

SYMBOLIC COMPUTATION

Helder Coelho José C.Cotta

Prolog by Example

How to Learn, Teach and Use It



Springer-Verlag

SYMBOLIC COMPUTATION

Artificial Intelligence

Managing Editor: D. W. Loveland

Editors: S. Amarel A. Biermann L. Bolc A. Bundy
H. Gallaire P. Hayes A. Joshi D. Lenat A. Mackworth
E. Sandewall J. Siekmann W. Wahlster

- N.J. Nilsson: Principles of Artificial Intelligence. XV, 476 pages, 139 figs., 1982
- J. H. Siekmann, G. Wrightson (Eds.): Automation of Reasoning 1. Classical Papers on Computational Logic 1957–1966. XXII, 525 pages, 1983
- J. H. Siekmann, G. Wrightson (Eds.): Automation of Reasoning 2. Classical Papers on Computational Logic 1967–1970. XXII, 638 pages, 1983
- L. Bolc (Ed.): The Design of Interpreters, Compilers, and Editors for Augmented Transition Networks. XI, 214 pages, 72 figs., 1983
- M.M. Botvinnik: Computers in Chess. Solving Inexact Search Problems. With contributions by A.I. Reznitsky, B.M. Stilman, M.A. Tsfasman, A.D. Yudin. Translated from the Russian by A.A. Brown. XIV, 158 pages, 48 figs., 1984
- L. Bolc (Ed.): Natural Language Communication with Pictorial Information Systems. VII, 327 pages, 67 figs., 1984
- R.S. Michalski, J.G. Carbonell, T.M. Mitchell (Eds.): Machine Learning. An Artificial Intelligence Approach. XI, 572 pages, 1984
- A. Bundy (Ed.): Catalogue of Artificial Intelligence Tools. Second, revised edition, IV, 168 pages, 1986
- C. Blume, W. Jakob: Programming Languages for Industrial Robots. XIII, 376 pages, 145 figs., 1986
- J.W. Lloyd: Foundations of Logic Programming. Second, extended edition, XII, 212 pages, 1987
- L. Bolc (Ed.): Computational Models of Learning. IX, 208 pages, 34 figs., 1987
- L. Bolc (Ed.): Natural Language Parsing Systems. XVIII, 367 pages, 151 figs., 1987
- N. Cercone, G. McCalla (Eds.): The Knowledge Frontier. Essays in the Representation of Knowledge. XXXV, 512 pages, 93 figs., 1987
- G. Rayna: REDUCE. Software for Algebraic Computation. IX, 329 pages. 1987
- L. Bolc, M.J. Coombs (Eds.): Expert System Applications. IX, 471 pages, 84 figs., 1988
- D.D. McDonald, L. Bolc (Eds.): Natural Language Generation Systems. XI, 389 pages. 84 figs., 1988
- C.-H. Tzeng: A Theory of Heuristic Information in Game-Tree Search. X, 107 pages, 22 figs., 1988

Helder Coelho José C. Cotta

Prolog by Example

How to Learn, Teach and Use It

With 68 Figures



Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo

Helder Coelho

Research Scientist at LNEC

Associate Professor at the Faculty of Economics
of the Technical University of Lisbon**José Carlos Cotta**Scientific Officer of DGXIII in Brussels
for the ESPRIT Programme

Laboratorio Nacional de Engenharia Civil
101, Av. do Brasil, P-1799 Lisbon, Portugal

ISBN-13:978-3-642-83215-4 e-ISBN-13:978-3-642-83213-0

DOI: 10.1007/978-3-642-83213-0

Library of Congress Cataloging-in-Publication Data

Coelho, Helder.

Prolog by example : how to learn, teach, and use it / H. Coelho, J.C. Cotta.

p. cm. -- (Symbolic computation. Artificial intelligence)

Bibliography: p.

Includes index.

ISBN-13:978-3-642-83215-4 (U.S.)

1. Prolog (Computer program language) I. Cotta, José Carlos.

II. Title. III. Series. QA76.73.P76064 1988

006.3--dc 19 87-30522 CIP

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in other ways, and storage in data banks. Duplication of this publication or parts thereof is only permitted under the provisions of the German Copyright Law of September 9, 1965, in its version of June 24, 1985, and a copyright fee must always be paid. Violations fall under the prosecution act of the German Copyright Law.

© Springer-Verlag Berlin Heidelberg 1988
Softcover reprint of the hardcover 1st edition 1988

The use of registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Table of Contents

Prolog(ue)	VII
Preface	IX
Chapter 1 Teaching Prolog	1
Chapter 2 One or Two Things About Prolog	11
Chapter 3 Using Prolog	21
Chapter 4 Proving with Prolog	71
Chapter 5 Doing Arithmetic with Prolog	99
Chapter 6 Doing Algebra with Prolog	119
Chapter 7 Playing with Prolog	123
Chapter 8 Searching with Prolog	147
Chapter 9 Learning with Prolog	161
Chapter 10 Modeling with Prolog	173
Chapter 11 Designing with Prolog	189
Chapter 12 Drawing with Prolog	195
Chapter 13 Planning with Prolog	219
Chapter 14 Seeing with Prolog	243
Chapter 15 Engineering Grammars with Prolog	247
Chapter 16 Constructing Personal Databases with Prolog	281
Chapter 17 Text Formatting with Prolog	305

VI Table of Contents

Chapter 18	Management with Prolog	315
Chapter 19	Building Up with Prolog	333
Acknowledgments		364
Appendix 1	Prolog Implementations	365
Appendix 2	Commercial Products	371
Appendix 3	Selection of Some Historical Prolog Applications	373
References		376

Prolog(ue)

Although I am an Artificial Intelligence (AI) specialist, this book is not strictly an AI work; yet, it has roots in Computer Science (or Informatics) where AI plays the role of a computing methodology.

The motivation for writing this book began years ago. At that time, I was interested in studying the uses of AI tools in civil engineering and architecture environments. My research interests evolved into something else over the years, but Prolog still played an important role: the study of a tool (programming language) was superceded by the study of (natural) language. Doings, not doings and undoings, i.e. reasoning and writing in logic, became part of my experience. Furthermore, the training in clear thinking and problem specification that I received proved a good investment for the future.

The idea behind this book was brought to my mind in those days as I started to learn how to declare my ideas in Prolog, at the University of Edinburgh. My teachers, Luís Moniz Pereira, Robert Kowalski and David Warren, gave me some examples and described to me the machinery behind how to read and interpret clauses, and how knowledge representation in logic was so clear and easy to construct. In those years of apprenticeship, I also had the privilege of meeting Alain Colmerauer at the University of Marseille.

The teachings of these men, the logic way of programming, led me into a world of beauty, ruled by simple and powerful concepts far beyond those of standard computer programming.

Some years later, and after designing and writing several systems and small programs, I realized that it would be interesting to record my initiation into the mysteries of logic by collecting a set of good examples. That was the time when the book “How to solve it with Prolog”, published by Laboratório Nacional de Engenharia Civil (LNEC) in 1979, was prepared by myself with the collaboration of my associate José Carlos Cotta, and with some help from Luís Moniz Pereira. This book consisted of problem statements and logic programs, without annotation or any further explanation of the ways in which those programs (clauses) were in fact written. The idea was to explore the expressive power of Prolog as a means of communicating with computers. But, readers claimed that reasoning behind clauses was, in some examples, too obscure, namely when recursion was present.

During the following years five editions were printed. Under heavy demands of eager readers all over the world, the book became a best-seller within the logic programming community, despite its weakness and lack of “semantic sugar”. Meanwhile, I received criticism and encouragement from my friend Ronald Lee

VIII Prolog(ue)

who pushed me in rewriting all the material. Also, I profited from my teaching experience of two Artificial Intelligence courses at the University of Lisbon in 1981-1982 and 1982-1983, from an advanced Prolog lecture course I held at the Technical University of Vienna, and from a series of talks I gave in Portugal and around the world on logic programming style, tools and methods.

It seems now that the primitive idea of a book on practical Prolog programming, directed to those becoming initiated, is clearer on goals, and in particular on the teaching style, and on learning materials and methods. The nature of programming is changing. These changes will accelerate as improved software development practices and more sophisticated development tools and environments are produced. A software system consists of a number of components. The construction of software systems requires that each component be built specifically for each system, but there is a growing need for reuse of these components. The direction is now towards the identification of basic software building blocks. So, readers may look to the program pieces published in this book as those blocks, and their job (of programming) is one of engineering the various pieces of description. That is why this book is about using Prolog by example.

The motivation set up by the Japanese R & D program on fifth generation computers obliged me to plan a new structure for this book, taking into account a larger number of examples/exercises fully referenced throughout the book, assembled in new chapters and sent to me by several colleagues around the world. The list of Prolog implementations, supplied with an efficiency (LIPS) comparison of some of them, and the selection of some historical Prolog applications were updated. A list of Prolog commercial products was added.

I hope my book may constitute a challenge to your reason, helping people working in Computing Sciences to change programming habits and styles, from a procedural to a declarative or descriptive way of knowledge.

Helder Coelho

Preface

Prolog has a declarative style. A predicate definition includes both the input and output parameters, and it allows a programmer to define a desired result without being concerned about the detailed instructions of how it is to be computed. Such a declarative language offers a solution to the software crisis, because it is shorter and more concise, more powerful and understandable than present-day languages. Logic highlights novel aspects of programming, namely using the same program to compute a relation and its inverse, and supporting deductive retrieval of information.

This is a book about using Prolog. Its real point is the examples introduced from Chapter 3 onwards, and so a Prolog programmer does not need to read Chapters 1 and 2, which are oriented more to teachers and to students, respectively. The book is recommended for introductory and advanced university courses, where students may need to remember the basics about logic programming and Prolog, before starting doing. Chapters 1 and 2 were also kept for the sake of unity of the whole material.

In Chapter 1 a teaching strategy is explained based on the key concepts of Prolog which are novel aspects of programming. Prolog is enhanced as a computer programming language used for solving problems that involve objects and the relationships between objects. This chapter provides a pedagogical tour of prescriptions for the organization of Prolog programs, by pointing out the main drawbacks novices may encounter.

In Chapter 2, we introduce and advocate the logical programming approach for software development in Prolog, as the one most appropriate to the implementation of ‘intelligent’ systems. Its expressive power allows users to articulate more of the nuances and details of thought processes by programming by assertion and query. And we say something about Prolog syntax and how to interact with the top level. Once a beginner knows how to consult (and reconsult) files and make queries, he or she is ready to learn Prolog by reading and using the example programs.

The main purpose of Chapters 3 to 19 is to serve as a guide book for software development in Prolog, helping the approach of teaching Prolog by example. The organization of a collection of programs for small problems and exercises is based upon specific application areas. This option may help programmers (aspiring knowledge engineers) to use those pieces of programming as building blocks for more complex systems. The programs are annotated and have been fully tested and adapted to the Prolog version running on the DEC-10 and VAX systems. The last chapters do not attempt to guide the novice programmer. They aim to back up

intermediate level students and advanced Prolog users. In particular, this book can be used either as a manual for the beginner or as an aid and source of ideas by the more experienced users. Those programmers with a background in computing wishing to learn Prolog will find the treatment simple and easy to follow, with useful examples of common situations. And, it is also a guide for knowledge engineers wishing to know more about the techniques and the art of building up complex systems by Prolog programming.

Over the past fifteen years there has been a great deal of research work in logic programming (Coelho 1985b). The programming language Prolog, based on predicate calculus, was developed, and several interpreters and compilers were written for the main computer systems available on the commercial market (see Appendices 1 and 2). A number of practical working systems have also been implemented from 1976 on (see Appendix 3).

December 1987

Helder Coelho
José C. Cotta

Chapter 1 Teaching Prolog

Discussion of the ways of teaching and learning Prolog is a very important matter. There are several reasons for this. Most instructors (in common with most computer scientists) have had very little experience with Prolog and there is more than one point of view to consider. In general, programming language texts rarely discuss how a language should be taught. Moreover, the discussion is far from being clarified and several tactics were adopted after Prolog was commercialized. Several projects on computer aided instruction in Prolog are in progress (one in LNEC in Lisbon) for the so-called benefit of programmers. But, it seems the intentions of the lecturers do not match the desires of students.

1.1 The History of Prolog

Logic programming is the use of the clausal form of first order logic as a practical computer programming language (Andreka et al. 1976; Bowen 1976). It is based upon the following **thesis**: predicate logic (calculus) is a useful language for representing knowledge. It is useful for stating problems and it is useful for representing the pragmatic information for effective problem solving (Kowalski 1979). The clausal form suggests an important **advantage**: it is a canonical form for the resolution principle (Robinson 1965), which is a complete and sound inference system consisting of only one inference rule (the resolution principle). This advantage was materialized into a programming language called **Pro**(gramming) **Log**(ic).

Around 1972, Prolog made its first appearance due to Colmerauer and Roussel. But it was in 1973 that the first version in FORTRAN IV (Battani & Meloni 1973) allowed Colmerauer and Kowalski to start campaigning for the revolutionary idea that logic could be used to express information in a computer, to present problems to a computer, and to solve these problems, i.e. the idea of logic as a programming language. In brief, these uses were abstracted in two main equations:

`program = set of axioms`

`computation = proof of a statement from the axioms`

The primitive idea was a direct outgrowth of earlier work in logic, automatic theorem-proving and artificial intelligence, done in the 1930s, 1950s and 1960s.

During the 1930s, predicate calculus, the notation for concepts invented by Frege, was developed into a type of computation, by discovering algorithms in

which the processes of deduction were captured in a systematic way. Herbrand proposed several versions of the proof procedure.

For some time logic was adopted as a specification or declarative language in Computer Science. The minimal model characteristic of the Horn clause logic was discovered and Horn sets found to be not decidable. But the breakthrough occurred when Horn logic was considered seriously with a practical point of view in mind.

Prawitz, in 1960, introduced the unification notion to automated theorem proving, and used it in an informal way. Robinson made the notion precise by discovering in 1965 the resolution principle, the way of involving the unification with the proof system. The work on resolution went on with a major contribution by Luckham and Loveland, who independently discovered linear resolution. Kowalski and Kuehner analyzed the complexity of a form of linear resolution in their paper on SL-resolution, which is a modification of model elimination, due to Loveland. Later on, Boyer and Moore invented structure sharing for speeding up the resolution process. But it was due to Colmerauer's group that the idea of logic programming was born as a Horn clause theorem prover, called Prolog. The interpreter embodied Kowalski's procedural interpretation of linear Horn clause resolution systems.

Kowalski proved that logic had a procedural interpretation, which made it very effective as a programming language: clauses which do not contain a head (the left-hand side) are regarded as queries requiring solutions, and actual execution starts as soon as a clause is input. Later on, Apt, van Emden and Kowalski provided the Prolog language with an operational, fixpoint, and model-theoretic semantics.

During the whole of the 1970s the concept of logic as a uniform language for data, programs, queries, views and integrity constraints gained theoretical and practical potential, mainly due to the Prolog groups established around Europe. Symbolic logic was recognized as the most suitable formalism for problem specification, and the task of specification and verification was made easier when the same formalism was adopted. Program transformation preserving correctness became easier when done in the same formalism. In databases, logic programs were regarded as a generalization of relational databases. The data description language of a relational database was proved equivalent to logic programs without conditional axioms. Also, Horn clause logic was shown to be adequate both as a query language and as a language for describing integrity constraints. These achievements were mainly due to Reiter, Gallaire and Nicolas.

In 1977, the first Prolog compiler was born, in Edinburgh (with LNEC collaboration), simultaneously with the best interpreter to date, the DEC-10 one, mainly due to Warren. At the same time, Warren, Luis Moniz Pereira and Fernando Pereira produced the first attack on Lisp by presenting a paper comparing Lisp and Prolog performances in an International Conference taking place in the USA. This controversy is still in progress, in spite of proofs made several times. One by O'Keefe, in 1983, tried to put an end to it, but later on Kowalski (1985 a and b) was again called to introduce new arguments.

In 1978, Luis Moniz Pereira and Monteiro examined the features of logic which make logic programming especially suitable for parallel and co-routined

modes of processing. These issues became very popular in 1982, when the Japanese project started. During this year Dahl published her thesis on the use of Prolog for natural language understanding (Spanish), following Colmerauer's ideas on metamorphosis grammars and on an interesting subset of French.

By the end of 1979, Colmerauer's group again had made the first implementation of Prolog for microcomputers. And, Colmerauer pushed the Prolog formalization by presenting additional theoretical foundations. He explained what a Prolog without occur check should be, by using a new Prolog interpreter for the unification of infinite trees in a finite time. Coelho pushed Colmerauer's and Dahl's ideas a little bit further forward by applying definite clause grammars to the Portuguese language, and by extending their use for supporting conversations.

Around 1982, Shapiro studied the computational complexity of logic programs. The results revealed similarities between logic programs and alternating Turing machines, and a link between the structural complexity and computational complexity of logic programs, a relation rarely found in practical programming languages (Shapiro 1982). As a side effect, more evidence was produced about the potential of logic programming for parallel machines.

The ideal of logic programming is constrained by the control problems and the negation problem. The first problem was heavily discussed in the 70s by Clark and McCabe, giving way to the IC-Prolog, and by Bruynooghe and Luis Moniz Pereira, giving way to diversified backtracking techniques. The second problem was pushed by Lloyd in 1984, in a remarkable paper (Lloyd 1983), where he proved the soundness and completeness of the negation as failure rule. In pure Prolog, only affirmative knowledge of the form 'If A Then B' could be expressed. Negative knowledge of the form 'If A Then not B' or 'If not A Then B' could not be either handled or stored. In the majority of the current implementations, the cut symbol and fail (non-logical devices) are used to deal with negative knowledge. There is one good argument for accepting only Horn clauses: the combinatorial explosion is slowed by narrowing the selection of the proof process.

1.2 Method

Those having experience in teaching Prolog to undergraduates and postgraduates know about the problems the students face when **learning to write** in Prolog. The majority of these problems come up because Prolog is a declarative language and those students have conventional programming experience in procedural languages. Also, the gap between a reading knowledge and a writing knowledge is much wider than in other languages, and the program compactness makes the reading and learning processes more difficult. Therefore, the **teaching strategy** adopted is of capital importance.

There are several tactics for teaching Prolog depending on the previous experience of the students with other programming languages. Conventional wisdom dictates starting by presenting the declarative viewpoint with examples of database search and list processing. Taking this approach, Bob Welham has been lecturing to those having some previous experience of Prolog, either from an introductory

course, or by having used Prolog a little. Those examples are illustrated after over-viewing syntax and terminology, and are followed by explanations about matching, searching, invoking procedures and recursion. The uses of cut for controlling the execution come afterwards and before discussing problem solving, program specification/verification, relational database handling, English parsing and knowledge engineering. Peter Ross defended a different approach based upon teaching Prolog to undergraduates. He stated that sticking with the conventional approach for any length of time is a mistake, because it works well for introducing the initial syntax (the ideas of clauses and the logical variable), but it promotes complacency. The students ran into trouble when they met examples that needed a procedural interpretation or that depended on backtracking. He suggested starting with recursion examples which are easily written in Prolog without conventional constructs such as IF..THEN..ELSE.. and FOR..=..TO... He also defended the topic of operators as a key item for seducing and motivating newcomers (Ross 1982).

Richard Ennals defended another approach which was tested during the project "Logic as a computer language for children" aged 10–11: "an advance in formal representation can only be done if supported on motivating examples". For him, the initiatory activity must be based upon writing sentences and defining relations (using networks) in order to ask questions. Only after this tactic is carried out may list processing and recursion be tackled. He stated that "a clear description of a subject can be regarded as a specification and run as a program" (Ennals 1982). The areas of difficulty identified by Ennals were the forms of queries, the use of variables, the translation from natural language into logical notation, the use of rules, lists as individuals, and list processing. His **teaching strategy** was based on attacking these difficulties.

Feliks Kluzniak and Stanislaw Szpakowicz (1981) think the usual presentation, stressing the programming-in-logic origin of Prolog and its problem solving explanation, is inadequate to demonstrate that Prolog is competitive with other languages. They defended the notion that the purely logical approach is the best method for introducing programming to non-programmers, but it is less suitable for practising programmers. Their strategy is based upon finding answers to four questions posed by a common programmer who attempts to learn Prolog: 1) how to deal with terms and what to make of unification; 2) how to interpret the multi-clause form of procedures; 3) what is backtracking and how to use it; and, 4) how to interpret and use the cut symbol.

