u.x))$	$\forall$I (n-1)

A.6 Round 6

Exercise 1:

```
Subset(∅,x) ←
Subset(u.x,y) ← Member(u,y) |,
        Subset(x,y)
```

Exercise 2:

```
Subtree(r,T(left,r,right),left,right) ← |
Subtree(x,T(left,r,right),left1,right1) ←
        Subtree(x,left,left1,right1)
Subtree(x,T(left,r,right),left1,right1) ←
        Subtree(x,right,left1,right1)
```

Exercise 3:

```
Subset(x,y) ←
        Cases (x = ∅: True,
               (x = u.x1 , Member(u,y)) : Subset(x,y))
```

Exercise 4:

Union defined without cut

```
Union(∅,list,list) ←
Union(elem.list1,list2,list3) ← Member(elem,list2),
        Union(list1,list2,list3)
Union(elem.list1,list2,elem.list3) ← Not Member(elem,list2),
        Union(list1,list2,list3)
```

Union defined with cut where all clauses express the union relation independently.

```
Union(∅,list,list) ←
Union(elem.list1,list2,list3) ← Member(elem,list2)|,
        Union(list1,list2,list3)
Union(elem.list1,list2,elem.list3) ← Not Member(elem,list2),
        Union(list1,list2,list3)
```

Union defined with cut where the third clause does not express the union relation.

```
Union(∅,list,list) ←
Union(elem.list1,list2,list3) ← Member(elem,list2)|,
        Union(list1,list2,list3)
Union(elem.list1,list2,elem.list3) ←
        Union(list1,list2,list3)
```

Exercise 5:

```
Insert(elem,∅,elem.∅) ←
Insert(newelem,elem.list,newelem.elem.list) ← newelem ≤ elem|
Insert(newelem,elem.list1,elem.list2) ← newelem > elem,
        Insert(newelem,list1,list2)
```

Exercise 6:

```
Transform((p → q),(¬p1 ∨ q1)) ← |
        Transform(p,p1),
        Transform(q,q1)
Transform((p ↔ q),((¬p1 V q1) & (¬q1 V p1))) ← |
        Transform(p,p1),
        Transform(q,q1)
Transform((p ∧ q),(p1 ∧ q1)) ← |
        Transform(p,p1),
        Transform(q,q1)
```

```
Transform((p ∨ q),(p1 ∨ q1)) ← |
        Transform(p,p1),
        Transform(q,q1)
Transform((¬p),(¬p1)) ← |
        Transform(p,p1)
Transform(p,p) ←
```

Exercise 7:

A maternal grandmother of x is y if the mother of x is z and the mother of z is y. The efficiency of the evaluation of a question is closely connected to the fact that the order of the conditions follows input-output patterns. We can find here the following control alternatives making up the definition.

```
Maternal_grandmother(x,y):Pattern(?,-) ←
        Mother(x,z),
        Mother(z,y)
Maternal_grandmother(x,y):Pattern(?,+) ←
        Mother(z,y),
        Mother(x,z))
```

The control makes the evaluation first look for who the grandmother is mother of when we know the grandmother, before looking for who her possibly unknown daughter is mother of. When the grandmother is not known, we look for a mother of x and then the mother's mother.

Exercise 8:

```
Same_leaves(tree1,tree2) ←
        Leaves(tree1,leaves),
        Leaves(tree2,leaves):Protected(leaves)

Leaves(T(∅,node,∅),node.∅) ←
Leaves(T(left,root,right),list) ←
        Leaves(left,leftlist),
        Leaves(right,rightlist),
        Append(leftlist,rightlist,list)
```

We protect the leaves in one of the conditions for becoming bound.
In the sequential variant using binding requirements we put the requirement in the first condition which implies that the second becomes the producer. The evaluation then succesively finds one leaf at each tree and compares them. This will give us the least possible computation for the case when the trees are different.

```
Same_leaves(tree1,tree2) ←
        Leaves(tree1,leaves):Bound(leaves),
        Leaves(tree2,leaves)
```