Our experience dictates **another approach to teaching**. We believe Prolog is easily learned when the following **three key aspects** are enhanced during the first lessons:

- **knowledge representation and architecture**, i.e. Prolog syntax, (declarative and operational) semantics and pragmatics in terms of classical logic (the ideas of term, clause/rule, and Prolog variable as compared to equivalent ideas in conventional programming languages), and the art of structuring a view of reality;
- **unification machinery**, i.e. unification as pattern matching plus logical variable;
- **novel aspects of programming**, i.e. computation of relations, symbol manipulation and deductive retrieval of information.

As a matter of fact, these issues interact and cannot be disassociated. The representation of knowledge per se is not sufficient, because the modeling of reality is based upon the ways of building up architectures in layers of abstraction.

By implementing this **strategy**, we may explain a different style of thinking about programming to novices used to other programming languages: how the clausal form of logic, as a practical computer programming language, is different and advanced in what concerns other high level (procedural) programming practices. Therefore, we may achieve a distortion of the mental imagery of programming of the students to make Prolog fit into it, and force them to unlearn certain static ideas. This unlearning process is crucial on account of the wide gap between reading and writing knowledge, algorithmic and non-algorithmic languages, determinism and non-determinism. Such a different style of thinking about programming may be understood when the meaning of a clause is identified as a rule (theorem of first order logic), a goal or query as an instruction, and the meaning of a multi-clause as the possible methods to solve the problem.

Our **strategy** is also based upon the importance of motivating examples. We agree with Ross that a large number of exercises is a good way to bridge the reading/writing gap, and that individual projects and programs support a better class teaching approach.

1.3 Logic by the Clause Approach

Software development in Prolog can be based on “programming by assertion and query” (Robinson 1980) rather than by insertion and search of data structures. In a way, the Prolog programming style is fully declarative: each individual clause (If-Then rule) has a self-contained meaning (basic building block) which can be understood independently of other parts of the program, and a predicate (procedure) definition includes the input and output variables.

The choice of data structures in Prolog is a matter of great importance, and not only for reasons of efficiency. The natural data structure in Prolog is a tree, but sometimes it is natural to represent an assertion by a record. The use of lists, implemented as trees, is not encouraged because they lead to less efficient and less readable programs.

Programs in Prolog are treated as the data in a database. This uniform view of both programs and data (also, knowledge) as items in a high level database is one of the reasons for the elegance of the Prolog programming style. The style of assertive programming may be used as an advantage in developing software very rapidly.

Logic programming style is achieved in full by taking into account **Kowalski's equation** (the main thesis of logic programming)¹:

¹ Several authors do not agree with the designation “equation”, claiming that it is impossible to read the expression from both sides. Therefore, they assert the word “inequation” to classify the expression.

Algorithm	=	Logic	+	Control
specifications				declarations of logical commitment
(properties of data)				(impositions on the order of logic specifications to achieve computational effectiveness)

By separating logic from control, programs reveal a static structure containing the most suitable knowledge to be used, but not the explicit ways of how to explore it. The non-determinism is present, yet restricted by the order of clauses in a program and the order of atoms in each clause. Cuts, providing an If-Then-Else control structure, and other extralogic primitives simulate the programming styles of conventional programming languages.

The form of the above equation is different from the one commonly associated with conventional programming, **Wirth's famous equation:**

$$\text{Algorithms} + \text{Data Structures} = \text{Programs}$$

where programs reveal a dynamic structure.

When comparing both equations, we discover that logic programs are also activated but through a completely different means. In fact, logic programs are triggered by goals or queries, the so-called instructions of conventional programming.

Pure Prolog programs free from side effects (e.g. 'assert' and 'retract') and control machinery (e.g. cut symbol) are all too rare in real-life programming. But, the same occurs with conventional programming, and even with functional programming.

In logic programming, when we do not care too much about execution efficiency, we gain in clarity. But how is it attained? There are twelve **golden rules** for writing universally readable programs:

- 1) avoid the cut (attain non-determinism): functional dependencies are better alternatives to cuts
- 2) don't use 'assert' and 'retract' (explore the logical variable to simulate assignment)
- 3) don't use 'write' (data output is automatic in some Prolog systems)
- 4) adopt, when possible, a binary record representation scheme for unit clauses
- 5) include, by redundancy, the inverses of pre-defined predicates, whenever convenient, to avoid the use of negation
- 6) use infix operators to enhance legibility
- 7) capture knowledge and encode general facts through deduction rules (non-unit clauses)
- 8) place, for the same relation, unit clauses before non-unit ones
- 9) avoid more than two levels of nesting in data structures
- 10) use special predicates to control the types and domains of objects and values to be inserted in the knowledge base
- 11) organize objects, domains and relations into hierarchical structures
- 12) use Prolog in conjunction with different knowledge representation methods (e.g. semantic nets, frames, etc.) in different applications

Otherwise, when **efficiency** (in terms of time and space) is urgently required, the so-called good Prolog programming practice, we may take into account the following restricted **golden rules**:

- 1') use mode declarations (to replace calls to a general unification routine by in-line code) when available (this means constraining some predicate arguments to be only input or output), even when using an interpreter because it is good documentation practice
- 2') avoid lists, unless the data structure is a sequence of variable length, a set or a collection of things to be sorted
- 3') avoid lists when record structures are appropriate
- 4') use cut to provide an If-Then-Else effect
- 5') re-order the clauses and the goals of a query, if possible, so that those easier to satisfy (which will have the fewest number of solutions) are first considered (thus reducing backtracking effort)
- 6') avoid inessential use of the internal database
- 7') substitute recursive definitions, which build structures, by a combination of internal database manipulations and cut-fail sequences

Expert system design suffers from the same illness that attacks conventional system design effort: inexact programs are designed because people are inexact when they try to explain what they do and need. During the design, knowledge architecture is one of the bottlenecks because there is no methodology to structure a mass of information from a string of different sources in a way that makes sense and fits in with the software tools which will be used to build the system.

Knowledge comes in all shapes and sizes, and each piece of knowledge interlocks with other pieces in highly complex, richly structured ways. Items in a knowledge base are linked to concepts, often each item linking individually to its own set of concepts, quite unlike the orderly repetition of items and their relationships in a traditional database.

A **global knowledge base** (KB) is architected into three layers (3-L architecture), each of them composed of a database and a rule base:

- the meta knowledge base (**MKB**)
- the specifications knowledge base (**SKB**)
- the ground knowledge base (**GKB**)

The **meta knowledge base** contains syntactic and semantic knowledge about the integrated model of data and processes to be used, and it consists mainly of rules, both for deduction (knowledge flow) and checking (constraints). These rules designate structures.

The **specifications knowledge base** contains information about particular applications, not necessarily disjoint, especially if those applications are related to different aspects of the same management environment. This information is covered by clusters of connected frames and captures the presence or absence of contradictions in the ground knowledge. Each application models the information about a system or part of it (an organization, or its functional departments, for instance). The specifications designate entities and expressions.

The **ground knowledge base** contains information about instances of the specification sort and, depending on the application, it may include a more or less large amount of extensive data. The naming of those instances is essential for most management purposes (e.g. the identification of the employees of an enterprise) but irrelevant for other purposes (e.g. the simulation of a production line).

The above description of the 3-L architecture provides us with the following **design principles**:

- 1) A 3-L architecture is a tower of declarative self-models, a structural field composed of structures, as elements, and of expressions.
- 2) Top level conditions are structures, held in the MKB as points of reflection (view points), designating other structures.
- 3) Conditions (internal structures) are held within layers, shared by rules and examined in three ways: as individual conditions, representing one chunk of causal knowledge; as parts of an individual rule, representing one cause for triggering that rule; or as common elements of more than one rule, representing common aspects of the pattern of triggering those rules.
- 4) Goals (specifications in our case) are held in the SKB.

Conventional Programming	Logic Programming
primitive unit: instruction	primitive unit: goal, query
type declaration	—
data types	terms
class of only one object	ground term
constant	constant
universal type	(universally quantified) or logical variable
record type	functor
field of a record	argument of a functor
program	set of clauses
procedure	non-unit clause, rule, definite clause, theorem of first-order logic
head of the procedure	head of clause, consequent
body of the procedure	body of the clause, antecedents/conditions
procedure calls	goals
data	unit clause
parameter-passing mechanism (pattern matching)	unification
binding mechanism,	unification
data selectors and	
constructors	
execution mechanism	nondeterministic, goal reduction
binding computation	answer substitution

Fig. 1. Relationships between programming concepts

- 5) Communicative objects (communication processes held within individual conditions) link conditions together with goals to form 'If-Then' rules, and provide the basis for the reasoning chain, i.e. the semantic basis upon which the truth of a condition is determined.
- 6) Interrelated collections of rules (inference network of knowledge structures) are based upon four building blocks: one condition (antecedent) implying one goal (consequent) – the so-called Horn non-unit clause in Prolog, multiple conditions implying the same goal, the same condition implying multiple goals, and chains of conditions and goals. These rules are also connected to two kinds of consequents (facts or database), the ground unit clauses with no variable arguments and the unit clauses with a variable argument.

1.4 Programming Concepts

Logic programming, and the work behind it on constructing Prolog systems, has contributed a lot to clarification of the relationship between computing and logic. Figure 1 shows some relationships between concepts from conventional programming languages and concepts from first order logic languages. During a teaching process students may learn how to compare these concepts, in order to understand why logic programming is a new way (point of view) of looking at computation.

Chapter 2 One or Two Things About Prolog

Prolog is different from most programming languages in that it does not presuppose a von Neumann architecture and does not have assignment as the basic underlying operation (Warren 1981 c). The primitive unit is not an instruction, but a theorem of first order logic, called a clause. Variables inside clauses are universally quantified and may be substituted by well-formed formulas, but predicates cannot be quantified. Unlike conventional languages such as FORTRAN and PASCAL, Prolog is well suited to parallel processing machines because program evaluations can be run in any order without regard to sequence.

For applications requiring an easy-to-use and transparent language for symbol processing, Prolog offers significant advantages over LISP (Warren et al. 1977). LISP does not have a good syntax or variable binding mechanism, but its major barrier to readability is the size and degree of nesting of typical function definitions. Prolog allows a program to be built in small modules, each having a natural declarative reading. In addition it gives the programmer generalized record structures with an elegant mechanism for their manipulation. The pure LISP view of computation as simple function evaluation is too restrictive for typical applications. Prolog allows programs with similar behavior to be written without having implementation-oriented concepts. Experiments carried out by Warren et al. (1977) showed that a particular Prolog system, namely the DEC-10 one, was more efficient than a comparable LISP system.

For applications requiring interactive programming, a system language which is easily and quickly modified, and a simple and fast language between analyzing a problem and building a running system, such as those in CAAD, Prolog offers real advantages over FORTRAN (Swinson 1981). In this study, it was shown that Prolog takes less man-power, produces half the number of bugs found when using FORTRAN, occupies half of the FORTRAN code space and needs half of the working time required by FORTRAN.

For query processing applications, a Prolog database system called Chat-80 was compared to ‘System R’ and ‘Ingres’ regarding the general underlying strategies, answer time and query formalisms (Warren 1981 c). Chat-80 deals with interactive queries. Its query planning is not exhaustive as in System R, and the search is limited because it can make a “best guess” at each stage. Chat-80 algorithms take a time which is nearly linear in the size of the query, whereas System R is at least exponential. Experiments with running Ingres on the Chat-80 database showed that Ingres often accesses more tuples than Chat-80 does on the same query. Queries which Chat-80 answers in under a second on a DEC-10 typically take several minutes of CPU time with Ingres on a PDP-11. Query formalisms

such as Quel and SQL make life difficult for the implementor, both conceptually and in terms of implementation code. At the SYSLAB, in Sweden, Bubenko's group is currently studying the use of Prolog as an implementation language for a design tool. The idea is to have the database designer working at the interpreter top level, with a number of predefined utility procedures at his disposal. At the Technical University of Lisbon, Sernadas's group is also applying Prolog to the implementation of knowledge bases for a system specification support system (Coelho et al. 1983).

A major attraction of the language, from a user's point of view, is ease of programming. Clear, readable, concise programs can be written quickly with minimum error and reduced development costs. And, its expressive power allows experts to articulate more of the nuances and details of thought processes.

2.1 Learning Prolog Programming Style

Prolog is a step towards a more declarative style and can be viewed as a descriptive language as well as a prescriptive language. The Prolog approach is rather to describe known facts and relationships about a problem than to provide the sequence of steps taken by a computer to solve the problem. When a computer is programmed in Prolog the actual way the computer carries out the computation is specified partly by the logical declarative semantics of Prolog, partly by what new facts Prolog can "infer" from the given ones, and only partly by explicit control information supplied by the programmer.

A Prolog program is a sentence in predicate logic and consists of a sequence of Horn clauses. The clauses are implications of the form

$p_1 \text{ if } p_2 \& \dots \& p_k$

written as

$p_1 :- p_2, \dots, p_k.$

where each Prolog data object p_i , is of the form $R(T_1, \dots, T_n)$, consisting of a relation name R followed by a list of argument terms T_1, \dots, T_n . Terms provide a description of data types. A term is either a variable or a constant or a compound term (structured data object). Variables represent the universal type. Terms with free variables are the data structures of a logic program, i.e. they represent the class of objects which may be transformed through the instantiation of variables. Ground terms represent the class of only one object. A compound term comprises a symbol function with a sequence of terms. A functor may be thought of as a record type, and the arguments as the fields of a record.

The constants include integers such as:

0 1 - 999

Constants also include atoms such as:

a void ! = 'C-Prolog' []

Variables are distinguished by an initial letter in uppercase or by the initial character ‘_’, for example:

```
X      Point      _OUTPUT
```

Anonymous variables (those referred to once) are indicated by a single underline character _.

Compound terms comprise a functor and a reference of one or more terms called arguments. For example, the compound term whose functor is named ‘works_with_at’ of arity 3, with arguments X, Y and Z, is written:

```
works_with_at(X, Y, Z)
```

Lists, trees and strings are other important classes of data structures. A list is either the atom [], representing the empty list, or a compound term with functor ‘.’ and two arguments which are respectively the head and tail of the list, for example,

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

which could be written, using the standard syntax, as:

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

In general, a list is written as:

```
[Head|Tail]
```

A (binary) tree is represented as a compound term, whose functor is named ‘tree’, of arity 3,

```
tree(Subtree1, Root, Subtree2)
```

where the first and third arguments represent the subtrees and the second argument the root.

A further notational variant is allowed for lists of integers which correspond to ASCII character codes. Lists written in this notation are called strings. For example, the word ‘system’ is written as:

```
[115, 121, 115, 116, 101, 109]
```

Functors are analogous to common nouns in natural language. For example, the sentences ‘Ron gave a paper to Helder’ and ‘Ron and Helder are scientists’ can be described as the following structures:

```
sentence(np(n(ron)), vp(v(gave), np(art(a), n(paper))),  
        compls(prep(to), np(n(helder)))))  
sentence(np([ron, holder]), vp(v(are), adj(scientists)))
```

These sentences may also been seen as a part of an abstract data structure called a semantic network:

```
object(event1, paper).  
recipient(event1, holder).  
isa(holder, scientist).  
actor(event1, ron).  
action(event1, gave).  
isa(ron, scientist).
```

Sometimes it is advisable to write certain functors as operators: 2-ary functors may be declared as infix operators and 1-ary functors as prefix or postfix operators, for example:

```
: - op(600,yfx,'and').
:- op(500,xfy,'loves').
```

where ‘loves’ is an infix operator with precedence 500 (‘:-’ is scaled to 255) and right associative. These operators simplify the written form of terms with nested parentheses and emphasize the power of expression in Prolog:

```
joao loves maria.
helder loves margarida.
maria loves jose.
P and Q :- P,Q.
```

A possible query to the above program (two operator declarations, four unit clauses and one Horn clause) would be written as:

```
?- X loves Y and Y loves X.
```

meaning “who loves someone and is also loved by the same person?”.

In KBS terminology, the clauses are called rules (If-Then rules, situation-action rules or production rules), with antecedents (patterns that can be matched against entries in the database) and consequents (actions that can be performed or conclusions that can be deduced if all the antecedents match). In Computer Science terminology the antecedents constitute the so-called body of a procedure, and the consequent its head.

A Prolog clause

```
b(X) :- a1(Y), a2(Z), a3(W). (1)
```

has two interpretations:

1) it is a problem reduction method (the declarative reading)

The clause is a problem solving method that deals with the reduction of problems in form ‘b’ to a set of dependent sub-problems {a1, a2, a3}, and

2) it is a procedure (the procedural reading).

The clause is a procedure declaration whose head ‘b’ identifies the form of procedure calls to which it may answer, and whose body {a1, a2, a3} is an ordered set of procedure calls ‘ai’.

When a Prolog program is to be executed in order to attain some goal or to solve some problem, the objective of the Prolog machinery is to verify if the goal statement is true or false in all possible interpretations through a proof of its validity or inconsistency. The Prolog proof procedure is composed of an inference system and a search strategy. The inference system specifies what to do, i.e. what are the admissible derivations. The search strategy determines (how it is done) the sequence in which derivations in the search space are generated in the search for a refutation, i.e. a sequence of goal statements beginning with the initial goal and ending with the empty one. The proof can also be interpreted as a computation.

In brief, Prolog has an inference system, defined by the resolution (Robinson 1965) rule of inference, and a proof procedure built upon a selection rule (the left-most term), and a search strategy (depth-first, left-most-descendent first: ordering fixed by the order of clauses in the program). The resolution principle derives a new clause from two clauses that have complementary terms (placed on opposite sides of ‘:-’) that, with an appropriate substitution, can be made identical. The deduced clause consists of the disjunction of the original two clauses where the complementary terms (terms with the same predicate and number of arguments and placed on opposite sides of ‘:-’ in different clauses) unified in the two clauses are deleted and the unifying substitution is applied to the resultant clause.

Example 1: Execution of a Prolog Program

Given four facts (assertions) and a general rule (conditional) check whether a goal statement is true.

facts

(1)	scientist(helder).	Helder is a scientist.
(2)	scientist(ron).	Ron is a scientist.
(3)	portuguese(helder).	Helder is portuguese.
(4)	american(ron).	Ron is american.

rule (5) $\text{logician}(X) :- \text{scientist}(X).$ Every scientist is logician.

goal (6) $?- \text{logician}(X), \text{american}(X).$ Which scientist is logician and american?

The goal statement is a denial (“none is logician and american”) and contradicts what is implicit in (1) to (5), i.e. “Ron is logician and american”. Top-down inference derives new denials from this one, as is illustrated in the tree structure of Fig. 1. Therefore, (1) to (6) is an inconsistent set of clauses. Also, goal (6) may be viewed as a query (question) to the knowledge base defined by (1) to (5). The proof of inconsistency invokes automatically $X=ron$ in deducing the answer to the query.

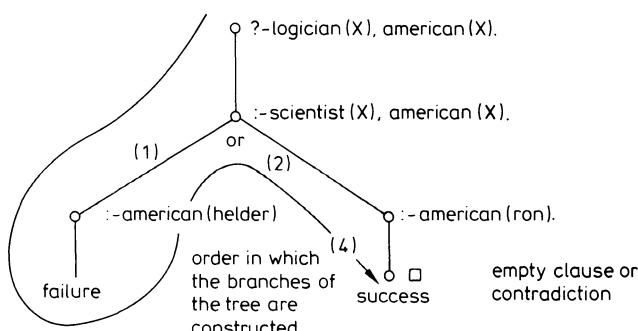


Fig. 1. The generated proof

Observe that the natural language sentence (1) could also be translated by "is(holder, scientist)", closer to the logic. Also, sentence (5) is translated into logic as " $\forall X [is(X, \text{scientist}) \rightarrow is(X, \text{logician})]$ ", and question (6) as " $\text{which}(X, is(X, \text{logician}) \& is(X, \text{american}) \& is(X, \text{scientist}))$ ".

The goal is proved false because there is some american logician called Ron. The refutation of (1)-(6) is carried out top-down and left-to-right. The first branch conducts the proof to a failure, but the program backtracks and finds an alternative branch giving success.

Note that Prolog programming style is axiomatic, and it allows all the paths (theorems) of computation to be shown, as opposed to, for example, FORTRAN where we may find only one path or solution. The computation can be viewed as controlled deduction. The underlying reasoning style is known as backward-chaining or consequent reasoning, and the control strategy is known as goal-driven.

Figure 1 shows that a Prolog program is executed (computed) according to the following algorithm:

- 1) to execute a goal, match it against the head of a clause and then execute the goals in the body of the clause;
- 2) if no match can be found, backtrack, i.e. go back to the most recently executed goal and seek an alternative matching;
- 3) execute goals in left-to-right order; and
- 4) try clauses in top-to-bottom order.