A.7 Round 7

Exercise 1:

```
Assemble_Info ←
        Outflow(Person),
        Read_File(Name,name),
        Read_File(Address,no),
        Check_Info(name,no),
        Close_Files
Check_Info(End_of_file,End_of_file) ←
Check_Info(End_of_file,no) ←
        no ≠ End_of_file,
        Fault('Different number of entries in the files')
Check_Info(name,End_of_file) ←
        name ≠ End_of_file,
        Fault('Different number of entries in the files')
Check_Info(name,no2) ←
        Read_File(Name,no1),
        Store_Info(name,no2,no1)
Store_Info(name,no,no) ← |
        Read_File(Address,addr),
        Write(name), Write(no), Write(addr),
        Read_File(Name,new_name),
        Read_File(Address,new_no),
        Check_Info(new_name,new_no))
Store_Info(name,no2,no1) ←
        no1 ≠ no2, Fault('The clients numbers do not match')
Read_File(file,info) ←
        Inflow(file), In_term(info)
Write(x) ←
        Out_term(x), Out_term('.'), Line_shift
Fault(comment) ←
        Inflow(TTY), Out_term(comment), Line_shift
Close_Files ←
        Close('Name'), Close('Address'), Close('Person')
```

Exercise 2:

```
Assemble_addresses ←
        Inflow('Born'),
        In_term(name),
        Outflow('Born.new'),
        Store_Birthdays(name),
        Close_Files
```

```
Store_Birthdays(End_of_file) ←
Store_Birthdays(name) ←
        Write(name),
        In_term(year), Write(year),
        In_term(month), Write(month),
        In_term(day), Write(day),
        Read_Address(name,addr), Write(addr),
        Inflow('Born'), In_term(new_name),
        Store_Birthdays(new_name)
Read_Address(name,addr) ←
        Inflow(TTY), Out_term('The address of '),
        Out_term(name), Line_shift
        In_term(addr)
Write(x) ←
        Out_term(x), Out_term('.'), Line_shift
Close_Files ←
        Close('Born'),
        Close('Born.new')
```

Appendix B. Program Traces

B.1 Trace of Sum_of_triads

Our Prolog program for Sum_of_triads (45) – (47), which is a relation between a number and the sum of all numbers divisible by three that are less than the first number, will be shorter if we use the built-in predicates for arithmetic and represent our numbers by our common arabic numerals instead of by the constructor S. Since some of the predefined relations can only be used when some of their arguments are instantiated, the order of the conditions is important in a Prolog clause. It is not sufficient that all conditions be there; we also have to consider Prolog's order of execution. For instance, the condition Value(x+ys,s) must be placed so that we can be certain that both x and ys are instantiated before the relation is evaluated.

```
Sum_of_triads(0,0) ←                                          (89)
Sum_of_triads(x,s) ← x > 0,                                   (90)
        Triad(x), Value(x-1,y),
        Sum_of_triads(y,ys), Value(x+ys,s)
Sum_of_triads(x,s) ← x > 0,                                   (91)
        NonTriad(x), Value(x-1,y),
        Sum_of_triads(y,s)
```

In clauses (45) - (47) the base case and the recursive case were separated by the structure of the first argument. Clauses (89) – (91) all match a question about Sum_of_triads where the first argument is zero. The relation is defined only for positive integers, hence we must complement the recursive case by the condition $x = 0$ to avoid getting negative numbers when backtracking. We can thus replace the relation Sum by the built-in relation Value.

```
Triad(n) ← Value(n Mod 3,rem), rem = 0                        (92)

NonTriad(n) ← Value(n Mod 3,rem), rem ≠ 0                     (93)
```

Let us follow the execution of the question ← Sum_of_triads(6,s):

(1a) ← Sum_of_triads(6,s)

x0=6, s0=s

(2a) ← 6 > 0, Triad(6), Value(6-1,y0), Sum_of_triads(y0,ys0), Value(6+ys0,s)

n0=6

(3a) ← Value(6 Mod 3,rem0), rem0 = 0, Value(6-1,y0), Sum_of_triads(y0,ys0), Value(6+ys0,s)

rem0=0

(4a) ← 0 = 0, Value(6-1,y0), Sum_of_triads(y0,ys0), Value(6+ys0,s)

y0=5

(5a) ← Sum_of_triads(5,ys0), Value(6+ys0,s)

x1=5, s1=ys0

(6a) ← 5 >0, Triad(5), Value(5-1,y1), Sum_of_triads(y1,ys1), Value(5+ys1,ys0), Value(6+ys0,s)

n1=5

(7a) ← Value(5 Mod 3,rem1), rem1 = 0, Value(5-1,y1), Sum_of_triads(y1,ys1), Value(5+ys1,ys0), Value(6+ys0,s)

rem1=2

(8a) ← 2 = 0, Value(5-1,y1), Sum_of_triads(4,ys1), Value(5+ys1,ys0), Value(6+ys0,s)

The goal Sum_of_triads(6,s) cannot be unified with (89) because the first argument is not 0. It matches clause (90), however. The conditions are executed from left to right which means that Sum_of_triads will be executed first. Then the predicate Value(6-1,y) is executed. The goal Sum_of_triads(5,ys0) comes next and is matched with clause (90). We reach a dead end in (8a) since the predicate Triad(5) is not true. Prolog must backtrack to the latest choice point, i.e. the latest predicate which has an alternative clause able to give another solution. It is not possible to execute Value(5 Mode 3,rem1) in any other way. The same goes for Triad(5) because that clause has only one set of conditions and none of them can give an alternative solution. Not until (5a) do we see a solution to the problem. Sum_of_triads consists of three alternative clauses. The first does not match, the second we tried and failed to satisfy, but the third remains. Prolog unifies the goal Sum_of_triads(5,ys0) with clause (91) and the execution continues.

x1=5, s1=ys0

(6b) ← 5 > 0, NonTriad(5), Value(5-1,y1), Sum_of_triads(y1,ys0), Value(6+ys0,s)

n1=5

(7b) ← Value(5 Mod 3,rem1), rem1 ≠ 0, Value(5-1,y1), Sum_of_triads(y1,ys0), Value(6+ys0,s)

rem1=2

(8b) ← 2 ≠ 0, Value(5-1,y1), Sum_of_triads(y1,ys0), Value(6+ys0,s)
y1=4

(9b) ← Sum_of_triads(4,ys0), Value(6+ys0,s)
x2=4, s2=ys0

(10b) ← 4 > 0, Triad(4), Value(4-1,y2), Sum_of_triads(y2,ys2), Value(4+ys2,ys0), Value(6+ys0,s)
n2=4

(11b) ← Value(4 Mod 3,rem2), rem2 = 0, Value(4-1,y2), Sum_of_triads(y2,ys2), Value(4+ys2,ys0), Value(6+ys0,s)
rem2 = 1

(12b) ← 1 = 0, Value(4-1,y2), Sum_of_triads(y2,ys2), Value(4+ys2,ys0), Value(6+ys0,s)

New backtracking to (9b).

x2=4, s2=ys0

(10c) ← 4 > 0, NonTriad(4), Value(4-1,y2), Sum_of_triads(y2,ys0), Value(6+ys0,s)
n2=4

(11c) ← Value(4 Mod 3,rem2), rem2 ≠ 0, Value(4-1,y2), Sum_of_triads(y2,ys0), Value(6+ys0,s)
rem2=1

(12c) ← 1 ≠ 0, Value(4-1,y2), Sum_of_triads(y2,ys0), Value(6+ys0,s)
y2=3

(13c) ← Sum_of_triads(3,ys0), Value(6+ys0,s)
x3=3, s3=ys0

(14c) ← 3 > 0, Triad(3), Value(3-1,y3), Sum_of_triads(y3,ys3), Value(3+ys3,ys0), Value(6+ys0,s)
n3=3

(15c) ← Value(3 Mod 3,rem3), rem3 = 0, Value(3-1,y3), Sum_of_triads(y3,ys3), Value(3+ys3,ys0), Value(6+ys0,s)
rem3=0

(16c) ← 0 = 0, Value(3-1,y3), Sum_of_triads(y3,ys3), Value(3+ys3,ys0), Value(6+ys0,s)
y3=2

(17c) ← Sum_of_triads(2,ys3), Value(3+ys3,ys0), Value(6+ys0,s)
x4=2, s4=ys3

(18c) ← 2 > 0, Triad(2), Value(2-1,y4), Sum_of_triads(y4,ys4), Value(2+ys4,s3), Value(3+ys3,s0), Value(6+ys0,s)
n4=2