Example 2: Generation of a Proof

Finding the reverse of a list is implemented by the following recursive program:

```
(1) reverse([],[]).
(2) reverse([X|Y],Z):- reverse(Y,W), append(W,[X],Z).
(3) append([],X,X).
(4) append([X|Y],Z,[X|W]):- append(Y,Z,W).
```

There are two recursive definitions, one for the reverse relation and the other for the append relation. Both definitions behave like a relational database which can be queried in different ways to construct the list.

The two queries

```
?- reverse([a,b,c],X).
```

and

```
?- reverse(X,[a,b,c]).
```

ought to be the same, but they behave quite differently if you ask them for a second solution. This is in contrast to append which behaves much better with regard to backtracking.

Note that a programming language such as BASIC does not support recursion, but PL/1 and PASCAL do. In those languages all reentrant subroutines must explicitly save their local values on push-down stacks before transferring control.

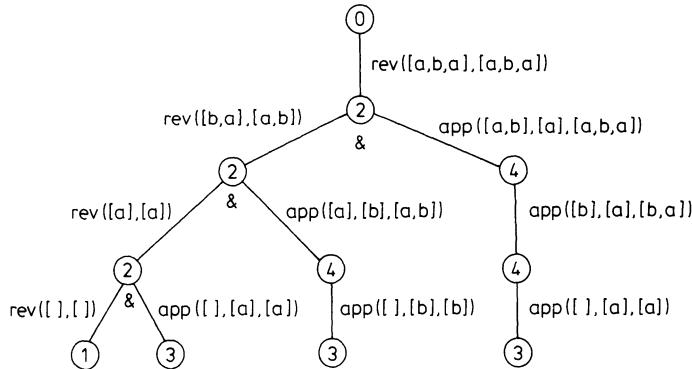


Fig. 2. The first proof found

The command,

```
?- reverse([a,b,X],[a,b,X]).
```

triggers the execution of the above program, and the first proof found is sketched in Fig. 2, a tree structure with one main branch.

Note that, in a logic program, functions that we want to compute become relations that are described by a set of assertions and simple implications. A small program, such as the “append” relation defined above through clauses (3) and (4), may be used in several tasks: to append lists, to decompose lists, to find all substrings and to split out an element, depending only on the way the calls are made.

Prolog specifications are executable because there is a parameter-passing mechanism (unification) which accesses the type descriptions of the formal and actual parameters and performs all the necessary actions.

The purpose of the Prolog unification process is threefold: 1) to decide which clause to invoke, 2) to pass the actual parameters to a clause, and 3) to deliver the results.

A collection of clauses alone does not fully specify an algorithm. The clauses determine only the logic of an algorithm. The remaining component is supplied by the proof procedure which controls the way the clauses are used to solve problems.

`algorithm = logic + control`

In a Prolog program there are as many control paths as there are clauses. The choice between the clauses can often be made explicitly dependent on parameter properties, thanks to the generality of terms as type descriptions.

Backtracking is controlled in Prolog by an extralogical primitive called `cut (!)`. The invocation of cut is a means of stopping the production of solutions when we are satisfied with the results obtained so far, and we do not want any others.

Example 3: Use of Cut

The logic for checking whether a given item is a member of a list is a recursive definition:

`member(Head, [Head|_]).` (1)

`member(Element, [_|Tail]):- member(Element, Tail).` (2)

The composition and decomposition of a list is again defined through recursion, providing a clean-cut and compact program.

The program implements a way of formulating interrelationships between variables in order to create the result, which at the end of the computation consists of all the intermediate results. If we are interested in getting only the first element of the list, clause (1) is modified to:

`member(Head, [Head|_]):- !.` (1')

The cut added to the first clause prevents the Prolog system from later backtracking and trying to use the second clause. Thus the second clause will be used only if the first clause (before '!') has failed. The cut can be used in any conjunction or disjunction of goals. Note that the two versions of the first clause are strongly dependent on what kind of questions we want to ask.

Consider a piece of a simple program in which only two clauses 'P' and 'B' are shown, where A, B, C, D, E, F and P are metavariables standing for predicate instances. The flow of control consists of a call to 'P', followed by a path that goes from 'P' to 'A', from 'A' to its execution elsewhere, from 'A' to 'B' in the first clause, then to B at the head of the second clause, etc. and eventually back to A.

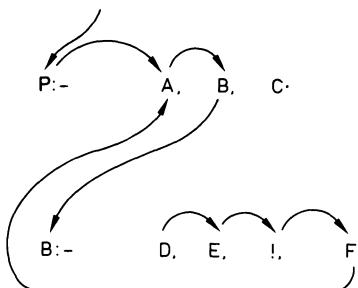


Fig. 3. The effect of a cut

In Fig. 3 we show the effect of a cut placed between 'E' and 'F' in the body of the second clause. The flow of control is now different from previously, when goal F fails, because backtracking now returns to goal 'A' immediately, i.e. to before the goal 'B' that activated the clause with the cut.

Example 4: Negation as Failure

Prolog does not include any explicit negation symbol and negation is treated as unprovability. The definition of negation as failure is written in Prolog by combining the cut symbol '!' with the built-in predicate 'fail':

```
not(X):- X, !, fail.  
not(X).
```

Variable 'X' is a procedure call, say a meta-variable. Because of the cut, the two clauses behave together as a conditional branching, with the following interpretation:

to prove `not(X)`

```
first try to prove X  
if X succeeds then not(X) fails,  
if X fails then not(X) succeeds.
```

'fail' behaves as an arbitrary atom which matches the head of no clause.

A good programming style avoids the technique of cut-fail because logic almost disappears. In the cases when this technique is necessary, the use of `not(X)` is mostly advisable.

Example 5: Implementation of if-then-else Statement

The clause:

```
a1:- b1, !, b2.  
a1:- b3.
```

is equivalent to the following:

```
to prove a1 if you can prove b1 then prove b2 else prove b3
```

2.2 Creating Programs

There are now several interpreters and compilers for the Prolog language (see Appendixes 1 and 2). In spite of an effort to establish a standard, led by the British Standards Institute and the National Physical Laboratory in the United Kingdom, the available commercial products still present some differences. Therefore, before starting to use any Prolog system it is advisable to consult its manual.

Here we adopt C-Prolog on a VAX in a VMS environment as an example, to show how users may create programs and interact with the top level. Once a beginner knows how to consult (and reconsult) files and make queries, he is ready to learn Prolog by reading and using the example programs, presented in this book from Chapter 3 onwards.

It is recommended that the text of a Prolog program be built up in a number of files by using a text editor. C-Prolog can then be instructed to read in programs from these files (consulting the file). To change parts of a program being run, it is possible to reconsult files containing the changed parts. This means that definitions for procedures in the file will replace any old definitions for these procedures.

We assume that there is a command on the selected computer, for example `prolog`, that invokes C-Prolog. When the '`prolog`' command is typed, C-Prolog will output some message, a headline and prompt the user for directives as follows:

```
C-Prolog version 1.5
```

```
yes
```

```
| ?-
```

To input a program from a file called text, it is necessary to give the directive:

```
| ?- [text].
```

or

```
| ?- [text1, text2, text3].
```

when the program is distributed through three files.

Sometimes, when a file has an extension like new in text.new, one gives the directive:

```
| ?- ['text.new'].
```

After the program interpretation is over a message is delivered to the user, stating its correctness. When the program has a syntax error, information is given to help the user to discover it. Otherwise, program execution may be started by some query. For example, suppose that the program reverse of example 2 is in the file text. So, the query:

```
| ?- reverse([a,b,c,],R).
```

will be followed by the result:

```
R=[c,b,a]
```

Clauses may also be typed in directly at the terminal. To enter clauses at the terminal, it is necessary to give the directive:

```
| ?- [user].
```

The interpreter enters a state where it expects input of clauses or directives. To get back to the top level of the interpreter, one end-of-file character (e.g. control-Z) must be typed.

Once a program has been read, the interpreter will have available all the information needed for its execution. This information is called a program state, and it may be saved in a file, with an extension sav, for future execution by performing the command:

```
| ?- save(text).
```

Save can be called at top level, from within a break level, or from anywhere within a program. Once a program has been saved in a file, e.g. text.sav, CProlog can be restored to this saved state by invoking it as follows:

```
prolog text
```

or by delivering the following command after entering the Prolog system:

```
| ?- restore(text).
```

Chapter 3 Using Prolog

In this chapter we present some simple problems regarding the representation and the manipulation of the current data structures, such as lists, sets, stacks, queues, chains, networks and trees. And, we call the attention of the reader to the power of Prolog to describe labeled combinatorial objects, such as trees. We dedicate also some problems to the manipulation of relations, such as permutation and sorting. By doing this we introduce the basic building blocks that every programmer would like to see as a built-in predicate or in some library of routines. These small programs are useful for current applications, and our purpose is to make them independent so the reader may use them. Some of the programs are too compact and therefore difficult to read. Others are also difficult to understand because they are based on recursive definitions, which in all cases cover a complex machinery. But in all cases we give straight comments to improve the readability of these programs.

Problem 1

Verbal statement:

Write a predicate to check whether an element is a member of a list or to search all members of a list. Apply your predicate to search out the members of a list that are greater than some integer. In brief, you are required to write a program to deal with the membership relation.

Logic program:

```
member(Head, [Head|_]).  
member(Element, [_|Tail]):- member(Element, Tail).
```

Comments:

The two clauses encapsulate the definition of member of a list, i.e. “H is member of a list when it is its head or when it belongs to its tail”.

As the definition is recursive, it is important to uncover the underlying machinery. The list is searched sequentially from the head to the element H, or to its end. The second clause has the means of disaggregating the list through the pattern placed in the second argument of the head.

Observe that a simple reordering of the clauses above results in an unacceptable computational behavior. The new program is not able to establish the truth of any consequent involving a call of ‘member’ for a list with variable tail.

Execution:

For the program above, if we execute the goal expressed by the question:

```
?-member(a,[a,b,c]).
```

we obtain the following solution:

```
yes
```

Other questions:

```
?-member(a,List).
List=[a|_274];
List=[_280,a|_274]
```

Notice:

Strings `_274` and `_280` are internal representations of variables. There are certain implementations that give names to variables (example: Arity Prolog); however our executions were obtained with C-Prolog which displays the internal representation of variables.

```
?-member(a,[a,b,c,a]).
```

```
yes
```

The goal concerning searching subject to some condition may be phrased:

```
?-member(X,[1,2,3,4,5]),X>2.
X=3 ;
X=4 ;
X=5 ;
no
```

Problem 2

Verbal statement:

Check whether Head is a member of a list; search for the first member of a list. Consider a list such as [a,b,c].

Logic program:

```
member(Head,[Head|_]) :- !.
member(Element,[_|Tail]) :- member(Element,Tail).
```

Execution:

When searching for the first member of the list there is only one result:

```
?-member(Element,[a,b,c]).  
Element=a
```

The other two potential solutions are discarded on account of the cut symbol ‘!’.

When insisting by typing ‘;’, and forcing backtracking, we get:

```
no
```

A similar result is obtained when trying to confirm ‘a’ as the first element of the list:

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

Problem 3 [Clocksin; Mellish 1981]

Verbal statement:

When manipulating lists there are several basic predicates that every Prolog programmer would like to have as a built-in predicate. Write the programs for “find the last element of a list”, “check consecutive elements of a list”, “erase one element from a list and generate another list”, and “erase all occurrences of an element from a list and generate a new list”.

Logic program:

```
find_last(X,[X]).  
find_last(X,[_|Y]):- find_last(X,Y).  
  
consecutive(X,Y,[X,Y|_]).  
consecutive(X,Y,[_|Z]):- consecutive(X,Y,Z).  
  
erase_one(X,[X|L],L):- !.  
erase_one(X,[Y|L],[Y|M]):- erase_one(X,L,M).  
erase_one(_,[],[]).  
  
erase_all(_,[],[]).  
erase_all(X,[X|L],M):- !, erase_all(X,L,M).  
erase_all(X,[Y|L1],[Y|L2]):- erase_all(X,L1,L2).
```

Execution:

```
?- find_last(X,[helder,loves,icha]).  
X=icha
```

Problem 4

Verbal statement:

Specify the relation append (list concatenation) between three lists which holds if the last one is the result of appending the first two. Consider the last list as [a,b].

Logic program:

```
append([], L, L).
append([H|T], L, [H|U]) :- append(T, L, U).
```

Comments:

The two clauses define the concatenation of two lists, i.e. “when one of the lists is nil (represented by ‘[]’) the result is equal to the other list; and, in general, the list whose first element is ‘H’ and tail ‘U’ is the result of concatenating ‘L’ with a list whose first element is ‘H’ and tail ‘T’ if ‘U’ is the result of concatenating ‘T’ with ‘L’”.

Again, the definition is recursive. The machinery of disaggregating the first list is realized through ‘T’ and supported by the pattern ‘[H|T]’, and aggregating is done through the last list, and supported by the pattern ‘[H|U]’. This last pattern is able to link all the copies generated through recursion.

This simple program is a good example of how to compute relations. It can be used for several tasks: to concatenate lists, to decompose lists, to find all substrings, and to split out an element.

Execution:

For the example proposed, we execute the goal expressed by the question:

```
?-append(X, Y, [a, b]).  
X=[]  
Y=[a, b];  
X=[a]  
Y=[b];  
X=[a, b]  
Y=[]
```

This is an example of decomposing lists. The following question illustrates the use of the program to append lists:

```
?-append([2], [3], Z).  
Z=[2, 3]
```

The following query illustrates the use of “append” for finding all the possible sublists:

```
?-append(Z,W,[a,b,c,d,e]), append(X,[U|Y],Z).
Z=[a]
W=[b,c,d,e]
X=[]
U=a
Y=[];

Z=[a,b]
W=[c,d,e]
X=[]
U=a
Y=[b];

Z=[a,b]
W=[c,d,e]
X=[a]
U=b
Y=[];

Y=[b,c];
Z=[a,b,c]
W=[d,e]
X=[a]
U=b
Y=[c];

Z=[a,b,c]
W=[d,e]
X=[a,b]
Z=c
Y=[]
```

And, the following query illustrates splitting out an element:

```
?-append(X,[e|Y],L).
X=[]
Y=_1
L=[e|_1];

X=[_12]
Y=[_1]
L=[_12,e|_1];

X=[_12,_19]
Y=_1
L=[_12,_19,e|_1]
```

Problem 5

Verbal statement:

Check whether T is a sublist of U; search all incompletely specified sublists (patterns).

Logic program:

```
sub_list(T,U):- append(H,T,V), append(V,W,U).
append([],L,L).
append([H|T],L,[H|U]) :- append(T,L,U).
```

Comments:

The program uses a building block defined previously, the relation ‘append’. The clause ‘sublist’ is in charge of decomposing the list ‘U’ into three lists ‘H’, ‘T’ and ‘W’. Considering that ‘H’ and ‘W’ are variables, the purpose of ‘T’ is the generation of all sublists of ‘U’.

Execution:

```
?-sub_list(T,[a,b]).  
T=[] ;  
T=[a] ;  
T=[a,b] ;  
T=[b] .
```

Problem 6

Verbal statement:

Check whether U is the intersection of lists L1 and L2; search the intersection of lists L1 and L2.

Logic program:

```
intersect([H|T],L,[H|U]) :- member(H,L), !, intersect(T,L,U).  
intersect([_|T],L,U) :- !, intersect(T,L,U).  
intersect(____,____,[]).  
member(H,[H|_]).  
member(I,[_|T]) :- member(I,T).
```

Comments:

The presence of control, through the cut symbol ‘!’, is necessary in order to eliminate a side-effect of the logic program ‘intersect’: the generation of the intersec-

tion's sublists. Again, the whole program integrates a building block, the relation 'member'.

Problem 7

Verbal statement:

Check whether List1 is the reverse of List; search for the reverse of List.

Adapt your program in order to apply it to the calculation of the number of Logical Inferences Per Second (LIPS test: 1 LIPS=400 instructions). This test is traditionally performed with what is called the “naive reverse”.

Logic programs:

```
/* program 1 */

reverse([],[]).
reverse([Head|Tail],List):- reverse(Tail,Rest),
                           append(Rest,[Head],List).

append([],List,List).
append([Head|Tail],List,[Head|U]):- append(Tail,List,U).

/* program 2 */

reverse2(List1,List):- reverse_append(List1,[],List).

reverse_append([Head|Tail],List,M):-
                           reverse_append(Tail,[Head|List],M).

reverse_append([],List,List).

/* Adaptation of program 1 for LIPS measuring */
/* The reverse predicate used is the one of program 1 */
/* 'reverse', applied to a list with N elements, */
/* does 1/2(N*N+3*N+2) LI's. */
r:- reverse([1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,
       6,7,8,9,0],_),!.
rev:- r, fail.

rev.

/*      'r' (30 element list) performs 496 LI's.      */
/*      so 'rev' does approximately 500 LI's.      */
lips:- rev, rev, rev, rev, rev, rev, rev, rev, rev, rev.
/*      'lips' does approximately 5000 LI's.      */
```

Comments:

Again, we have in program 1 a recursive definition. The ‘reverse’ term of the head of the second clause disaggregates the list until ‘nil’. The end of the recursion is controlled by the first clause. The ‘append’ relation places the elements of the list in the opposite order, one by one. The term ‘reverse’ in the body of the second clause aggregates the elements of the list, starting from its end and working to the beginning. In brief, checking whether ‘List1’ is the reverse of ‘List’ is accomplished by comparing systematically the first element of ‘List’ with the last element of ‘List1’, till the end of ‘List’ or till the finding of two different elements.

Program 2 envisages a different solution. In the first place the reverse of ‘List’ is built up through ‘reverse_append’, and only after that is a comparison done between ‘List’ and ‘List1’.

Notice:

In the LIPS measuring the existence of the predicate ‘rev’, which is used in order to save the place occupied by each execution.

Execution:

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

Problem 8

Verbal statement:

Check whether a word is a palindrome, i.e. the same if read backwards, as for example “MADAM”. Write a program able to check a palindrome for each word you give.

Logic program:

```
/* Interface with the user */
begin(X):- read(X), (X=stop ; test_palindrome(X), begin(Y)).

test_palindrome(X):- name(X,Nx), palindrome(Nx), write(X),
                   write(' is a palindrome'), nl,!.
test_palindrome(X):- write(X),
                   write(' is not a palindrome'), nl.

/* Test mechanism */
palindrome(X):- reverse2(X,X).

reverse2(L1,L):- reverse_append(L1,[],L).

reverse_append([H|T],L,M):- reverse_append(T,[H|L],M).
reverse_append([],L,L).
```

Comments:

The second logic program for ‘reverse’ is plugged into the global program, because the checking mechanism is based upon a test done on the equality of two lists. After reading the user input, a list of ASCII codes is associated to each word through the built-in predicate ‘name’. The cut symbol ‘!’ placed in the first clause of ‘test – palindrome’ is responsible for avoiding any stupid backtracking.

Predicate ‘name(A,NA)’ is a common Prolog built-in which, given a string in variable A, gives in variable NA a list with the ASCII codes of its elements and vice-versa.

Execution:

```
?-begin(X).
madam. john. astyuytsa. horse. bull. stop.

madam is a palindrome
john is not a palindrome
astyuytsa is a palindrome
horse is not a palindrome
bull is not a palindrome
```

Problem 9

Verbal statement:

Given two lists, where the second one has the first as one of its parts, find the difference of the two lists and build a palindrome with it.

Logic program:

```
/* Interface: getting the initial lists */
palindrome(L1,L2):- list(L1,Y),list(L2,Z),
concatenate(X,Y,Z),
write('Third list: '),write(X),nl,
reverse(X,W),
concatenate(X,W,K),
write('Palindrome: '),write(K).

/* Mechanisms of concatenation and reverse */
concatenate([],L,L).
concatenate([E|R],L2,[E|L]):- concatenate(R,L2,L).
reverse([],[]).
reverse([E|R],L):- reverse(R,R1),
concatenate(R1,[E],L).

/* Initial lists */
list(a,[1,3,5,7,9]).
list(b,[a,b,c,d,e,1,3,5,7,9]).
```

Execution:

The execution of:

```
?- palindrome(a,b).
```

gives us the following result:

```
Third list: [a,b,c,d,e]
```

```
Palindrome: [a,b,c,d,e,e,d,c,b,a]
```

Comments:

Notice that the interface can be easily changed in order to read the lists to be handled from the user and not to use the unit clauses ‘list(a, . . .)’ and ‘list(b, . . .)’.