(19c) ← Value(2 Mod 3,rem4), rem4 = 0, Value(2-1,y4), Sum_of_triads(y4,ys4), Value(2+ys4,ys3), Value(3+ys3,ys0), Value(6+ys0,s)

rem4 = 2

(20c) ← 2 = 0, Value(2-1,y4) ,Sum_of_triads(y4,ys4), Value(2+ys4,ys3), Value(3+ys3,ys0), Value(6+ys0,s)

Stop again! This time we will have to backtrack to (17c).

x4=2, s4=ys3

(18d) ← 2 > 0, NonTriad(2), Value(2-1,y4), Sum_of_triads(y4,ys3), Value(3+ys3,ys0), Value(6+ys0,s)

n4=2

(19d) ← Value(2 Mod 3,rem4), rem4 ≠ 0, Value(2-1,y4), Sum_of_triads(y4,ys3), Value(3+ys3,ys0), Value(6+ys0,s)

rem4=2

(20d) ← 2 ≠ 0, Value(2-1 ,y4), Sum_of_triads(y4,ys3), Value(3+ys3,ys0), Value(6+ys0,s)

y4=1

(21d) ← Sum_of_triads(1,ys3), Value(3+ys3,ys0), Value(6+ys0,s)

x5=1, s5=ys3

(22d) ← 1 > 0, Triad(1), Value(1-1,y5), Sum_of_triads(y5,ys5), Value(1+ys5,ys3), Value(3+ys3,ys0), Value(6+ys0,s)

n5=1

(23d) ← Value(1 Mod 3,rem5),rem5 = 0,Value(1-1,y5), Sum_of_triads(y5,ys5), Value(1+ys5, ys3), Value(3+ys3, ys0), Value(6+ys0, s)

rem5 = 1

(24d) ← 1 = 0, Value(1-1,y5), Sum_of_triads(y5,ys5), Value(1+ys5,ys3), Value(3+ys3,ys0), Value(6+ys0,s)

Time to backtrack to (21d).

x5=1, s5=ys3

(22e) ← 1 > 0, NonTriad(1), Value(1-1,y5), Sum_of_triads(y5,ys3), Value(3+ys3,ys0), Value(6+ys0,s)

n5=1

(23e) ← Value(1 Mod 3,rem5), rem5 ≠ 0, Value(1-1,y5), Sum_of_triads(y5,ys3), Value(3+ys3,ys0), Value(6+ys0,s)

rem5=1

(24e) ← 1 ≠ 0, Value(1-1,y5), Sum_of_triads(y5,ys3), Value(3+ys3,ys0), Value(6+ys0,s)

y5=0

(25e) ← Sum_of_triads(0,ys3), Value(3+ys3,ys0), Value(6+ys0,s)
ys3=0

(26e) ← Value(3+0,ys0), Value(6+ys0,s)
ys0=3

(27e) ← Value(6+3,s)
s = 9

(28e) ←

B.2 Trace of Knights_tour

The program:

```
Knights_tour(1,S(1,1),n) ←
Knights_tour(2,Route(square,previous_square),n) ←
        Knights_tour(1,previous_square,n),
        Possible_step(previous_square,square,n)
Knights_tour(visited,Route(square,Route(previous_square,itinerary)),n) ←
        Value(visited - 1,previously_visited),
        Knights_tour(previously_visited,Route(previous_square,itinerary),n),
        Possible_step(previous_square,square,n),
        Unvisited(square,itinerary)
/* It is possible to advance to a square if it is a permitted advance for the knight
        and the square is within the limits of the board. */
Possible_step(previous_square,square,n) ←
        Step(previous_square,square),
        Within(square,n)
/* The valid advances of the knight */
Step(S(row,col),S(row1,col1)) ←
        Value(row-2,row1), Value(col-1,col1);
        Value(row-2,row1), Value(col+1,col1);
        Value(row-1,row1), Value(col+2,col1);
        Value(row+1,row1), Value(col+2,col1);
        Value(row+2,row1), Value(col+1,col1);
        Value(row+2,row1), Value(col-1,col1);
        Value(row+1,row1), Value(col-2,col1);
        Value(row-1,row1), Value(col-2,col1)
/* The board has squares from 1 upto n */
Within(S(row,col),n) ←
        row > 0, col > 0,
        row ≤ n, col ≤ n
```

```
/* A square is unvisited S(row,col) if every previously visited square of
      the itinerary differs from S(row,col) */
Unvisited(S(row,col),S(row1,col1)) ←
      S(row,col) ≠ S(row1,col1)
Unvisited(square,Route(square1,itinerary)) ←
      square ≠ square1,
      Unvisited(square,itinerary)
```

Let us study how the knight can make a tour of five advances on a board of size 4 x 4 squares. We choose not to include all the resolution steps, though. We will look only at those resolution steps where we resolve against relations which we find interesting. We omit all steps that involve an evaluation of the relation Value, for instance. In the same way we will leave out the evaluation of the relation Unvisited.

(1a) ← Knights_tour(5,route,4)

route = Route(sq0,Route(t0,re0))

(2a) ← Knights_tour(4,Route(t0,re0),4), Possible_step(t0,sq0,4), Unvisited(sq0,re0)

re0 = Route(t1,re1),sq1 = t0

(3a) ← Knights_tour(3,Route(t1,re1),4), Possible_step(t1,t0,4), Unvisited(t0,re1), Possible_step(t0,sq0,4), Unvisited(sq0,Route(t1,re1))

re1 = Route(t2,re2), sq1 = t1

(4a) ← Knights_tour(2,Route(t2,re2),4), Possible_step(t2,t1,4), Unvisited(t1,re2), Possible_step(t1,t0,4), Unvisited(t0,Route(t2,re2)), Possible_step(t0,sq0,4), Unvisited(sq0,Route(t1,Route(t2,re2)))

re2 = t3, sq2 = t2

(5a) ← Knights_tour(1,t3,4), Possible_step(t3,t2,4), Possible_step(t2,t1,4), Unvisited(t1,t3), Possible_step(t1,t0,4), Unvisited(t0,Route(t2,t3)), Possible_step(t0,sq0,4), Unvisited(sq0,Route(t1,Route(t2,t3)))

t3 = S(1,1)

(6a) ← Possible_step(S(1,1),t2,4), Possible_step(t2,t1,4), Unvisited(t1,S(1,1)), Possible_step(t1,t0,4), Unvisited(t0,Route(t2,S(1,1))), Possible_step(t0,sq0,4), Unvisited(sq0,Route(t1,Route(t2,S(1,1))))

(7a) ← Step(S(1,1),t2), Within(t2,4), Possible_step(t2,t1,4), Unvisited(t1,S(1,1)), Possible_step(t1,t0,4), Unvisited(t0,Route(t2,S(1,1))), Possible_step(t0,sq0,4), Unvisited(sq0,Route(t1,Route(t2,S(1,1))))

t2 = S(-1,0)

Possible_step generates the various moves the knight is permitted to make. That these are kept within the limits of the board is controlled by the relation Within. If the new square is within the board we backtrack to Possible_step which gives a new alternative.

(8a) ← Within(S(-1,0),4), Possible_step(S(-1,0),t1,4), Unvisited(t1,S(1,1)), Possible_step(t1,t0,4), Unvisited(t0,Route(S(-1,0),S(1,1))), Possible_step(t0,sq0,4), Unvisited(sq0,Route(t1,Route(S(-1,0),S(1,1))))

t2 = S(-1,2)

(8b) ← Within(S(-1,2),4), Possible_step(S(-1,2),t1,4), Unvisited(t1,S(1,1)), Possible_step(t1,t0,4), Unvisited(t0,Route(S(-1,2),S(1,1))), Possible_step(t0,sq0,4), Unvisited(sq0,Route(t1,Route(S(-1,2),S(1,1))))

t2 = S(0,3)

(8c) ← Within(S(0,3),4), Possible_step(S(0,3),t1,4), Unvisited(t1,S(1,1)), Possible_step(t1,t0,4), Unvisited(t0,Route(S(0,3),S(1,1))), Possible_step(t0,sq0,4), Unvisited(sq0,Route(t1,Route(S(0,3),S(1,1))))

t2 = S(2,3)