Problem 10

Verbal statement:

This problem is a sophisticated application of list concatenation. It pretends to build, on the basis of a list of French words (for instance, animal names), a list of new words, each one obtained from the ancestors by juxtaposition of the words whose final characters are the first ones of another. For example the words VACHE (cow) and CHEVAL (horse) give the word VACHEVAL (mutation problem).

(This problem was suggested by H. Meloni).

Logic program:

```
/* Interface */

begin:- mutation(X), name(Nn,X), write(Nn), nl, fail.
begin:- nl, write('Done.'), nl.

/* Internal mechanisms */

mutation(X):- animal(Y), animal(Z), Y\==Z, name(Y,Ny), name(Z,Nz),
append(Y1,Y2,Ny), Y1\==[], append(Y2,Z2,Nz),
Y2\==[], append(Y1,Nz,X).

append([],X,X).
append([A|X],Y,[A|Z]):- append(X,Y,Z).

/* Database */

animal(alligator).          /* crocodile */
animal(tortue).              /* turtle */
animal(caribou).             /* caribou */
animal(ours).                 /* bear */
animal(cheval).               /* horse */
animal(vache).                /* cow */
animal(lapin).                 /* rabbit */
```

Comments:

The logic program is composed of four components: the reading clause ‘begin’, the definition of the desired object ‘mutation’, the relation ‘append’, and the database defined by the set of unit clauses ‘animal’.

The goal consists of building up a new list upon the already existing ones, in the following way: when the list ‘Y2’, defined by the last elements of list ‘Ny’, is equal to the list of the first elements of ‘Nz’, the new list ‘X’ is built through concatenation of ‘Y1 = Ny-Y2’ with ‘Nz’. Note that the final part of the first list is put aside because its absence is required in the new lists. However, another alternative would be to take out the initial part of the second list. A modification would be required in the clause ‘mutation’: the last term ‘append(Y1,Nz,X)’ would be substituted by ‘append(Ny,Yz,X)’.

See comments on problem 8 for the definition of the built-in ‘name’.

Execution:

```
?- begin.
alligator
caribours
chevalligator
chevalapin
vacheval
Done.
```

Problem 11

Verbal statement:

Write the ‘pickup the first n members of a list’ relation.

Logic program:

```
get_till_n([],[],_):- !.
get_till_n(L,_,N):-length(L,N2),N2<N,! ,write('Nonsense').
get_till_n([X|L],[X|P],N):- N>0,N1 is N-1,
                                get_till_n(L,P,N1).
get_till_n(_,[],0).
```

Execution:

```
?-get_till_n([a,b],X,3).
Nonsense
?-get_till_n([a,b,c,d],X,3).
X=[a,b,c]
```

Comments:

The term ‘length(L,N)’ is a built-in predicate that for a certain list L gives back its length N.

Problem 12

Verbal statement:

Check whether all the elements of a list satisfy some property (i.e. a unary predicate).

Logic program:

```
satisfy_property([],_).
satisfy_property([X|L],P):- R=..[P,X],R,
                           satisfy_property(L,P).

beautiful(mary).
beautiful(anne).
beautiful(louise).
```

Execution:

```
?-satisfy_property([mary,anne,louise],beautiful).
yes
```

Comments:

The term ‘X=..[N,A]’ represents a built-in operator ‘=..’, normally available in all the Prolog implementations, used for building or splitting terms.

Example:

```
a(b)=..[a,b].
```

Problem 13 [Emden, personal communication]

Write the definition of combination of a list based upon the notion of successor.
Use a successor notation to define the level of the combination.

Logic program:

```
combinations(s(N),[H|T],[H|U]):- combinations(N,T,U).
combinations(s(N),[_|T],U):- combinations(s(N),T,U).
combinations(0,_,[]).
```

Comments:

The second clause is in charge of generating all combinations.

Execution:

```
?-combinations(s(s(0)), [a,b], L).
L=[a,b] ;
no
?-combinations(s(0), [a,b], L).
L=[a] ;
L=[b] ;
no
```

Problem 14 [Emden, personal communication]

Verbal statement:

Define permutations of a list.

Logic program 1:

```
permutations1(List, [Head|Tail]) :- append(V, [Head|U], List),
                                         append(V, U, W),
                                         permutations1(W, Tail).

permutations1([], []).

append([], List, List).
append([Head|Tail], List, [Head|U]) :- append(Tail, List, U).
```

Comments:

The number of permutations of a list is the factorial of its length. This definition is written in the first clause of ‘permutations1’, where the pattern of the second argument of its head disaggregates the input list. The overall decomposition of the list is realized through the recursive definition until there are no elements in the list, i.e. the ‘nil’ is reached. This control mechanism is written in the second clause. The first ‘append’ relation works out to compose the permuted list. It is fed through its output argument (the third one, from left to right) in order to generate all the permutations in the input arguments (first and second arguments). The second ‘append’ relation concatenates the resulting list by linking the successive copies generated during the recursion.

In brief, the behavior of the execution of ‘permutations1’ is the following:

- (1) :-permutations1([1,2],L).
- (2) :-permutations1([1,2],[H|T]).
- (3) :-append(V,[H|U],[1,2]),append(V,U,W),permutations1(W,T).
- (4) :-append([], [1,2], [1,2]),append([], [2], [2]),permutations1([2],T).

```
(5) :-permutations1([2],T).
(6) :-append(V1,[H1|U1],[2]),append([],[],[2]),permutations1
     ([2],T).
(7) :-append([],[],[2]),append([],[],[]),permutations1([],T1).
(8) :-permutations1([],[]).
     :-permutations1([2],[2]).
     :-permutations1([1,2],[1,2]).
```

Some steps were discarded in order to make the behavior understandable. The permutations are built from left to right. When the first solution is rejected, other alternatives are searched through backtracking. The last elements are the first ones to be changed. When the program ends the changes with the elements on the right side of the first element of the list, it starts to generate the permutations constructed with the second element of the initial list. This fact justifies the fact that the clause ‘permutations1’ with empty lists is placed at the bottom of the program.

Logic program 2:

```
permutations2([],[]).
permutations2(L,[H|T]):-permutations2(W,T),append(V,U,W)
                           append(V,[H|U],L).
```

Comments:

In brief, the behavior of the execution of ‘permutations2’ is the following:

```
(1) :-permutations2([1,2],L).
(2) :-permutations2([1,2],[H|T]).
(3) :-permutations2(W,T),append(V,U,W),append(V,[H|U],[1,2]).
(4) :-permutations2([],[]),append(V,U,W),append(V,[H|U],[1,2]).
(5) :-append([],[],[]),append(V,[H],[1,2]).
(6) :-append([],[],[1,2]).
(7) :-append([],[],[1,2]).
(8) :-append([],[],W).
(9) :-permutations2([2],T),append([],[],[2]).
```

.

.

.

Now, the permutation is built up from the end to the beginning. So, the clause ‘permutations2’ with empty lists is placed at the top of the program. The variables ‘U’ and ‘V’ are a guarantee that all permutations are done.

Logic program 3:

```
permutations3([],[]).
permutations3([X|L1],[X|L2]):- permutations3(L1,L2).
permutations3([X|L1],[Y|L2]):- delete(Y,L1,L3),
                           permutations3([X|L3],L2).

delete(H,[H|L],L).
delete(H,[X|L],[X|NL]):- delete(H,L,NL).
```

Logic program 4:

```
permutations4([],[]).
permutations4(List,[Element|Rest]):-
    delete(Element,List,New_list),
    permutations4(New_list,Rest).
```

Comments:

Program 3 and program 4 are based upon the delete relation instead of the append relation. Program 3 builds the new list from left to right (see its second clause), and program 4 from right to left (see its second clause and the first one of ‘delete’).

Now the definition of permutations is the following: “to permute a list get a list starting with ‘Element’, delete the ‘Element’ from the initial List, permute the result and jam the ‘Element’ back on at the start of that”.

Execution:

```
?-permutations1([a,b,c],L).
L=[a,b,c] ;
L=[a,c,b] ;
L=[b,a,c] ;
L=[b,c,a] ;
L=[c,a,b] ;
L=[c,b,a] ;
no

?-permutations2([a,b,c],L).
L=[a,b,c] ;
L=[b,a,c] ;
L=[c,a,b] ;
L=[a,c,b] ;
L=[b,c,a] ;
L=[c,b,a] ;
no

?-permutations3([a,b,c],L).
L=[a,b,c] ;
L=[a,c,b] ;
L=[b,a,c] ;
L=[b,c,a] ;
L=[c,a,b] ;
L=[c,b,a] ;
no

?-permutations4([a,b,c],L).
L=[a,b,c] ;
L=[a,c,b] ;
L=[b,a,c] ;
L=[b,c,a] ;
L=[c,a,b] ;
L=[c,b,a] ;
no
```

Problem 15

Verbal statement:

Write the set equality relation. Consider a set represented by a list.

Logic program:

```
set_equal(X,X):- !.
set_equal(X,Y):- equal_lists(X,Y).

equal_lists([],[]).
equal_lists([X|L1],[Y|L2]):- delete(X,L2,L3),
                           equal_lists(L1,L3).

delete(X,[X|Y],Y).
delete(X,[Y|L1],[Y|L2]):- delete(X,L1,L2).
```

Comments:

The choice of a list structure for the representation of a set poses some problems, because in a set the elements are unordered, but the notion of order is associated with a list. So, every element of a list may be compared with all the elements of the other list.

The program ‘equal_lists’ works as a generator of the permutation. Therefore any program for the ‘permutation relation’ could be used.

Execution:

```
?-equal_lists([],L).
L=*[; *]
no

?-set_equal([a,b,c],[b,c,a]).
yes

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

Problem 16

Verbal statement:

Write a program for building the subtraction of two lists.

Logic program:

```
subtract(L,[],L):- !.
subtract([H|T],L,U):- member(H,L),!,subtract(T,L,U).
subtract([H|T],L,[H|U]):- !,subtract(T,L,U).
subtract( _,_,[]).

member(H,[H|_]). 
member(I,[_|T]):- member(I,T).
```

Problem 17

Verbal statement:

Write a program where the main operations on sets are defined, in particular subset, disjoint, intersection, union, and complement.

Use the predicate “member” defined in PROBLEM 1.

Logic program:

```
/* subset */

subset([A|X],Y):- member(A,Y),subset(X,Y).
subset([],Y).

/* disjoint */

disjoint(X,Y):- member(Z,X),not(member(Z,Y)).

/* intersection */

intersection([],X,[ ]).
intersection([X|R],Y,[X|Z]):- member(X,Y),!,
                           intersection(R,Y,Z).
intersection([X|R],Y,Z):- intersection(R,Y,Z).

/* union */

union([],X,X).
union([X|R],Y,Z):- member(X,Y),!,union(R,Y,Z).
union([X|R],Y,[X|Z]):- union(R,Y,Z).

/* complement */

subtract([],L,L).
subtract([H|T],L,U):- member(H,L),delete(H,L,L1),
                     subtract(T,L1,U).

delete(X,[X|Y],Y).
delete(X,[Y|Z],[Y|U]):- delete(X,Z,U).
```

Comments:

The cut symbol, used for the definition of the operations ‘union’ and ‘intersection’, guarantees the uniqueness and the correction of the solution.

Execution:

```
?- member(a,[a,b,c]).  
yes  
?- subset([a,b],[c,a,b,d]).  
yes  
?- subset([b,a],[c,a,b,d]).  
yes  
?- intersection([a],[a,b],[a]).  
yes  
?- union([], [a], [a]).  
yes  
?- subtract([], [a], L).  
L=[a];  
no  
?- intersection([a],[a,b],L).  
L=[a];  
no
```

Problem 18 [Emden, personal communication]

Verbal statement:

Define the bubble sort operation.

Logic program:

```
bubble_sort(L, S) :- append(U, [A, B|V], L),  
    B < A,  
    append(U, [B, A|V], M),  
    bubble_sort(M, S).  
  
bubble_sort(L, L).  
  
append([], L, L).  
append([H|T], L, [H|V]) :- append(T, L, U).
```

Comments:

The second clause of the ‘bubble sort’ relation guarantees the maintenance of the list ordering. The cut symbol avoids the use of the second clause by backtracking and the generation of an incorrect solution. Such use would be possible when two consecutive elements were out of the order.

Execution:

```
?- bubble_sort([3,6,2,4,7,1],L).
L=[1,2,3,4,6,7] ;
no

?- bubble_sort([3,7,8,5,1,4],L).
L=[1,3,4,5,7,8] ;
no
```

Problem 19 [Emden, personal communication]

Verbal statement:

Split a list with head H1 and tail T1 into two lists U1 and U2, where all the elements of U1 are = < H, all the elements of U2 are > H, and the original order is preserved.

Logic program 1:

```
split1(H,[H1|T1],[H1|U1],U2):- H1=<H,split(H,T1,U1,U2).
split1(H,[H1|T1],U1,[H1|U2]):- H1>H,split(H,T1,U1,U2).
split1(_,[],[],[]).
```

Logic program 2:

```
split2(H,[H1|T],[H1|U1],U2):- H1=<H,! ,split2(H,T,U1,U2).
split2(H,[H1|T],U1,[H1|U2]):- split2(H,T,U1,U2).
split2(_,[],[],[]).
```

Logic program 3:

```
split3(H,[H1|T],U1,[H1|U2]):- H1>H3!,split3(H,T,U1,U2).
split3(H,[H1|T],[H1|U1],U2):- split3(H,T,U1,U2).
split3(_,[],[],[]).
```

Comments:

The method of splitting adopted here is the following: the list is partitioned in such a way that all the elements of one list are less than all the elements in the other. A different recursive definition could be obtained by dividing the list into two nearly equal halves.

The first two programs are almost identical. The cut symbol placed in the first clause of the second program has the same function as the comparison ‘H1>H’, placed in the second clause of the first program.

Execution:

```
?- split1(2,[1,2,3],L1,L2).
L1=[1,2]
L2=[3];
no

?- split2(3,[2,4,1,5],L1,L2).
L1=[1,2]
L2=[4,5];
no

?- split3(6,[1,6,3],L1,L2).
L1=[1,6,3]
L2=[];
no
```

Problem 20 [Emden, personal communication]

Verbal statement:

Define a quicksort operation using append.

Logic program:

```
q_sort1([H|T],S):- split(H,T,U1,U2),
q_sort1(U1,V1),
q_sort1(U2,V2),
append(V1,[H|V2],S).

q_sort1([],[]).

split(H,[H1|T],[H1|U1],U2):- H1<H, split(H,T,U1,U2).
split(H,[H1|T],U1,[H1|U2]):- H1>H, split(H,T,U1,U2).
split(_,[],[],[]).

append([],L,L).
append([H|T],L,[H|U]):- append(T,L,U).
```

Comments:

Different strategies for divide and conquer lead to different sort programs. The merge part is accomplished by appending lists.

The program for ‘split’ was changed: the element ‘H’ was excluded because at the end of ‘split’ its position in the ordered list is fixed, and it is advisable to hold it.

Execution:

```
?- q_sort1([c,q,w,e,r,t,y,u,i,o,p,a,s,d,f,g,h,j,k,l,z,x,c,
           v,b,n,m],L).
L=[a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z];
no
?- q_sort1([8,7,6,5,4,3],2).
L=[3,4,5,6,7,8];
no
```

Problem 21 [Warren, personal communication]

Verbal statement:

Define the quicksort operation without append.

Logic program 1:

```
q_sort2(L,S):- sort2(L,S,[]).
sort2([H|T],S,X):- split(H,T,U1,U2),
                  sort2(U1,S,[H|Y]),
                  sort2(U2,Y,X).
sort2([],X,X).
split(H,[H1|T1],[H1|U1],U2):- H1<H,split(H,T1,U1,U2).
split(H,[H1|T1],U1,[H1|U2]):- H1>H,split(H,T1,U1,U2).
split(_,[],[],[]).
```

Logic program 2:

```
sort2([H|T],S-X):- split(H,T,U1,U2),
                  sort2(U1,S-[H|Y]),
                  sort2(U2,Y-X).
sort2([],X-X).
```

Comments:

Program 1 embeds the current meaning of the ‘concatenation’ (append) relation, the implicit subtraction of lists. Program 2 makes explicit this meaning. The term ‘S-X’ has a binary operator, representing the difference of two lists. So, the empty list is represented by ‘X-X’. This device allows an easy concatenation of the two lists. The expansion mechanism works on because there is a variable instead of the empty list ‘[]’, and we may unify the end of the list after ‘-’ with another list. The list ‘L-Z’ can be expressed as the concatenation of ‘L-Y’ with ‘Y-Z’. ‘Y’ has the list ‘Z’ as its end, and ‘Z’ is also the end of ‘L’. The second clause of program 2 is the implicit concatenation of ‘S-Y’ with ‘Y-X’: the result is ‘S-X’.

Execution:

```
?- q_sort2([4,7,6,3,1],L).
L=[1,3,4,6,7]
```

Problem 22 [Emden, personal communication]

Verbal statement:

Define the insert sort operation.

Logic program:

```
i_sort([H|T],S):- i_sort(T,L),insert(H,L,S).
i_sort([],[]).

insert(X,[H|T],[H|L]):- H<X,! ,insert(X,T,L).
insert(X,L,[X|L]).
```

Comments:

The list is constructed by inserting immediately each element in the right spot. The first clause of the program asserts that the new list starts with the last element of the initial list. Afterwards, the element next to last in the initial list is inserted, and so on.

The cut symbol is placed in the ‘insert’ definition as a guarantee that each element is inserted in the right spot, and that no undesirable solutions will be generated through backtracking.

Execution:

```
?- i_sort([4,5,8,5,3,6,1],L).
L=[1,3,4,5,6,7,8] ;
no
```

Problem 23

Verbal statement:

Write a program for sorting a list as an ordered permutation of that list.

Logic program:

```
p_sort(L,S):- permutation(L,S),ordered(S).

permutation([],[]).
permutation(L,[H|S]) :- delete(H,L,N2),permutation(NL,S).
```

```

delete(H,[H|L],L).
delete(X,[H|L],[H|NL]) :- delete(X,L,NL).

ordered([]).
ordered([X]).
ordered([X,Y|Z]) :- X =< Y, ordered([Y|Z]).
```

Comments:

Any program for the ‘permutation’ relation, introduced previously, can be chosen as a building block. Observe that behind this program is the “generate and test” paradigm. The problem can be broken up into a generator of permutations of a list and a predicate which decides if a list is ordered. This program is too slow, which is directly inherited from the paradigm used.

Execution:

```
?- p_sort([3,5,2,7,4],L).
L=[2,3,4,5,7];
no
```

Problem 24**Verbal statement:**

Write a program for implementing the relation ‘admissible’ between two lists a and b,

$$b_i = 2a_i \quad \text{and} \quad a_{(i+1)} = 3b_i \quad \text{for} \quad i < n$$

where a_i and b_i are the elements number i of lists a and b respectively.
(This problem was suggested by Robert Kowalski).

Logic program 1:

```

admissible1(X,Y) :- double(X,Y), triple(X,Y).

double([],[]).
double([X|Y],[U|V]) :- U is 2*X, double(Y,V).

triple([],[]).
triple([X],[U]).
triple([X,Y|Z],[U|V]) :- Y is 3*X, triple([Y|Z],V).
```

Logic program 2:

```

admissible2([],[]).
admissible2([X,Y|Z],[U|V]) :- U is 2*X, Y is 3*X,
admissible2([Y|Z],V).

admissible2([X],[U]) :- U is 2*X.
```

Comments:

Program 1 needs to use the clause ‘triple([X],[U])’ for definition of triple because two single lists are admissible if and only if the element of the second list is double the element of the first list.

Execution:**The execution of**

```
?-admissible2([1|U],V),write([1|U]),write(','),write(V),fail.
```

gives us the following pairs of lists:

```
[1],[2]
[1,6],[2,12]
[1,6,36],[2,12,72]
.
.
.
?- admissible1([1,6],[2,2]).
no
?- admissible1([1,6],[2,12]).
yes
?- admissible2([1,6],[2,12]).
yes
```

Problem 25**Verbal statement:**

Write a program to simplify (“flatten”) a list whose members are lists, and to transform it into a list of single members, containing no lists as members.

Logic program 1:

```
simplify2(L,NL):- compact(L,L1), simplify1(L1,NL).
compact([L,[]],X):- !, compact(L,X).
compact([[]|L],L0):- !, compact(L,L0).
compact([L],L0):- !, compact(L,L0).
compact([L1|L2],L):- !, compact(L1,X1), !, compact(L2,X2),
append(X1,X2,L).
compact(L,[L]). 
simplify1([X|L],NL):- member(X,L), !, simplify1(L,NL).
simplify1([X|L],[X|NL]):- simplify1(L,NL).
simplify1([],[]).
append([],L,L).
append([H|T],L,[H|U]):- append(T,L,U).
member(A,[A|_]). 
member(A,[_|L]):- member(A,L).
```

Execution:

```
?- simplify2([[0]],L).
L=[0]

?- simplify2([[1,2],[2,1,2],[1]],L).
L=[2,1]
```

Logic program 2:

```
simplify3(L,NL):- flatten(L,NL).

flatten([],[]).
flatten([[]|T],L):- !,flatten(T,L).
flatten([[X|Y]|T],L):- !,flatten([X|Y],L1),
                     flatten(T,L2),
                     append(L1,L2,L).
flatten([H|T],[H|L]) :- flatten(T,L).
```

Execution:

```
| ?- simplify2([[1,2,3,[1,2,3]],[[]],L).
L=[1,2,3]

| ?- simplify3([[1,2,3,[1,2,3]],[[]],L).
L=[1,2,3,1,2,3]

| ?- simplify2([1,[],3,[1,2,3],[],L].
L=[1,2,3]

| ?- simplify3([1,[],3,[1,2,3],[],L).
L=[1,3,1,2,3]

| ?- simplify2([[],[[1]],[1,[2,[3],4],5]],L).
L=[1,2,3,4,5]

| ?- simplify3([[],[[1]],[1,[2,[3],4],5]],L).
L=[1,1,2,3,4,5]

| ?- simplify2([1,3,[1,2],[[3]]],L).
L=[1,2,3]

| ?- simplify3([1,3,[1,2],[[3]]],L).
L=[1,3,1,2,3]

| ?- simplify2([[1,2],[2],[[1,2]]],L).
L=[1,2]
```

Comments:

Programs ‘simplify2’ and ‘simplify3’ are able to flatten input lists but only ‘simplify2’ extracts all the redundancies.

Problem 26

Verbal statement:

Extend the previous program to simplify lists when their members are nodes of a tree structure. Now the simplification means not only the extraction of all redundancies, but the elimination of dominant nodes.

Suggestion: the tree is used as a classification scheme of computing reviews; each category may be represented as follows:

```
category('computer sciences', 0, 'computer sciences
applications',1).
category('computer sciences applications',1,'artificial
intelligence',12).
```

Logic program:

```
simplify(L,NL):- compact(L,L1),simplify1(L1,L2),
simplify2(L2,NL).
simplify2(L,NL):- simplify3(L,L1),subtract(L,L1,NL).

simplify3([X|L],NL):- compare(L,X,[]),simplify3(L,NL).
simplify3([X|L],[Y|NL]):- compare(L,X,Y),simplify3(L,NL).
simplify3([],[]).

compare(_,[],[]):- !.
compare([Y|L],X,[Z|NL]):- (linked(X,Y),Z=X ; linked(Y,X),Z=Y),
compare(L,X,NL).
compare([Y|L],X,NL):- compare(L,X,NL).
compare([],_,[]).

subtract(L,[],L):- !.
subtract([H|T],L,U):- member(H,L),!,subtract(T,L,U).
subtract([H|T],L,[H|U]):- !,subtract(T,L,U).
subtract(_,_,[]).

linked(X,Y):-category(X,_,Y,_).
linked(X,Z):-category(X,_,Y,_),linked(Y,Z).
```

Problem 27 [Tarnlund 1976b]

Verbal statement:

Define a representation for stacks. Consider a stack, such as

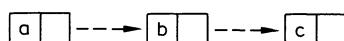


Fig. 1

Write the goal statements for deleting an element from the stack, and for inserting a new element e.

Logic program:

```
stack(s(X,Y),X,Y).
```

Comments:

This program pops-up a stack. The goal statement for deleting is:

```
?- stack(s(a,s(b,s(c,0))),X,Y).
```

X=a and Y=s(b,s(c,0))

The goal statement for inserting is:

```
?- stack(Z,e,s(a,s(b,s(c,0)))).
```

Z=s(e,s(a,s(b,s(c,0))))

The goal statement for inserting is obtained by dual programming, and it is achieved by shifting the roles between input and output variables in a goal statement.

Execution:

```
?- stack(S,a,b).
```

S=s(a,b)

```
?- stack(SA,a,s(b,s(c,0))).
```

SA=s(a,s(b,s(c,0)))

```
?- stack(s(a,s(b,s(c,0))),X,Y).
```

X=a

Y=s(b,s(c,0))

Problem 28 [Tarnlund 1976b]

Verbal statement:

Define a set of procedures for manipulating queues.

Consider the following queue:

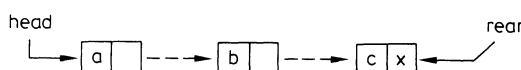


Fig. 2

Write the goal statements for deleting the head element of a queue, for inserting an element at the rear of a queue, and for inserting a new element into the queue.

Logic program:

```
delete(q(X,Y),X,Y).
insert(Y,q(W,X1),W,Y).
insert(Y,q(W,X1),W,X1,Y).
```

Comments:

The goal statement for deleting the first element of the queue ‘q(a,q(b,q(c,X)))’ is:

```
?- delete(q(a,q(b,q(c,Z))),X,Y).
X=a
Y=q(b,q(c,Z))
```

The goal statement for inserting an element ‘d’ at the rear of the queue is:

```
?- insert(q(a,q(b,q(c,X))),X,d,Z).
X=q(d,X1)
Z=q(a,q(b,q(c,q(d,X1))))
```

Observe that by applying the second assertion of ‘insert’, ‘Y’ gets instantiated with ‘q(a,q(b,q(c,X)))’ and ‘X’ with ‘q(d,X1)’. Afterwards, ‘Y’ is instantiated with ‘q(a,q(b,q(c,q(d,X1))))’, and finally the program delivers the results for ‘X’ and ‘Z’.

In this example, logic variables ‘X’ and ‘X1’ play a very important role by indicating the rear of the queues. This kind of pointer belongs to the data structure, and so it is suitable to include it in the definition of ‘insert’. The second clause of ‘insert’, the one with five arguments, takes care of this feature.

The pointer gives the spot where a new element is inserted. For example, to insert ‘d’ at the rear of some queue:

```
?- insert(q(a,q(b,q(c,X))),X,d,X1,Z).
Z=q(a,q(b,q(c,q(d,X1))))
```

The query can be re-asserted to accomplish a different goal, the insertion of ‘d’ at another spot in the queue:

```
?- insert(q(a,q(b,X)),X,d,q(c,Y1),Z).
Z=q(a,q(b,q(d,q(c,X1))))
```

Execution:

```
?- delete(q(a,q(b,X)),a,Q).
Q=q(b,X)
?- insert(q(a,q(b,X)),X,c,Q).
X=q(c,_).
Q=q(a,q(b,q(c,_)))
?- insert(q(a,q(b,q(c,X))),X,d,X1,Q).
X=q(d,X1)
Q=q(a,q(b,q(c,q(d,X1))))
?- insert(q(a,X),X,c,q(d,q(e,X1)),Q).
X=q(c,q(d,q(e,X1)))
Q=q(a,q(c,q(d,q(e,X1))))
```

The goal statement for inserting a new element ‘d’ into the queue is:

```
?- insert(q(a,q(b,q(c,X))),X,d,X1,Z).
```

Problem 29

Verbal statement:

Define a set of procedures for manipulating chains.

Logical program:

```
delete(A,q(A,X),X):- !.
delete(A,q(X,Y1),q(X,Y2)):- delete(A,Y1,Y2).

insert(A,B,q(B,X1),q(A,q(B,X1))).
insert(A,B,q(C,X),q(C,X1)):- insert(A,B,X,X1).

reverse(X,X,Z,Z).
reverse(c(U,V),X,Y,Z):- reverse(V,X,c(U,Y),Z).
```

Comments:

In order to delete element ‘b’ from the chain

Fig.3

do the following:

```
?- delete(b,q(a,q(b,q(c,nil))),Q).
Q=q(a,q(c,nil))
```

Execution:

```
?- delete(A,q(a,q(b,q(c,nil))),Q).
A=a
Q=q(b,q(c,nil))

?- insert(a,b,q(b,nil),Q).
Q=q(a,q(b,nil))

?- insert(a,nil,q(b,nil),Q).
Q=q(b,q(a,nil))

?- insert(b,c,q(a,q(c,nil)),Q).
Q=q(a,q(b,q(c,nil)))
```

The recursive definition of reverse is controlled by the first clause. Let us see the behavior of the program through an example:

```
?- reverse(c(a,c(b,x)),x,x',Z).
```

where U=a, V=x, X=x and Y=c(a,x').

The second call of 'reverse' is:

```
?- reverse(c(b,x),x,c(a,x'),Z).
```

where U=a, V=x, X=x and Y=c(a,x').

The third call of 'reverse' is:

```
?- reverse(x,xc(b,c(a,x')),Z).
```

where X=x and Z=c(b,c(a,x')).

and, finally:

```
?- reverse(x,x,c(b,c(a,x')),c(b,c(a,x'))).
Z=c(b,c(a,x'))
```

where the second argument, the rear of the initial list, controls the halt condition. The third argument of 'reverse' contains the variable with the halt of the reversed chain, and the fourth argument delivers the reversed chain.

Execution:

```
?- reverse(c(a,c(b,c(c,x))),x,x,Z).
Z=c(c,c(b,c(a,x)))
?- reverse(c(1,c(7,c(3,0))),0,u,Z0).
Z=c(3,c(7,c(1,u)))
```

Problem 30 [Bratko 1981, personal communication]

Verbal statement:

Graphs can be undirected. Choose an appropriate representation for a graph and write a program that establishes that P is a non-cyclic path between vertices U and V in a graph G.

Logic program:

```
path(U,V,G,P):- path1(U,[V],G,P).
path1(U,[U|P],G,[U|P]).
path1(U,[W|P2],G,P):- edge(V-W,G), not(member(V,P2)),
path1(U,[V,W|P2],G,P).

edge(V-W,[V-W|G]). 
edge(W-V,[V-W|G]). 
edge(V-W,[E|G]):- edge(V-W,G).
```

```

node(V,G):- edge(V-W,G).
conc([],L,L).
conc([X|L1],L2,[X|L3]):- conc(L1,L2,L3).
member(X,L):- conc(L1,[X|L2],L).

```

Comment:

A graph is represented by a list of edges:

```
G=[a-b, b-c, b-d]
```

and a path is represented by:

```
path(a,d,G,[a,b,d])
```

Problem 31 [Bratko 1981, personal communication]

Verbal statement:

Consider the previous problem and extend the logic program to handle a few more operations on undirected graphs, such as the Hamiltonian cycle and the spanning-tree of a graph.

Logic program:

```

/* Hamiltonian cycle in a graph */
hamilton(G,C):- edge(U-V,G), path(U,V,G,C), not(uncovered(C,G)).
uncovered(C,G):- node(V,G), not(member(V,C)).
/* Spanning-tree: version 1 (declarative) */
stree1(G,T):- sublist(G,T), tree(T), not(graphuncovered(T,G)).
sublist(G,[]).
sublist([E|G],[E|T]) :- sublist(G,T).
sublist([E|G],T) :- sublist(G,T).
tree(T):- not(unconnected(T)), not(hascycle(T)).
unconnected(T):- node(U,T), node(V,T), not(path(U,V,T,P)).
hascycle(T):- edge(U-V,T), path(U,V,T,[U,X,Y|P]).
graphuncovered(T,G):- node(V,G), not(node(V,T)).
/* Spanning-tree: version 2 (procedural) */
stree2(G,T):- member(E,G), spread([E],T,G), not(spread(T,T1,G)).
spread(T,T1,G):- addedge(T,T1,G).
spread(T,T2,G):- addedge(T,T1,G), spread(T1,T2,G).

```

```

addedge(T,[V-W|T],G):- member(V-W,G), (node(V,T),not(node(W,T));
node(W,T),not(node(V,T))).

/* Some graphs */
g1([a-b,b-c,b-d]). 
g2([a-b,b-d,b-c,d-e,c-e,a-c]).
```

Comments:

Let G be a graph and T a spanning-tree of G. We define two logic programs to compute spanning-trees:

(1) stree1(G,T) and (2) stree2(G,T).

The first is defined in declarative fashion, while the second is more procedurally oriented.

The declarative version is based on the following definition of spanning-tree:

- (1) T is a spanning tree of G iff T is a tree and T covers all nodes.
- (2) T is a tree iff T is connected and T has no cycle.

The procedural version of a spanning-tree of a graph is constructed by adding edges from the graph starting with the null set. To make sure that the structure thus constructed is a tree at any time, a new edge can be added only if it is connected to the current edge-set by one node exactly. At the moment that no more edge can be added we have a spanning-tree.

The above procedure assumes that a spanning-tree exists.

Problem 32

Verbal statement:

Write the description of a line connecting two points in a tridimensional space.
(Suggestion: adopt a tree for data structure).

Logical program:

```
line(point(X1,Y1,Z1),point(X2,Y2,Z2))
```

Comments:

The tree pictured below has two objects ‘point’:

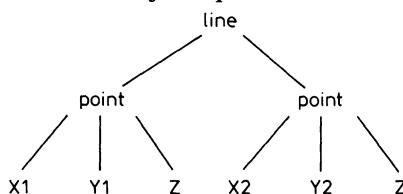


Fig. 4

Problem 33

Verbal statement:

Verify if two nodes of a tree are connected.

Logic program:

```
get_branches(X,Y) :- connected(X,Y); connected(Y,X).
connected(X,Y) :- node(X,Y).
connected(X,Z) :- node(X,Y), connected(Y,Z).
```

Comments:

Suppose we have the following tree:

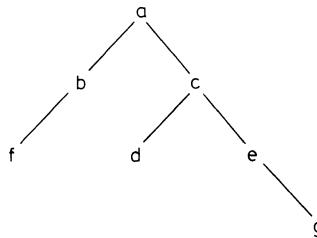


Fig.5

represented by

```
node(a,b).
node(b,f).
node(a,c).
node(c,d).
node(c,e).
node(e,g).
```

Execution:

```
?- get_branches(a,f).
yes
?- get_branches(e,f).
no
?- get_branches(e,X).
X=g ;
X=c ;
X=a ;
no
```

Problem 34

Verbal statement:

Get all nodes of a tree above and below a specified node.

Logic program:

```
get_node(X, [M|R], above) :- node(M, X), get_node(M, R, _).
get_node(_, [], above).

get_node(X, L, below) :- (down_nodes(X);
                           retract(c(L))).

down_nodes(X) :- assert(c([])), node(X, M),
                replace(c(L), c([M|L])), fail.

replace(X, Y) :- retract(X), !, assert(Y).
replace(_, Y) :- assert(Y).
```

Problem 35 [L. Pereira 1976, personal communications]

Verbal statement:

Find a subtree ST in a tree T and change it by a different subtree CH. Call R the final tree. Apply your program to the tree:

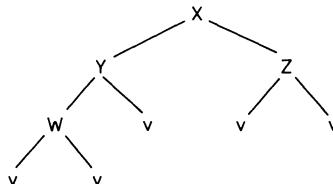


Fig. 6

Suggestion:

Use the following representation for the tree in Figure 6.

```
T=t(T1,x,T2)
  =t(t(T11,Y,T12),x,t(v,Z,v))
  =t(t(t(v,w,v),y,v),x,t(v,z,v))
```

Logic program:

```
change(CH, ST, ST, CH).
change(CH, ST, t(T1, X, R2), t(T1, X, R1)) :- change(CH, ST, T1, T2).
change(CH, ST, t(T1, X, R1), t(T1, X, R2)) :- change(CH, ST, R1, R2).
```

Execution:

```
?- change(t(v,a,v),t(v,w,v),t(t(t(v,w,v),y,v),x,t(v,z,v)),T).
   T=t(t(t(v,a,v),y,v),x,t(v,z,v))

?- change(t(v,a,v),t(v,z,v),t(t(t(v,w,v),y,v),x,t(v,z,v)),T).
   T=t(t(t(v,w,v),y,v),x,t(v,a,v))

?- change(t(v,a,v),t(v,w,v),t(v,w,v),T).
   T=t(v,a,v)
```

Problem 36 [L. Pereira; Monteiro 1978]

Verbal statement:

Construct a program for relating a binary tree with the list of its leaves (terminal nodes), and another one for testing whether two binary trees have the same leaves.

Logic program 1:

```
leaves1(void,X-X).
leaves1(t(void,N,void),[N|Z]-Z).
leaves1(t(ST1,N,STr),L-Z):- leaves1(ST1,L-X),
                           leaves1(STr,X-Z).

same_leaves(T1,T2):- leaves1(T1,L-[]),
                   leaves1(T2,L-[]).
```

Logic program 2:

```
leaves2(void,[]).
leaves2(t(void,N,void),[N]).
leaves2(t(ST1,N,ST2),L):- leaves2(ST1,L1),leaves2(ST2,L2),
                           append(L1,L2,L).

same_leaves2(T1,T2):- leaves2(T1,L),leaves2(T2,L).
```

Comments:

```
void='the empty tree'
t(void,A,void)=terminal node 'A'
t(ST1,N,STr)='the binary tree with root N,
               left subtree ST1, and right subtree STr'
```

The term ' $[N \setminus Z] - Z$ ', where ' \setminus ' is a binary functor, stands for a 'difference list'. It denotes the list whose sole member is 'N', obtained from the list ' $[N \setminus Z]$ ' 'minus' 'Z'.

The empty difference list is 'X-X'. Difference lists allow for easy concatenation of lists. Because a variable stands in place of the usual 'nil', the list may expand by

unifying the variable at the end, which is individuated after the ‘minus’ sign, with another list. Thus, in the second clause, the (difference) list ‘L-Z’ is expressed as a concatenation of ‘L-X’ and ‘X-Z’. Since ‘X’ terminates with ‘Z’, ‘L’ will also terminate with ‘Z’. A call to this procedure is

```
leaves(t(t(void,a,void),b,STr),[A,c]-[])
```

Notice that the ‘[]’ in any call has the effect of ‘closing’ the final list obtained.

Let us now briefly look at how this goal is actually executed. The goal matches only the second clause for ‘leaves’. The body of the matching clause instance is `leaves(t(void,a,void),[A,c]-X), leaves(STr,X-[])`.

The result of executing the first of these two goals against the only clause which solves it is to instantiate ‘A’ to ‘a’ and ‘X’ to ‘c’. The second goal matches the first clause, thereby instantiating ‘STr’ to ‘t(void,c,void)’, because ‘X’ is already instantiated to ‘c’.

Execution:

```
?- leaves1(t(t(void,a,void),b,STr),[A,c]-[]).
   A=a
   Str=t(void,c,void)

?- leaves1(t(void,a,void),[L]-[]).
   L=a

?- leaves1(t(t(void,a,void),b,t(void,c,void)),L-[]).
   L=[a,c]

?- same_leaves1(t(void,a,void),t(void,b,t(void,a,void))).
   Yes

?- same_leaves1(t(void,a,void),T).
   T=t(void,a,void);
   T=t(void,_,t(void,a,void)).