(8d) ← Within(S(2,3),4), Possible_step(2,3),t1,4), Unvisited(t1,S(1,1)), Possible_step(t1,t0,4), Unvisited(t0,Route(S(2,3),S(1,1))), Possible_step(t0,sq0,4), Unvisited(sq0,Route(t1,Route(S(2,3),S(1,1))))

(9d) ← Possible_step(S(2,3),t1,4), Unvisited(t1,S(1,1)), Possible_step(t1,t0,4), Unvisited(t0,Route(S(2,3),S(1,1))), Possible_step(t0,sq0,4), Unvisited(sq0,Route(t1,Route(S(2,3),S(1,1))))

(10d) ← Advance(S(2,3),t1), Within(t1,4), Unvisited(t1,S(1,1)), Possible_step(t1,t0,4), Unvisited(t0,Route(S(2,3),S(1,1))), Possible_step(t0,sq0,4), Unvisited(sq0,Route(t1,Route(S(2,3),S(1,1))))

t1 = S(0,2)

(11d) ← Within(S(0,2),4), Unvisited(S(0,2),S(1,1)), Possible_step(S(0,2),t0,4), Unvisited(t0,Route(S(2,3),S(1,1))), Possible_step(t0,sq0,4), Unvisited(sq0,Route(S(0,2),Route(S(2,3),S(1,1))))

t1 = S(0,4)

(11e) ← Within(S(0,4),4), Unvisited(S(0,4),S(1,1)), Possible_step(S(0,4),t0,4), Unvisited(t0,Route(S(2,3),S(1,1))), Possible_step(t0,sq0,4), Unvisited(sq0,Route(S(0,4),Route(S(2,3),S(1,1))))

t1 = S(1,5)

(11f) ← Within(S(1,5),4), Unvisited(S(1,5),S(1,1)), Possible_step(S(1,5),t0,4), Unvisited(t0,Route(S(2,3),S(1,1))), Possible_step(t0,sq0,4), Unvisited(sq0,Route(S(1,5),Route(S(2,3),S(1,1))))

t1 = S(3,5)

(11g) ← Within(S(3,5),4), Unvisited(S(3,5),S(1,1)), Possible_step(S(3,5),t0,4), Unvisited(t0,Route(S(2,3),S(1,1))), Possible_step(t0,sq0,4), Unvisited(sq0,Route(S(3,5),Route(S(2,3),S(1,1))))

t1 = S(4,4)

(11h) ← Within(S(4,4),4), Unvisited(S(4,4),S(1,1)), Possible_step(S(4,4),t0,4), Unvisited(t0,Route(S(2,3),S(1,1))), Possible_step(t0,sq0,4), Unvisited(sq0,Route(S(4,4),Route(S(2,3),S(1,1))))

(12h) ← Unvisited(S(4,4),S(1,1)), Possible_step(S(4,4),t0,4), Unvisited(t0,Route(S(2,3),S(1,1))), Possible_step(t0,sq0,4), Unvisited(sq0,Route(S(4,4),Route(S(2,3),S(1,1))))

(13h) ← Possible_step(S(4,4),t0,4), Unvisited(t0,Route(S(2,3),S(1,1))), Possible_step(t0,sq0,4), Unvisited(sq0,Route(S(4,4),Route(S(2,3),S(1,1))))

(14h) ← Advance(S(4,4),t0), Within(t0,4), Unvisited(t0,Route(S(2,3),S(1,1))), Possible_step(t0,sq0,4), Unvisited(sq0,Route(S(4,4),Route(S(2,3),S(1,1))))

t0 = S(2,3)

(15h) ← Within(S(2,3),4), Unvisited(S(2,3),Route(S(2,3),S(1,1))), Possible_step(S(2,3),sq0,4), Unvisited(sq0,Route(S(4,4),Route(S(2,3),S(1,1))))

(16h) ← Unvisited(S(2,3),Route(S(2,3),S(1,1))), Possible_step(S(2,3),sq0,4), Unvisited(sq0,Route(S(4,4),Route(S(2,3),S(1,1))))

t0 = S(2,5)

(15i) ← Within(S(2,5),4), Unvisited(S(2,5),Route(S(2,3),S(1,1))), Possible_step(S(2,5),sq0,4), Unvisited(sq0,Route(S(4,4),Route(S(2,3),S(1,1))))

t0 = S(3,6)

(15j) ← Within(S(3,6),4), Unvisited(S(3,6),Route(S(2,3),S(1,1))), Possible_step(S(3,6),sq0,4), Unvisited(sq0,Route(S(4,4),Route(S(2,3),S(1,1))))

t0 = S(5,6)

(15k) ← Within(S(5,6),4), Unvisited(S(5,6),Route(S(2,3),S(1,1))), Possible_step(S(5,6),sq0,4), Unvisited(sq0,Route(S(4,4),Route(S(2,3),S(1,1))))

t0 = S(6,5)

(15l) ← Within(S(6,5),4), Unvisited(S(6,5),Route(S(2,3),S(1,1))), Possible_step(S(6,5),sq0,4), Unvisited(sq0,Route(S(4,4),Route(S(2,3),S(1,1))))

t0 = S(6,3)

(15m) ← Within(S(6,3),4), Unvisited(S(6,3),Route(S(2,3),S(1,1))), Possible_step(S(6,3),sq0,4), Unvisited(sq0,Route(S(4,4),Route(S(2,3),S(1,1))))

t0 = S(5,2)

(15n) ← Within(S(5,2),4), Unvisited(S(5,2),Route(S(2,3),S(1,1))), Possible_step(S(5,2),sq0,4),

Unvisited(sq0,Route(S(4,4),Route(S(2,3),S(1,1))))

t0 = S(3,2)

(15o) ← Within(S(3,2),4), Unvisited(S(3,2),Route(S(2,3),S(1,1))),
Possible_step(S(3,2),sq0,4),
Unvisited(sq0,Route(S(4,4),Route(S(2,3),S(1,1))))

(16o) ← Unvisited(S(3,2),Route(S(2,3),S(1,1))), Possible_step(S(3,2),sq0,4),
Unvisited(sq0,Route(S(4,4),Route(S(2,3),S(1,1))))

(17o) ← Possible_step(S(3,2),sq0,4),
Unvisited(sq0,Route(S(4,4),Route(S(2,3),S(1,1))))

(18o) ← Advance(S(3,2),sq0), Within(sq0,4),
Unvisited(sq0,Route(S(4,4),Route(S(2,3),S(1,1))))

sq0 = S(1,1)

(19o) ← Within(S(1,1),4),
Unvisited(S(1,1),Route(S(4,4),Route(S(2,3),S(1,1))))

(20o) ← Unvisited(S(1,1),Route(S(4,4),Route(S(2,3),S(1,1))))

sq0 = S(1,3)

(19p) ← Within(S(1,3),4),
Unvisited(S(1,3),Route(S(4,4),Route(S(2,3),S(1,1))))

(20p) ← Unvisited(S(1,3),Route(S(4,4),Route(S(2,3),S(1,1))))

(21p) ←

We get the resulting route from the substitutions route = Route(sq0,Route(t0,re0)), re0 = Route(t1,re1), re1 = Route(t2,re2), re2 = t3, t3 = S(1,1), t2 = S(2,3), t1 = S(4,4), t0 = S(3,2) and sq0 = S(1,3).

That is,
route = Route(S(1,3),Route(S(3,2),Route(S(4,4),Route(S(2,3),S(1,1)))))

Appendix C. Transformation Rules

$$A \to B \quad \Leftrightarrow \quad \neg(A \wedge \neg B)$$

$$A \to B \quad \Leftrightarrow \quad \neg A \vee B$$

$$A \vee B \to C \quad \Leftrightarrow \quad (A \to C) \wedge (B \to C)$$

$$A \leftrightarrow B \quad \Leftrightarrow \quad (A \to B) \wedge (B \to A)$$

$$A \leftrightarrow B \quad \Leftrightarrow \quad (A \wedge B) \vee (\neg A \wedge \neg B)$$