?- leaves2(t(void,a,void),L).
   L=[a]

?- leaves2(t(t(void,a,void),b,t(void,c,void)),L).
   L=[a,c]

?- same_leaves2(t(void,a,void),t(void,b,t(void,a,void))).
   Yes

?- same_leaves2(t(void,a,void),T).
   T=t(void,a,void);
   T=t(void,_,t(void,a,void))
```

Problem 37 [Tarnlund 1976b]

Verbal statement:

A fundamental database operation is to search for a record R_i (node) in a tree given a key K_i , and insert it as a new record if the record is not in the tree. Define an algorithm for binary tree search.

Logic program:

```
binary(t(X,K,Z),K,t(X,K,Z)).  
binary(t(X,Y,Z),K,t(X1,Y,Z)):- name(K,NK), name(Y,NY), NK<NY,  
                                binary(X,K,X1).  
binary(t(X,Y,Z),K,t(X,Y,Z1)):- name(K,NK), name(Y,NY), NK>NY,  
                                binary(Z,K,Z1).  
binary(void,K,t(void,K,void)).
```

Comments:

Suppose we have the following binary tree:

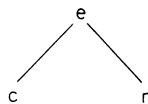


Fig. 7

and we would like to search for a key m , and insert it if it is not in the tree. Therefore, we would write the goal statement

```
?- binary(t(t(void,c,void),e,t(void,n,void)),Z).  
Z=t(t(void,c,void),e,t(t(void,void),n,void))
```

Now we have the tree 'Z' and we would like to delete node 'm'. In Prolog, we can change the roles of the logical variables, and we would write:

```
?- binary(T,m,t(t(void,c,void),e,t(t(void,m,void),n,void))).  
T=t(t(void,c,void),e,t(t(void,m,void),n,void));  
T=t(t(void,c,void),e,t(void,n,void))
```

The second alternative, generated by backtracking, is the desired solution.

Execution:

```
?- binary(t(t(void,c,void),e,t(void,n,void)),m,Z).  
Z=t(t(void,c,void),e,t(t(void,m,void),n,void)).  
?- binary(t(t(void,c,void),e,t(t(void,m,void),n,void)),m,Z).  
Z=t(t(void,c,void),e,t(t(void,m,void),n,void));  
Z=t(t(void,c,void),e,t(void,n,void))
```

Problem 38 [Tarnlund 1976b]

Verbal statement:

A frequent database operation is to delete a node in a tree. Define an algorithm for binary tree deletion.

Logic program:

```

delete(t(X,K,void),K,X).
delete(t(X,K,t(void,Y1,Z1)),K,t(X,Y1,Z1)).
delete(t(X,K,t(X1,Y1,Z1)),K,t(X,Y,t(X2,Y1,Z1))):- 
    successor(X1,Y,X2).
delete(t(X,Y,Z)K,t(X1,Y,Z)):- K<Y,delete(X,K,X1).
delete(t(X,Y,Z),K,t(X,Y,Z1)):- K>Y,delete(Z,K,Z1).
delete(void,K,void).

successor(t(void,Y,Z),Y,Z).
successor(t(X,Y,Z),Y1,t(X1,Y,Z)):- successor(X,Y1,X1).

```

Comments:

It is necessary to maintain the order of the elements. When a node has nonempty subtrees (left or right) it is necessary to choose which element to substitute for the node. There are two alternatives: pick the biggest element of the left subtree or pick the least element of the right subtree. This program takes the second option. Within its third clause the right subtree is to be searched, in particular the left leaf. The second clause handles the case of a right subtree with an empty left leaf. The node to be eliminated is substituted by the least mode of the right subtree.

The program ‘successor’ is built upon a search on the left subtree; this is the guarantee that the least element will be found.

In the third clause of program ‘delete’, the leaf ‘X1’ is substituted by ‘X2’, the result of operation ‘successor’ on ‘X1’. Consider the following tree:

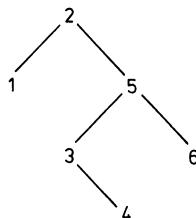


Fig.8

Suppose we want to eliminate node 2. Therefore, we substitute the least node of the right subtree for it, i.e., the node 3:

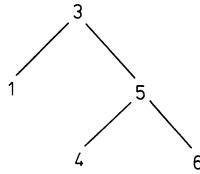


Fig.9

Execution:

```

?- delete(void,a,T).
T=void

?- delete(t(t(void,2,void),3,t(void,4,void)),3,T).
T=t(6(void,2,void),4,void)

?- delete(t(void,1,t(void,2,t(void,3,void))),2,T).
T=t(void,1,t(void,3,void))

?- delete(t(t(void,1,void),2,t(t(void,3,t(void,4,void)),5,
  t(void,6,void)),2,T).
T=t(t(void,1,void),3,t(t(void,4,void),5,t(void,6,void)))
  
```

Consider another example, such as the tree,

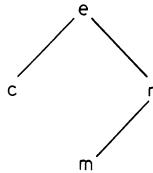


Fig.10

and delete node m. The goal statement,

```
?- binary(Z,m,t(t(void,c,void),e,t(t(void,m,void),n,void))).
```

consists in the call of the previous program ‘binary’. However, the dual programming technique, consisting in shifting the role of the input and output arguments in the goal statement, does not give as complete a solution to the deletion problem as the program ‘delete’.

Problem 39 [Tarnlund 1976b]

Verbal statement:

Write an algorithm for inserting a new node in a three-dimensional tree at the right subtree:

$t(X, r(U, U_1, U_2), Z)$

where X and Z are left and right subtrees respectively and r(U,U1,U2) is a root having the attributes U, U1 and U2.

Suppose that 'U' represents a name, 'U1' an age and 'U2' a profession. The information is to be structured in levels, such as:

Name level:	name,age,profession
Age level:	age,profession,name
Profession level:	profession,name,age

Logic program:

```
insert(t(X,r(U,U1,U2),Z,K,K1,K2,t(X1,r(U,U1,U2),Z))):-  
          K<U, insert(X,K1,K2,K,X1).  
insert(t(X,r(U,U1,U2),Z),K,K1,K2,t(X,r(U,U1,U2),Z1)):-  
          K>U, insert(Z,K1,K2,K,Z1).  
insert(void,K,K1,K2,t(void,r(K,K1,K2),void)).  
insert(t(X,r(K,K1,K2),Z),K,K1,K2,t(X,r(K,K1,K2),Z)).
```

Comments:

This program is identical to the one for the 'binary' relation, with a slight difference. Instead of adopting one key, we now have three alternative keys as can be observed in the second and third clauses.

The built-in predicate '<' must be adopted. But, for the proposed example, we assume that 'name', 'age' and 'profession' are given as lists. So, '<' must be substituted by 'less'.

```
less([],_):- !.  
less([X|L1],[X|L2]) :- !, less(L1,L2).  
less([X|L1],[Y|L2]) :- X<Y.
```

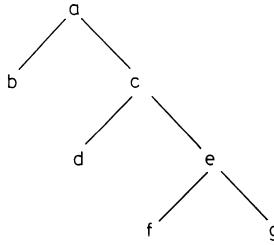
Execution:

```
?- insert(t(void,[a,u,n,e],[35],[e,n,g,i,n,e,e,r]),void),  
          [m,a,r,i,e],[23],[m,a,t,h,e,m,a,t,i,c,i,a,n],x).  
X=t(void,r([a,n,n,e],[35],[e,n,g,i,n,e,e,r]),  
      t(void,r([23,[m,a,t,h,e,m,a,t,i,c,i,a,n]),void))
```

Problem 40 [Bratko 1981, personal communication]

Verbal statement:

Consider a binary tree such as the following one:

**Fig. 11**

Write a program to print out, in order, the nodes of a tree, level by level and in a bottom-up fashion.

Logic program:

```

/* Procedure printtree(T) lists nodes of T,
level by level and bottom up */
printtree(T):- printtrees([T]). 

printtrees([]):- !.
printtrees(Tlist):- roots(Tlist,Roots), subtrees(Tlist,Trees),
                printtrees(Trees), printlist(Roots).

roots([],[]):- .
roots([t(L,X,R)|Tlist],[X|Rlist]):- !, roots(Tlist,Rlist).
roots([X|Tlist],[X|Rlist]):- roots(Tlist,Rlist).

subtrees([],[]):- !.
subtrees([t(L,X,R)|Tlist],[L,R|Rest]):- !, subtrees(Tlist,Rest).
subtrees([X|Tlist],Trees):- subtrees(Tlist,Trees).

printlist([]):- !.
printlist([X|L]):- printnode(X), printlist(L).

printnode(void):- !.
printnode(X):- write(X).
  
```

Execution:

```
?- printtree(tree(t(t(d,b,void),a,t(e,c,t(g,f,void))))).
```

Problem 41

Verbal statement:

Write a program that builds a binary tree from a given list of numbers, and then outputs the list in an ordered way (which constitutes a special path in the tree).

Suggestion:

The way of building the tree is:

- 1) The root is the first member of the list.
- 2) The left branch will consist of the less or equal nodes, and the right branch will consist of the greater nodes.
- 3) Each node is represented by $t(L,N,R)$ where N is the node, L is the left subtree and R is the right subtree.

For example: [3,2,1] will give

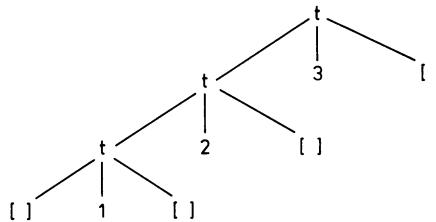


Fig. 12

Logic program:

```

goal(X):- make_tree(X,T),nl,write(T),nl,nl,list_tree(T,L,[]),
          write(L),nl,nl.

make_tree([H|T],t(T1,H,T2)):- split(T,H,U1,U2),
                                make_tree(U1,T1),
                                make_tree(U2,T2).

make_tree([E],t([],E,[])).
make_tree([],[]).

list_tree(t(T1,H,T2),S,X):- list_tree(T1,S,[H|Y]),
                                list_tree(T2,Y,X).

list_tree([],X,X).

split([A|L],C,[A|U1],U2):- A=<C,! ,split(L,C,U1,U2).
split([A|L],C,U1,[A|U2]) :- A>C,! ,split(L,C,U1,U2).
split([],_,[],[]).
  
```

The tree is built up from the root by applying program ‘split’ to fix the elements of the left and right subtrees.

Once more, it is necessary to implement the implicit difference in order to define the concatenation relation. This time it is up to list_tree to handle it. The list of all nodes of the tree ‘ $t(T1,H,T2)$ ’ is ‘ $S-X$ ’.

Execution:

Applying this program to the list:

```
[14,12,16,21,19,9,0,15,13,6,3,2,5,1,4,7,11,18,17].
?- goal([14,12,16,21,19,9,0,15,13,6,3,2,1,4,7,11,18,17]).
t(t(t(t([]),0,t(t(t(t([],1,[]),2,[]),3,t([],4,[])),6,t([],7,[]))),9,t([],11,[])),12,t([],13,[])),14,t(t([],15,[])),16,t(t(t(t([],17,[]),18,[])),19,[])),21,[])
[0,1,2,3,4,6,7,9,11,12,13,14,15,16,17,18,19,21]
?- goal([14,12,16,21,19,9,0,15,13,6,3,2,1,4,7,11,18,17]).
t(t(t([]),0,t(t(t(t([],1,[]),2,[]),3,t([],4,[])),6,t([],7,[]))),9,t([],11,[])),12,t([],13,[])),14,t(t([],15,[])),16,t(t(t(t([],17,[]),18,[])),21,[])
[0,1,2,3,4,6,7,9,11,12,13,14,15,16,17,18,19,21]
```

Problem 42 [Warren et al. 1977c]

Verbal statement:

Generate a list of serial numbers for the items of a given list, the members of which are to be numbered in alphabetical order.

For example, the list [p,r,o,l,o,g] must generate [4,5,3,2,3,1]

Logic program:

```
serialise(L,R):- pairlists(L,R,A), arrange(A,T),
               numbered(T,1,N).

pairlists([X|L],[Y|R],[pair(X,Y)|A]) :- pairlist(L,R,A).
pairlists([],[],[]).

arrange([X|L],tree(T1,X,T2)) :- partition(L,X,L1,L2),
                                arrange(L1,T1),
                                arrange(L2,T2).

arrange([],_).

partition([X|L],X,L1,L2) :- partition(L,X,L1,L2).
partition([X|L],Y,[X|L1],L2) :- before(X,Y),
                                partition(L,Y,L1,L2).
partition([X|L],Y,L1,[X|L2]) :- before(Y,X),
                                partition(L,Y,L1,L2).

partition([],_,[],[]).

before(pair(X1,Y1),pair(X2,Y2)) :- X1 < X2.

numbered(tree(T1,pair(X,N1),T2),NO,N) :- numbered(T1,NO,N1),
                                         N2 is N1+1,
                                         numbered(T2,N2,N).

numbered(void,N,N).
```

Comments:

The relation ‘arrange’ is in charge of building up a tree from a list of nodes. The list ‘L’ is divided into two lists ‘L1’ and ‘L2’ with respect to ‘X’: ‘L1’ has all the elements of ‘L’ less than ‘X’ and ‘L2’ has all the elements of ‘L’ greater than ‘X’. The relation ‘partition’ has four definitions. The first one states that if an element of ‘L’ is equal to ‘X’ the process of dividing ‘L’ must go on, ignoring this element. The second definition states that if an element of ‘L’ is previous to ‘Y’, it must be inserted into ‘L1’, and the process of dividing ‘L’ must go on. The third one states that if ‘Y’ is previous to an element of ‘L’, it must be inserted into ‘L2’, and the process of dividing ‘L’ must go on. The fourth one states that when ‘L’ is empty, ‘L1’ and ‘L2’ are also empty.

The program ‘partition’ is identical to ‘split’, with a slight difference concerning the way to handle the comparison, because we are now dividing a list with an even number of elements.

The relation ‘before’ states that a pair is previous to another when the letter of the first pair is alphabetically inferior to the letter of the second. The relation ‘numbered’ defines the numbering process: to give a number to a binary tree between N0 and N, we start by giving a number to a left subtree T1 between N0 and N1, giving the number N1 to the root. We end by giving a number to the right subtree between N2 and N.

Execution:

```
?- serialise([p,r,o,l,o,g]).  
[4,5,3,2,3,1]  
  
?- serialise([i,n,t,.,a,r,t,i,f,i,c,i,a,l]).  
[5,7,9,1,2,8,9,5,4,5,3,5,2,6]
```

Problem 43

Verbal statement:

Write a program able to sort a list of words alphabetically. The list of words obeys the syntax: word1.word2.wordn.stop. , where “stop” means that there are no more words to sort.

(This problem was suggested by Henri Meloni).

Suggestion: Imagine a tree having each node as a function of 3 arguments ‘nd(L,W,R)’, where ‘L’ represents the left descendant of the node, ‘R’ its right descendant, and ‘W’ the word just read. The input word is read and compared to the word associated to the node, beginning with the root, and afterwards compared recursively to its left or right descendant. The whole process finishes when the descendant is a free variable or when the word already exists in the tree. In the first case, a new node representing the word is created, in the place of the variable.

Example: The list john. mary. james. edward. stop. will generate the tree:

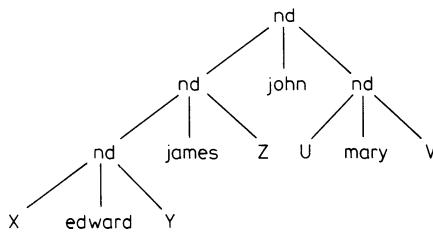


Fig. 13

Logic program:

```

/* Interface */
order(X):- reading(W),name(W,W1),classify(W1,X,Y),order(Y).
order(X):- nl,write('ordered words:'),nl,write_tree(X),nl,nl,
          write('Done.'),nl.

reading(W):- read(W),(W==stop,!),fail;true.

/* Internal mechanisms */
classify(W,nd(L,W,R),nd(L,W,R)):- !.
classify(W,nd(L,W1,R),nd(U,W1,R)):- lower(W,W1),!,
                                         classify(W,L,U).
classify(W,nd(L,W1,R),nd(L,W1,U)):- !,classify(W,R,U).

lower([],_):- !.
lower([X|Y],[X|T]):- !,lower(Y,T).
lower([X|Y],[Z|T]):- X<Z.

write_tree(X):- var(X),!.
write_tree(nd(L,W,R)):- !,write_tree(L),name(W1,W),write(W1),
                      write('. '),write_tree(R).
  
```

Comments:

Observe with care the obsession to use ‘!’ in order to maintain the uniqueness of the solution.

Execution:

To run the program just give the command ‘?-order(X).’, followed by the list of words.

?- order(X).

helder. antonio. luis. jose. fernando. jorge. carlos. stop.

Ordered words:

```
antonio. carlos. fernando. holder. jorge. jose. luis.  
Done.
```

Problem 44 [Warren 1977b]

Verbal statement:

Write a term ‘dictionary’ for representing an alphabetically ordered dictionary pairing English words with French equivalents.

Consider the following example:

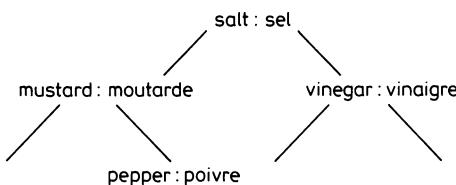


Fig. 14

Write also procedures for:

- ‘lookup’ a name in a dictionary and find its paired value
- ‘lookup’ a name which is before (on the left) a certain name
- ‘lookup’ a name which is after (on the right) a certain name

and take into account that these procedures may also serve to construct the dictionary. The relation ‘lookup’ applies also to symbol tables built upon binary trees, where the values are addresses.

Logic program:

```

dictionary(void).
dictionary(dic(X,Y,D1,D2)):- dictionary(D1),
                                         dictionary(D2).

look(X,dic(A,B,C,D),Y):- dic(A,B,C,D), lookup(X,dic(A,B,C,D),Y).

lookup(Name,dic(Name,Value,_,_),Value):- !.
lookup(Name,dic(Name1,_,Left,_),Value):- Name<Name1,
                                         lookup(Name,Left,Value).
lookup(Name,dic(Name1,_,_,Right),Value):- Name>Name1,
                                         lookup(Name,Right,Value).

dic(salt,sel,
    dic(mustard,moutarde,
        void,
        dic(pepper,poivre,void,void)),
    dic(vinegar,vinaigre,void,void)).
  
```

Execution:

?- look(salt,D,X1).

is interpreted as: construct a dictionary D such that “salt” is paired with X1.

The execution gives:

D=dic(salt,X1,D1,D2)

?- look(mustard,D,X2).

gives as result:

D=dic(salt,X1,dic(mustard,X,D,D4),D2).

The relation ‘lookup’ is in charge of inserting new entries in a partially specified dictionary.

After executing:

?- look(vinegar,D,X3),look(pepper,D,X4),look(salt,D,X5).

D is instantiated to a dictionary which may be pictured as:

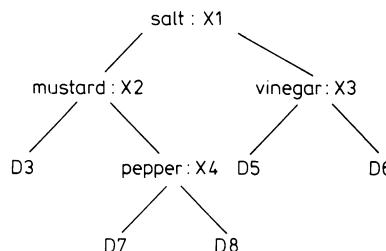


Fig. 15

Problem 45

Verbal statement:

Suppose you have an universe of eight animals

$U=[a1, a2, r1, b1, b2, h1, o1, o2]$

classified in the following way:

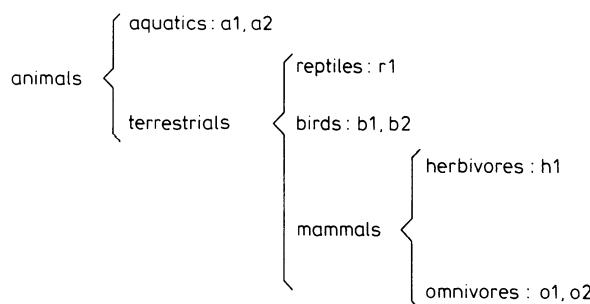


Fig. 16

Write a data structure representing this classification, and build up a program for answering the following questions:

- 1) What is the classification of animal X?
- 2) What animals have the classification Y?

Logic program:

```

classification(A,C):- animal(C,L),member(A,L).

animal(animal(V),X):- aquatic(V,X) ; terrestrial(V,X).

aquatic(aquatic,[a1,a2]).

terrestrial(terrestrial(V),X):- reptile(V,X) ; bird(V,X) ;
mammal(V,X).

reptile(reptile,[r1]).

bird(bird,[b1,b2]).

mammal(mammal(V),X):- herbivore(V,X) ; omnivore(V,X).

herbivore(herbivore,[h1]).

omnivore(omnivore,[o1,o2]).
```

Comments:

The predicate ‘member(X,Y)’ is defined as in Problem 1, and it tests if ‘X’ is a member of list ‘Y’.

The question “1) What is the classification of animal X ?” where ‘X’ is instantiated, corresponds to the command ‘?- classification(X,C).’.

The other question “2) What animals have the classification Y?”, where ‘Y’ is instantiated, corresponds to the command ‘?-animal(Y,X).’.

The execution is top-down, as represented on the tree:

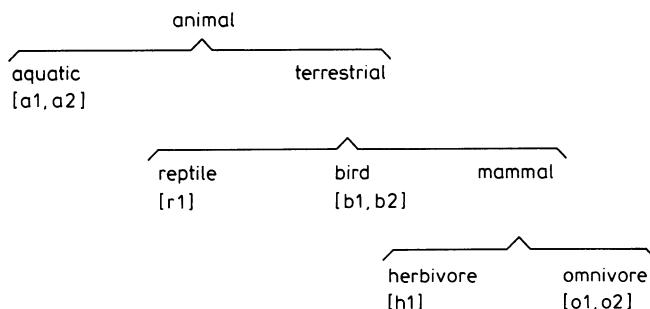


Fig. 17

Remark: This problem was suggested by [Dahl 1977b].

Execution:

```
?- classification(a1,X).
   X=animal(acquatic)

?- classification(h1,X).
   X=animal(terrestrial(mammal(herbivore)))

?- classification(X,animal(acquatic)).
   X=a1;
   X=a2;
   no
```

Chapter 4 Proving with Prolog

Theorem proving is one of those activities able to help us achieve a better understanding of how a symbolic programming language can cope with reasoning tasks. Also, it teaches us how to have in a program a useable map of specific knowledge, and how to simulate intelligent behavior.

Problem 46 [L. Pereira 1976]

Verbal statement:

Write a simple Propositional Calculus Theorem Prover, covering equivalence, implication, disjunction, conjunction and negation.

Logic program:

```
/* Propositional Calculus Theorem Prover */

:-op(700,xfy,eqv).    /* equivalence */
:-op(650,xfy,imp).    /* implication */
:-op(600,xfy,or).     /* disjunction */
:-op(550,xfy, and).   /* conjunction */
:-op(500,fy,not).     /* negation */

formulas:- repeat,read(T), ( T=stop ; theorem(T),fail ).

theorem(T):- ttynl,
           ( false(T), display('not valid'),! ;
             display('valid') ),
           ttynl,ttynl.

false( 'FALSE' ):-! .
false( not 'TRUE' ):-! .

false( P eqv Q ):-false(( P imp Q ) and ( Q imp P )).

false( P imp Q ):-false( not P or Q ).

false( P or Q ):-false( P ),false( Q ).

false( P and Q ):- (false( P ) ; false( Q )).

false( not not P ):-false( P ).
```

```

false( not(P eqv Q)):-false( not(P imp Q) or not(Q imp P)).
false( not(P im Q)):-false (not(not P or Q)).
false( not(P or Q)):-false( not P and not Q).
false( not(P and Q)):-false( not P or not Q).

```

Problem 47

Verbal statement:

Program Wang's algorithm for proving theorems of propositional calculus.

Wang's algorithm consists of writing down a series of lines, each simpler than the previous one, until a proof is completed – or shown to be impossible. Each line consists of any number of well formed formulas (wff), separated by commas, on each side of an arrow.

Wang's algorithm:

- As the first line, write the premises to the left of the arrow and the theorem to the right of the arrow:

premise1, premise2, premise3 \rightarrow theorem

- If the principal connective of a wff is a negation, drop the negation sign and move the wff to the other side of the arrow; for example,

$p \vee q, \neg(r \wedge s), \neg q, p \vee r \rightarrow s, \neg p$

is changed into

$p \vee q, p \vee r, p \rightarrow s, r \wedge s, q.$

- If the principal connective of a wff on the left of the arrow is \wedge or on the right of the arrow is \vee , replace the connective by a comma; for example,

$p \wedge q, r \wedge (\neg p \vee s) \rightarrow \neg q \vee r$

is changed into

$p, q, r, \neg p \vee s \rightarrow \neg q, \neg r$

- If the principal connective of a wff on the left of the arrow is \vee or that on the right of the arrow is \wedge , then produce two new lines, each with one of the two sub-wff's replacing the wff; for example,

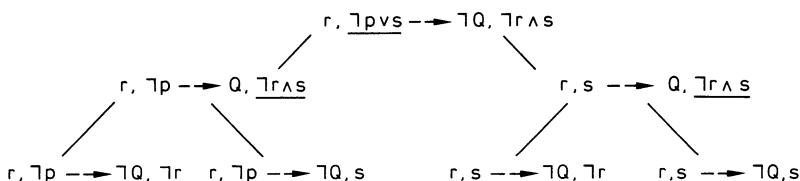


Fig. 1

All the resulting lines must be proved in order to prove the original theorem.

5. If the same wff occurs on both sides of an arrow, the line is proved.
 6. If no connectives remain in a line and no propositional variable occurs on both sides of the arrow, the line is not provable. In fact, assigning all the variables on the left to be TRUE and all the variables on the right to be FALSE will provide an example of the falsity of the theorem.

Logic program 1 [L. Pereira 1979, personal communication]:

```
/* Operators */
:-op(700,xfy,<=>).      /* equivalence */
:-op(650,xfy,=>).        /* implication */
:-op(600,xfy,V).          /* disjunction */
:-op(550,xfy,&).          /* conjunction */
:-op(500,fy,-).           /* negation */

/* Read_in and try to prove formula write; */
/* 'valid' or 'not valid' accordingly */

formulas:- repeat,write('Formula:'),nl,
          read(T),(T=stop; theorem(T),fail).

theorem(T):- nl,nl,
            (prove([]&[]=>[]&[T]),!,nl,
             write('Formula is valid');
              nl,write('Formula is not valid')),nl,nl.

to_prove(T):- write(' prove,'),nl,write(T),nl,nl,
            prove(T).

prove(E1):- rule(E1,E2,Rule),!,
           write(E2),by_rule(Rule),nl,
           prove(E2).

/* Case for V on l.h.s. */

prove(L & [H V I|T] => R):- !,
    first_branch,to_prove(L & [H|T] => R),
    branch_proved,
    second_branch,to_prove(L & [I|T] => R),
    branch_proved.

/* Case for & on r.h.s. */

prove(L => R & [H & I|T]):-!,
    first_branch, to_prove(L => R & [H|T]),
    branch_proved,
    second_branch, to_prove(L => R & [I|T]),
    branch_proved.

/* Case for atom */

prove(L & [H|T] => R):- !,prove([H|L] & T => R).
prove(L => R & [H|T]):- !,prove(L => [H|R]& T).
```

```

/* Finally, check whether tautology */

prove(T):- tautology(T), write('Tautology.'), nl.
prove(_):- write('This branch is not provable.'), fail.

/* Cases where => appears in one of the sides */

rule(L & [H => I|T] => R,
      L & [-H V I|T] => R, rule_5).
rule(L => R & [H => I|T],
      L => R & [-H V I|T], rule_6).

/* Cases where <=> appears in one of the sides */

rule(L & [H <=> I|T] => R,
      L & [(H => I)&(I => H)|T] => R, rule_7).
rule((L => R & [H <=> I|T],
      L => R & [(H => I) & (I => H)|T], rule_8).

/* Cases where - appears */

rule(L & [-H|T] => R & R2,
      L & T => R & [H|R2], rule_2).
rule(L1 & L2 => R & [-H & T],
      L1 & [H|L2] => R & T, rule_2).

/* Case for & on l.h.s. */

rule(L & [H & I|T] => R,
      L & [H, I|T] => R, rule_3).

/* Case for V on r.h.s. */

rule(L => R & [H V I |T],
      L => R & [H, I|T], rule_3).

tautology(L & [] => R & []):- member(M,L),
                                         member(M,R).

branch_proved:- write('This branch has been proved.'), nl.
first_branch:- nl, write('First branch:').
second_branch:- nl, write('Second branch:').
by_rule(R):- write('      by'), write(R), nl, nl.
member(H,[H|_]).  

member(I,[_|T]) :- member(I,T).

```

Execution:

Formula:

a => a.

[] & [] => [] & [-a V a]
[] & [] => [] & [-a,a]
[] & [a] => [] & [a]

by rule_6
by rule_3
by rule_2

Tautology

Formula is valid

Formula:

$$(a \Rightarrow b) \ \& \ (b \Rightarrow c) \Rightarrow (a \Rightarrow c).$$

$[] \ \& \ [] \Rightarrow [] \ \& \ [-((a \Rightarrow b) \ \& \ (b \Rightarrow c)) \vee (a \Rightarrow c)]$	by rule_6
$[] \ \& \ [] \Rightarrow [] \ \& \ [-((a \Rightarrow b) \ \& \ (b \Rightarrow c)), a \Rightarrow c]$	by rule_3
$[] \ \& \ [(a \Rightarrow b) \ \& \ (b \Rightarrow c)] \Rightarrow [] \ \& \ [a \Rightarrow c]$	by rule_2
$[] \ \& \ [(a \Rightarrow b) \ \& \ (b \Rightarrow c)] \Rightarrow [] \ \& \ [-a \vee c]$	by rule_6
$[] \ \& \ [a \Rightarrow b, b \Rightarrow c] \Rightarrow [] \ \& \ [-a \vee c]$	by rule_3
$[] \ \& \ [-a \vee b, b \Rightarrow c] \Rightarrow [] \ \& \ [-a \vee c]$	by rule_5
$[] \ \& \ [-a \vee b, b \Rightarrow c] \Rightarrow [] \ \& \ [-a, c]$	by rule_3
$[] \ \& \ [a, -a \vee b, b \Rightarrow c] \Rightarrow [] \ \& \ [c]$	by rule_2
First branch: prove,	
$[a] \ \& \ [-a, b \Rightarrow c] \Rightarrow [] \ \& \ [c]$	
$[a] \ \& \ [b \Rightarrow c] \Rightarrow [] \ \& \ [a, c]$	by rule_2
$[a] \ \& \ [-b \vee c] \Rightarrow [] \ \& \ [a, c]$	by rule_5
First branch: prove,	
$[a] \ \& \ [-b] \Rightarrow [] \ \& \ [a, c]$	
$[a] \ \& \ [] \Rightarrow [] \ \& \ [b, a, c]$	by rule_2

Tautology.

This branch has been proved.

Second branch : prove,

$$[a] \ \& \ [c] \Rightarrow [] \ \& \ [a, c]$$

Tautology.

This branch has been proved.

Second branch : prove,

$$[a] \ \& \ [b, b \Rightarrow c] \Rightarrow [] \ \& \ [c]$$

$$[b, a] \ \& \ [-b \vee c] \Rightarrow [] \ \& \ [c]$$

by rule_5

First branch : prove,

$$[b, a] \ \& \ [-b] \Rightarrow [] \ \& \ [c]$$

$$[b, a] \ \& \ [] \Rightarrow [] \ \& \ [b, c]$$

by rule_2

Tautology.

This branch has been proved.

Second branch : prove,

$$[b, a] \ \& \ [c] \Rightarrow [] \ \& \ [c]$$

Tautology.

This branch has been proved.

Formula is valid.

Formula:

$(a \Rightarrow b) \Leftrightarrow (\neg a \vee b)$
 $[] \& [] \Rightarrow [] \& [((a \Rightarrow b) \Rightarrow \neg a \vee b) \& (\neg a \vee b \Rightarrow a \Rightarrow b)]$ by rule_8

First branch : prove,

$[] \& [] \Rightarrow [] \& [(a \Rightarrow b) \Rightarrow \neg a \vee b]$	by rule_6
$[] \& [] \Rightarrow [] \& [-(a \Rightarrow b) \vee \neg a \vee b]$	by rule_3
$[] \& [] \Rightarrow [] \& [-(a \Rightarrow b), \neg a \vee b]$	by rule_2
$[] \& [a \Rightarrow b] \Rightarrow [] \& [\neg a \vee b]$	by rule_3
$[] \& [\neg a \vee b] \Rightarrow [] \& [\neg a \vee b]$	by rule_3
$[] \& [\neg a \vee b] \Rightarrow [] \& [\neg a, b]$	by rule_2
$[] \& [a, \neg a \vee b] \Rightarrow [] \& [b]$	by rule_2

First branch : prove,

$[a] \& [\neg a] \Rightarrow [] \& [b]$	by rule_2
$[a] \& [] \Rightarrow [] \& [a, b]$	by rule_2

Tautology.

This branch has been proved.

Second branch : prove,
 $[a] \& [b] \Rightarrow [] \& [b]$

Tautology.

This branch has been proved.

Second branch : prove,

$[] \& [] \Rightarrow [] \& [\neg a \vee b \Rightarrow a \Rightarrow b]$	by rule_6
$[] \& [] \Rightarrow [] \& [-(\neg a \vee b) \vee (a \Rightarrow b)]$	by rule_3
$[] \& [] \Rightarrow [] \& [-(\neg a \vee b), a \Rightarrow b]$	by rule_2
$[] \& [\neg a \vee b] \Rightarrow [] \& [a \Rightarrow b]$	by rule_3
$[] \& [\neg a \vee b] \Rightarrow [] \& [\neg a \vee b]$	by rule_6
$[] \& [\neg a \vee b] \Rightarrow [] \& [\neg a, b]$	by rule_3
$[] \& [a, \neg a \vee b] \Rightarrow [] \& [b]$	by rule_2

First branch : prove,

$[a] \& [\neg a] \Rightarrow [] \& [b]$	by rule_2
$[a] \& [] \Rightarrow [] \& [a, b]$	by rule_2

Tautology.

This branch has been proved.

Second branch : prove,
 $[a] \& [b] \Rightarrow [] \& [b]$

Tautology.

This branch has been proved.

Formula is valid.

Logic program 2 [Bratko 1981, personal communication]:

```

/* Wang's Algorithm */

:-op(700,xfv,<=>).          /* equivalence */
:-op(600,xfv,=>).           /* implication */
:-op(500,xfv,v).             /* disjunction */
:-op(400,xfv,&).            /* conjunction */
:-op(300,fv,-).              /* negation */

wang:- nl,write('Formula: '),nl, read(T),
       (T=stop,!;  prove(T),wang).

prove(L => R):-
    nl, theorem(L&true => R v false),!, /* procedure theorem */
    write(' Formula is a theorem ');        /* requires this */
    write(' Formula is not a theorem ').

prove(T):- prove(true => T).

theorem(T):- write('Prove: '),write(T),nl, (tautology(T);
    perpartes(T);  transf(T,T1),theorem(T1)).

tautology(L => R):- conmember(Exp,L), dismember(Exp,R),!,
    write('This is tautology'),nl.

perpartes(L => R):- concconc(L1,(E1 v E2)&L2,L),
    concconc(L1,L2,LL),
    write('First branch:'),nl,
    theorem(E1&LL => R),
    write('Second branch:'),nl,
    theorem(E2&LL => R).

perpartes(L => R):- disconc(R1,(E1&E2) v R2,R),
    disconc(R1,R2,RR),theorem(L => E1 v RR),
    theorem(L => E2 v RR).

transf(L => R, LL => Exp v R):-      /* negation */
    concconc(L1,-Exp&L2,L),
    concconc(L1,L2,LL).

transf(L => R, Exp & L => RR):- disconc(R1,-Exp v R2, R),
    disconc(R1,R2,RR).

transf(L => R, LL => R):- concconc(L1,(A&B)&L2,L),
    concconc(L1,A&(B&L2),LL).

transf(L => R, L => RR):- disconc(R1,(A v B) v R2, R),
    disconc(R1, A v (B v R2), RR).

transf(L => R, LL => R):- concconc(L1,Exp&L2,L), rule(Exp,Exp1),
    concconc(L1,Exp1&L2,LL).

transf(L => R, L => RR):- disconc(R1,Exp v R2,R),rule(Exp,Exp1),
    disconc(R1,Exp1 v R2,RR).

```

```

/* Rules */

rule(A => B, -A v B).

rule(A <=> B, (-A & -B) v (A & B)).

/* Empty expression on the left is 'true', on the right is
   'false' */

conconc(true,Exp,Exp).

conconc(Term&Exp1,Exp2,Term&Exp3):- concconc(Exp1,Exp2,Exp3).

conmember(Term,Exp):- concconc(Exp1,Term&Exp2,Exp).

disconc(false,Exp,Exp).

disconc(Term v Exp1,Exp2,Term v Exp3):- disconc(Exp1,Exp2,Exp3).

dismember(Term,Exp):- disconc(Exp1,Term v Exp2, Exp).

```

Execution:

Formula:

```

| (a => b) & (b => c ) => (a => c) .
Prove: ((a=>b)&(b=>c))&true=>(a=>c) v false
Prove: (a=>b)&(b=>c)&true=>(a=>c) v false
Prove: (-a v b)&(b=>c)&true=>(a=>c) v false

```

First branch:

```

Prove: -a&(b=>c)&true=>(a=>c) v false
Prove: (b=>c)&true=>a v (a=>c) v false
Prove: (-b v c)&true=>a v (a=>c) v false

```

First branch:

```

Prove: -b&true=>a v (a=>c) v false
Prove: true=>b v a v (a=>c) v false
Prove: true=>b v a v (-a v c) v false
Prove: true=>b v a v-a v c v false
Prove: a&true=>b v a v c v false

```

This is tautology

Second branch:

```

Prove: c&true=>a v (a=>c) v false
Prove: c&true=>a v (-a v c) v false
Prove: c&true=>a v-a v c v false

```

This is tautology

Second branch:

Prove: $b \& (b \Rightarrow c) \& \text{true} \Rightarrow (a \Rightarrow c) \vee \text{false}$
 Prove: $b \& (\neg b \vee c) \& \text{true} \Rightarrow (a \Rightarrow c) \vee \text{false}$

First branch:

Prove: $\neg b \& b \& \text{true} \Rightarrow (a \Rightarrow c) \vee \text{false}$
 Prove: $b \& \text{true} \Rightarrow b \vee (a \Rightarrow c) \vee \text{false}$

This is tautology

Second branch:

Prove: $c \& b \& \text{true} \Rightarrow (a \Rightarrow c) \vee \text{false}$
 Prove: $c \& b \& \text{true} \Rightarrow (\neg a \vee c) \vee \text{false}$
 Prove: $c \& b \& \text{true} \Rightarrow \neg a \vee c \vee \text{false}$

This is tautology

Formula is a theorem

Formula:

```

| 
| - a v - b) <=> -(a & b) .
***syntax error***
- a v - b
***here***
) <=> -(a & b).
| (-a v -b) <=> -(a & b) .