$$\neg(A \wedge B) \quad \Leftrightarrow \quad \neg A \vee \neg B$$

$$\neg(A \vee B) \quad \Leftrightarrow \quad \neg A \wedge \neg B$$

$$\neg(\neg A) \quad \Leftrightarrow \quad A$$

$$\neg \forall x P(x) \quad \Leftrightarrow \quad \exists x \neg P(x)$$

$$\neg \exists x P(x) \quad \Leftrightarrow \quad \forall x \neg P(x)$$

$$A \vee (B \wedge C) \quad \Leftrightarrow \quad (A \vee B) \wedge (A \vee C)$$

$$A \vee \exists x\, B \quad \Leftrightarrow \quad \exists x\, (A \vee B)^{1}$$

$$A \vee \forall x\, B \quad \Leftrightarrow \quad \forall x\, (A \vee B)^{2}$$

$$\exists x A(x) \to B \quad \Leftrightarrow \quad \forall x\, (A(x) \to B)^{3}$$

---

[1] On condition that x does not occur in A

[2] On the same condition as above

[3] On condition that x does not occur in B

$$A \to \exists x B(x) \quad \Leftrightarrow \quad \exists x\,(A \to B(x))^4$$

$$A \vee B \quad \Leftrightarrow \quad B \vee A$$

$$\forall x\,(A \wedge B) \quad \Leftrightarrow \quad \forall x\,A \wedge \forall x\,B$$

$\forall x_1 \forall x_2 \ldots \forall x_n\, \exists y\, P(y)$ is exchanged for $\forall x_1 \forall x_2 \ldots \forall x_n\, P(\,G(x_1, x_2, \ldots, x_n))$

[4] On condition that x does not occur unbound in A

Appendix D. Built-in Predicates

The built-in constructions in our Prolog system are assembled in alphabetical order below. Most of them have been defined previously.

All_answers(x,P(x),xl)	Searches all substitutions for x that satisfy P(x) and collects them in the list xl.
Assert(clause)	Inserts clause clause into the database.
Cases (P_1: Q_1, P_2: Q_2, ...,True: Q_n)	The first of the discriminating conditions P_i directs the evaluation to the expression after ":". If no P_i is satisfied Q_n is evaluated.
Close(file)	Closes the file file.
False	The predicate always fails, backtracking is started.
In_char(x)	x is the ASCII code for next character in the input stream.
Integer(x)	x is an integer.
In_term(x)	Reads the term x.
Input_stream(file)	File file becomes the current input stream.
Lineshift	Writes the contents in the output buffer in current file and lineshifts after printing.
Not P	The question P is not provable.
Out_char(x)	Writes a character, x is the ASCII code for the character.
Output(file)	Directs current output to the file file.
Out_term(x)	Writes the term x.
Retract(clause)	Retracts the first clause which matches clause.
Sorted_answers(x,P(x),xl)	Searches for all substitutions for x that satisfy P(x) and assembles them in the list xl. xl contains no duplicates and is sorted, compare All_answers.
Structure(s,c,al)	A relation between a structure s and the constructor of the structure c as well as a list of the arguments of the structure al.
Tab(n)	Writes n spaces in the current output.
Text_Ascii(x,al)	A relation between a constant or an integer x and a list of the corresponding ASCII code al.

True	The predicate always succeeds.
TTYEmpty_buffer	Writes the contents of the output buffer on the terminal.
Value(expr,res)	res is the numerical value of the numerical expression expr.
Variable(x)	x is a variable.
x^P(x)	There exists a x such that P(x) is true. The construction is only used in All_answers and Sorted_answers.
x = y	x and y can be unified.
x ≠ y	x and y cannot be unified.
x < y	x is less than y.
x > y	x is greater than y.
x ≤ y	x is less than or equal to y.
x ≥ y	x is greater than or equal to y.
x + y	The sum of x and y.
x − y	The difference of x and y.
x * y	The product of x and y.
x / y	The quotient between x and y.
x Mod y	The remainder in an integral division of x by y.
|	Cuts away branches of the proof tree.

Appendix E. ASCII Codes

ASCII-code	character
31	Carriage return
32	Space
33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	−
46	.
47	/
48–57	0–9
58	:
59	;
60	<
61	=
62	>
63	?
64	@
65–90	A–Z
91	[
92	\
93	]
94	^
95	_

96	`
97–122	a–z
123	{
124	|
125	}
126	~

Index