Prove: true&true=>(-av-b<=>-(a&b)) v false
Prove: true&true=>(-(-av-b)& - -(a&b) v (-av-b)& -(a&b)) v false
Prove: true&true=> -(-av-b)& - -(a&b) v (-av-b)& -(a&b) v false
Prove: true&true=> -(-av-b) v (-av-b)& -(a&b) v false
Prove: true&true=>(-av-b) v -(-av-b) v false
Prove: (-av-b)&true&true=>(-av-b) v false

```

This is tautology

Prove: true&true=> -(a&b) v -(-a v -b) v false
 Prove: (a&b)&true&true=> -(-a v -b) v false
 Prove: (-a v -b)&(a&b)&true&true=>false

First branch:

Prove: $\neg a \& (a \& b) \& \text{true} \& \text{true} \Rightarrow \text{false}$
 Prove: $(a \& b) \& \text{true} \& \text{true} \Rightarrow a \vee \text{false}$
 Prove: $a \& b \& \text{true} \& \text{true} \Rightarrow a \vee \text{false}$

This is tautology

Second branch:

Prove: $\neg b \& (a \& b) \& \text{true} \& \text{true} \Rightarrow \text{false}$

Prove: $(a \& b) \& \text{true} \& \text{true} \Rightarrow b \vee \text{false}$

Prove: $a \& b \& \text{true} \& \text{true} \Rightarrow b \vee \text{false}$

This is tautology

Prove: $\text{true} \& \text{true} \Rightarrow \neg(a \& b) \vee (\neg a \vee \neg b) \& \neg(a \& b) \vee \text{false}$

Prove: $\text{true} \& \text{true} \Rightarrow (\neg a \vee \neg b) \vee \neg(a \& b) \vee \text{false}$

Prove: $\neg(a \& b) \& \text{true} \& \text{true} \Rightarrow (\neg a \vee \neg b) \vee \text{false}$

Prove: $\text{true} \& \text{true} \Rightarrow a \& b \vee (\neg a \vee \neg b) \vee \text{false}$

Prove: $\text{true} \& \text{true} \Rightarrow a \vee (\neg a \vee \neg b) \vee \text{false}$

Prove: $\text{true} \& \text{true} \Rightarrow a \vee \neg a \vee \neg b \vee \text{false}$

Prove: $a \& \text{true} \& \text{true} \Rightarrow a \vee \neg a \vee \neg b \vee \text{false}$

This is tautology

Prove: $\text{true} \& \text{true} \Rightarrow b \vee (\neg a \vee \neg b) \vee \text{false}$

Prove: $\text{true} \& \text{true} \Rightarrow b \vee \neg a \vee \neg b \vee \text{false}$

Prove: $a \& \text{true} \& \text{true} \Rightarrow b \vee \neg b \vee \text{false}$

Prove: $b \& a \& \text{true} \& \text{true} \Rightarrow b \vee \text{false}$

This is tautology

Prove: $\text{true} \& \text{true} \Rightarrow \neg(a \& b) \vee \neg(\neg a \vee \neg b) \vee \text{false}$

Prove: $(a \& b) \& \text{true} \& \text{true} \Rightarrow \neg(\neg a \vee \neg b) \vee \text{false}$

Prove: $\neg(a \& b) \& (a \& b) \& \text{true} \& \text{true} \Rightarrow \text{false}$

Prove: $(a \& b) \& \text{true} \& \text{true} \Rightarrow a \& b \vee \text{false}$

This is tautology

Formula is a theorem

Comments:

Logic program 1 is Lisp oriented, while program 2 is more in the logic programming style.

Problem 48

Verbal statement:

The facts:

The maid said that she saw the butler in the living room.

The living room adjoins the kitchen.

The shot was fired in the kitchen, and could be heard in all nearby rooms.

The butler, who had good hearing, said he did not hear the shot.

To prove:

If the maid told the truth, the butler lied.

Logic program:

Use the previous program for Wang's algorithm.

Execution:

Use the following representation:

- p = The maid told the truth
- q = The butler was in the living room
- r = The butler was near the kitchen
- s = The butler heard the shot
- u = The butler told the truth

The premises are written:

- $\neg p \vee q$ (If the maid told the truth, the butler was in the living room.)
- $\neg q \vee r$ (If the butler was in the living room, he was near the kitchen.)
- $\neg r \vee s$ (If he was near the kitchen, he heard the shot.)
- $\neg u \vee \neg s$ (If he told the truth, he did not hear the shot.)

The theorem is written:

- $\neg p \vee \neg u$ (If the maid told the truth, the butler did not.)

The formula input to the program is:

$$(\neg p \vee q) \wedge (\neg q \vee r) \wedge (\neg r \vee s) \wedge (\neg u \vee \neg s) \Rightarrow (\neg p \vee \neg u)$$

Problem 49 [O'Keefe 1983]

Verbal statement:

Write a program to manipulate logical formulae and to perform proofs of theorems, according to the following four rules of inference:

Rule 0: to prove $A \Rightarrow B$, refute $\{A \wedge \neg B\}$.

Rule 1: to refute Set U $\{A \wedge B\}$, refute Set U $\{A \wedge B, A, B\}$

Rule 2: to refute Set U $\{A \wedge B\}$, refute each of

- Set U $\{A \wedge \neg B\}$,
- Set U $\{\neg A \wedge B\}$,
- Set U $\{\neg A \wedge \neg B\}$.

Rule 3: to refute Set U $\{A, \neg A\}$, do nothing.

The program must print the proof steps as it performs them. Choose an appropriate representation for the formulae.

Logic Program:

```

:- public
    go/1,          % quick test using stored problems
    implies/2,      % the prover proper
    timed/2.        % for getting CPU times

:- mode
    add_conjunction(+,+,+),
    expand(+,+,-),
    extend(+,+,+,-,+,-),
    go(+),
    implies(+,+),
    includes(+,+),
    opposite(+,-),
    problem(+,-,-),
    refute(+),
    try(+),
    timed(+,+).

:- op(950, xfy, $).           % disjunction
:- op(850, xfy, &).           % conjunction
:- op(500, fx, +).            % assertion
:- op(500, fx, -).            % denial

implies(Premise, Conclusion):-
    write('Trying to prove that '), write(Premise),
    write(' implies '), write(Conclusion), nl,
    opposite(Conclusion, Denial), !,
    add_conjunction(Premise, Denial, fs([],[],[],[])).

opposite(F0 & G0, F1 \$ G1):-
    opposite(F0, F1), !,
    opposite(G0, G1).
opposite(F1 \$ G1, F0, & G0):-
    opposite(F1, F0), !,
    opposite(G1, G0).
opposite(+Atom, -Atom).
opposite(-Atom, +Atom).

add_conjunction(F, G, Set):-
    write('Expanding conjunction '), write(F & G),
    write(' by Rule 1'), nl,
    expand(F, Set, Mid),
    expand(G, Mid, New), !,
    refute(New).

```

```

expand(Formula, refuted, refuted).
expand(F & G, fs(D,C,P,N), refuted):-
    includes(D, F & G),
    !.
expand(F & G, fs(D,C,P,N), fs(D,C,P,N)):- 
    includes(C, F & G),
    !.
expand(F & G, fs(D,C,P,N), New):- 
    expand(F, fs(D,[F&G|C],P,N), Mid), !,
    expand(G, Mid, New).
expand(F $ G, fs(D,C,P,N), Set):-
    opposite(F $ G, Conj), !,
    extend(Conj, D, C, D1, fs(D1,C,P,N), Set).
expand(+Atom, fs(D,C,P,N), Set):- !,
    extend(Atom, P, N, P1, fs(D,C,P1,N), Set).
expand(-Atom, fs(D,C,P,N), Set):- !,
    extend(Atom, N, P, N1, fs(D,C,P,N1), Set).

includes([Head|Tail], Head):-
    !.
includes([Head|Tail], This):-
    includes(Tail, This).

extend(Exp, Pos, Neg, New, Set, refuted):-
    includes(Neg, Exp),
    !.                                % there is a contradiction
extend(Exp, Pos, Neg, Pos, Set, Set):-
    includes(Pos, Exp),
    !.                                % seen it before
extend(Exp, Pos, Neg, [Exp|Pos], Set, Set).

refute(refuted):-
    write('Contradiction spotted (Rule 3).'), nl.
refute(fs([F1 & G1|D], C, P, N)):- 
    opposite(F1, F0),
    opposite(G1, G0),
    Set = fs(D,C,P,N),
    write('Case analysis on '), write(F0$G0),
    write(' using Rule 2'), nl,
    add_conjunction(F0, G1, Set),
    add_conjunction(F0, G0, Set),
    add_conjunction(F1, G0, Set).

refute(Set):-
    write('Can''t refute '), write(Set), nl,
    fail.

problem( 1, -a, +a).
problem( 2, +a, -a, & -a).

```

```

problem( 3, -a, +to_be $ -to_be).

problem( 4, -a & -a, -a).

problem( 5, -a, +b $ -a).

problem( 6, -a & -b, -b & -a).

problem( 7, -a, -b $ (+b & -a)).

problem( 8, -a $ (-b $ +c), -b $ (-a $ +c)).

problem( 9, -a $ +b, (+b & -c) $ (-a $ +c)).

problem(10, (-a $ +c) & (-b $ +c), (-a & -b) $ +c).

try(N):-
    problem(N, P, C), !,
    implies(P, C).

timed(0, _):- !.

imed(K, N):- (try(N),true), J is K-1, !, timed(J, N).

```

Comments:

Please note that the definitions of ‘public’ and ‘mode’ are only valid for the C-Prolog compiler.

The procedure ‘opposite’ transforms a formula into its opposite. It adopts the tree as the natural data structure in Prolog. All the data structures in the above program were very carefully chosen. Despite using record structures, whenever it made sense to do so, there is a residue of lists left, representing sets.

The procedure ‘decompose’ is responsible for selecting the left or right argument of a formula known to be binary. The set of formulae is partitioned into four subsets: a set of disjunctions, a set of conjunctions, a set of positive literals, and a set of negative literals. Since positive and negative literals are in different lists, there is no need to keep the sign with them. The disjunctions are kept in complemented form because it is necessary to test whether a disjunction and its opposite both exist. Rule 2 is applied because the opposites of the arguments of a disjunction were taken. The following representation was chosen:

fs(Disjunctions,Conjunctions,PositiveAtoms,NegativeAtoms)

for the set of formulae where

fs([d₁,...,d_m], [c₁,...,c_n], [a₁,...,a_p], [b₁,...,b_q])

represents the set:

{[d₁,...,d_m, c₁,...,c_n,...,a_n, +a₁,...,+a_p, -b₁,..., -b_q]}.

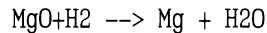
Note that the author started representing the set as a single list, then noticed that he could partition the set, and finally noticed that it was a good idea to store the complements of disjunctions. The structure of the program never changed when he altered this representation: a set was represented by a single term (with a different substructure), and only the code which depends on the contents of a set needed to change.

Problem 50 [L. Pereira 1976]

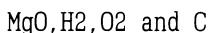
Verbal statement:

The facts:

The following chemical equations:



The existing components:



To prove:

It is possible to compose H_2CO_3 .

Logic program:

Use any of the previous Propositional Calculus theorem provers.

Execution:

The premises are written:

$$(\text{MgO} \wedge \text{H}_2) \Rightarrow (\text{Mg} \wedge \text{H}_2\text{O})$$

$$(\text{C} \wedge \text{O}_2) \Rightarrow \text{CO}_2$$

$$(\text{CO}_2 \wedge \text{H}_2\text{O}) \Rightarrow \text{H}_2\text{CO}_3$$

The theorem is written:

$$\begin{aligned} & ((\text{MgO} \wedge \text{H}_2) \Rightarrow (\text{Mg} \wedge \text{H}_2\text{O})) \\ & \& ((\text{C} \wedge \text{O}_2) \Rightarrow \text{CO}_2) \\ & \& ((\text{CO}_2 \wedge \text{H}_2\text{O}) \Rightarrow \text{H}_2\text{CO}_3) \wedge \text{MgO} \wedge \text{H}_2 \wedge \text{O}_2 \wedge \text{C} \Rightarrow \text{H}_2\text{CO}_3 \end{aligned}$$

Problem 51 [Emden; Kowalski 1976a]

Verbal statement:

Consider a machine defined as a set of commands, each of which is a binary relation over some supposedly given set of states.

In the set notation for the commands of this example, the variables u, v, w, u', v', w' are considered to range over the rational numbers.

The set of states = $\{(u, v, w)\} \cup \{(u, v)\} \cup \{(w)\}$

The commands are presented in Fig. 2.

Set notation for command	"Algol" notation for command
$\{(u, v, w), (u', v', w') : v' = v - 1 \text{ & } w' = u \times w\}$	$v, w := v - 1, u \times w$
$\{(u, v, w), (u', v', w') : u' = u \times u \text{ & } v' = v/2\}$	$u, v := u \times u, v/2$
(E.W. Dijkstra's "parallel assignment")	
$\{(u, v), (u, v, 1)\}$	<u>real</u> $w := 1$
("initializing declaration")	
$\{(u, v, w), (w)\}$	<u>und</u> u, v
("undeclaration")	
$\{(u, 0, w), (u, 0, w)\}$	$v = 0$
$\{(u, v, w), (u, v, w) : \neg v = 0\}$	$\neg v = 0$
("guard")	
$\{(u, v, w), (u', v', w') : \text{even}(v) \text{ & }$	$\text{even}(v); u, v := u \times u, v/2$
$u' = u \times u \text{ & } v' = v/2\}$	
$\{(u, 0, w), (w)\}$	$v = 0; \text{und} u, v$
("guarded command")	

Fig. 2

The machine is some set which is closed under product and contains the above commands.

The program schema =

```
(nonterminals: { S, P, Q }
, terminals: { W1, VO, UV, VW }
, productions: { S --> W1 P
, P --> VO
, P --> UV P
, P --> VW P
}
, start symbol: S
)
```

The program is obtained by the following identifications:

```
W1 = (real w:=1)
VO = (v=0; und u,v)
UV = (even(v); u,v:=u x u, v/2)
VW = (v,w:=v-1, u x w)
```

Some "computations" are presented in Fig.3.

i	Computation 1		Computation 2		Computation 3	
	t_i	x_i	t_i	x_i	t_i	x_i
0	--	(2,10)	--	(2,10)	--	(2,10)
1	W1	(2,10,1)	W1	(2,10,1)	W1	(2,10,1)
2	UV	(4,5,1)	UV	(4,5,1)	UV	(4,5,1)
3	VW	(4,4,4)	VW	(4,4,4)	VW	(4,4,4)
4	UV	(16,2,4)	UV	(16,2,4)	UV	(16,2,4)
5	UV	(256,1,4)	VW	(16,1,64)	VW	(16,1,64)
6	VW	(256,0,1024)	VW	(16,0,1024)	VW	(16,0,1024)
7	V0	(1024)	V0	(1024)	VW	(16, - 1, 2 ⁻¹⁴)
8					VW	(16, - 1, 2 ⁻¹⁸)
						.
						.
						ad inf

Fig.3

Two representations of this machine, the flow graph in Fig.4, and the flow diagram in Fig.5, are given.

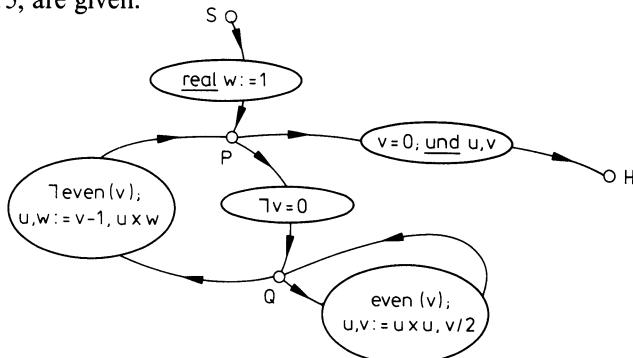


Fig.4

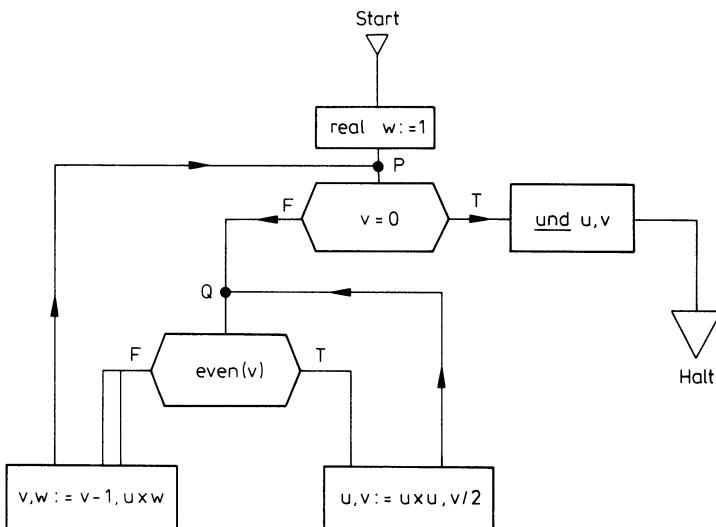


Fig.5

Write a program implementing this machine.

Logic program:

```

:-op(600,yfx,'.').

s(X):- w1(X,Y),p(Y,Z),write(Z),nl.

p(X,Y):- v0(X,Y).

p(X,Z):- uv(X,Y),p(Y,Z).

p(X,Z):- vw(X,Y),p(Y,Z).

w1(U,V,U.V.1).

v0(U.0,W,W).

even(V):- V1 is V mod 2, V1==0.

uv(U.V.W,U1.V1.W):- even(V),
                     U1 is U*U,
                     V1 is V/2.

vw(U.V.W,U.V1.W1):- V1 is V-1,
                     W1 is U*W.

```

Execution:

```
?-s(2.10).
1024
```

Problem 52

Verbal statement:

Write a theorem prover for Euclidean plane geometry. Try the following problem suggested by Gelernter: a diagram is given with four points, A(4,4), B(0,2), C(4,0) and D(5,2); there are five lines BA, BC, BD, DA and DC; two angles are equal, ABD and CBD; there are two right angles, BAD and BCD; it is requested to prove that AD and CD are equal sides.

Logic program [Coelho; L.Pereira 1976]:

```

/* Logic declaration of the problem: file PR1 */
title('Problem 1 Gelernter,1').

/* Diagram */

diagram.

a(4,4).
b(0,2).
c(4,0).
d(5,2).
end.
```

```

/* Given */
lines(ba,bc,bd,da,dc).
ea(abd=cbd).
ra(bad).
ra(bcd).
goal.
esi(ad=cd).

/* Operators */
:-op(400,xfy,'.').
:-op(200,xfy,':').
:-op(500,xfy,'=').
:-op(600,xfy,'!').
:-op(200,yfx,'\'').
:-op(200,xfy,'&').
:-op(300,xfy,'#').

/* Trigger the theorem-prover */
go:- write('Which is the name of the problem file? '),
     read(X), see(X), reading,! , continue(X).

title:- write(X), nl, nl, fail.

reading:- repeat, read(X), X.

continue(X):- see(User), read(Y), see(X), Y.

diagram:- repeat, read(X), (X=end,! , fail;
                           assert(X), fail).

goal:- nl, nl.

/* Input routines */
lines(S):- write('Given the lines: '), lns(S).

lns(L,S):- ! , write(L), write(','),
           separate(L,X.Y.Z), ln1(X|Y,X.Y.Z), lns(S).
lns(L):- write(L), separate(L,X.Y.Z), ln1(X|Y,X.Y.Z).

ln1(P|Q,X.L):- assert_if_desc(point(X)), member(U,L),
              assert_if_desc(d(X|U=P|Q)),
              assert_if_desc(d(U|X=Q|P)), fail.

ln1(DIR,X.L):- ! , ln1(DIR,L).
ln1(DIR,nil).

ea(ANG1=ANG2):- separate(ANG1,X.Y.Z.nil),
               separate(ANG2,U.V.W.nil),
               pea(X.Y.Z=U.V.W!given),
               d(X|Y=AL), d(Y|Z=BE), d(U|V=GA),
               d(V|W=DE),
               assert_ea(AL,BE,GA,DE,given).

```

```

ra(ANG):- separate(ANG,X.Y.Z.nil),
          write('given <'),write(ANG),
          write(' a right angle'),
          d(X\Y=D1),d(Y\Z=D2),opposite(D1.OD1),
          assert_ea(D1,D2,z,OD1,given_right_angle),
          median(X.Y.Z).

median(X.Y.Z):- d(X\Z=D),medra(X.Y.Z),!.
median(X.Y.Z):- perm_2(X.Z!U.V),collinear1(U.T,T.Y),
              d(T\Y=D),medra(T.Y.V),!.
median(_).

medra(X.Y.Z):- collinear1(X.M,M.Z),
               mpdiagram(M,X.Z),ln1(Y|M,Y.M.nil),
               assert_es(Y.M,X.M,median_of_right_angle).

/* Handling the diagram */

mpdiagram(X,Y.Z):- collinear1(Y.X,X.Z),
                  q_dist(X.Y,L),q_dist(X.Z,S),
                  q_dist(Y.Z,T),
                  equal(L,S),prod_r(4,L,G),equal(G.T).

q_dist(X,SQD):- alford(X,P.Q),qdr(P,Q,SQD).

qdr(P,Q,DSSQ):- coord(P,X1,Y1),coord(Q,X2,Y2),
                 dif_abs(X1,X2,DX),dif_abs(Y1,Y2,DY),
                 prod_r(DX,DX,DXSQ),prod_r(DY,DY,DYSQ),
                 sum_r(DXSQ,DYSQ,DSSQ),
                 asserta((qdr(P,Q,DSSQ):- !)). 

coord(P,X,Y):- point(P),G=..[P,X,Y],G,
               asserta(P,X,Y):- !).

/* Equal segment routines */

esi(S1=S2!WHY!database):- checks(S1.S2!P),esdata(S1=S2,WHY),
                           (P=co,!; true).
esi(S1=S2!proved!s1):- gentrionseg(S1,S2,T1,T2,EC1,Ec2),
                       congr(T1!T2),erase(EC1),erase(EC),
                       prtt,
                       pes(S1=S2!by_congruent_triangles),!.

gentrionseg(X.Y,Sz,X.Y.Z,U1.V1.W.EC1,EC2):-
    perm2(ECz,K!nc.c),perm_2(Ec1.L!nc.c),
    exist_triang((X.Y).Z,P,EC1),
    find_w((X.Y).Z,(S2).W),
    construct(P.Z),exist_triang((S2).W,Q,EC2),
    construct(Q.W),perm_2(S2!U1.V1).

exist_triang((X.Y).Z,X,nc):- d(Y\Z=YZ,d(Z\X=ZX),
                                distinct(Yz,ZX)).

exist_triang((X.Y).Z,X,c):- d(Y\Z=D),not(d(X\Z=D1)).
exist_triang((X.Y).Z,Y,c):- d(X\Z=D),not(d(Y\Z=D1)).

```

```

find_w((X.Y).Z,(U.V).W):-
    q_dist(X.Z,XZ),q_dist(Y.Z,YZ),point(W),
    diffe(W,U),diffe(W,V),perm_2(U.V!U1.V1),
    q_dist(W.U1,A),equal(A,XZ),
    q_dist(W.V1,B),equal(B,YZ).

construct(X.Y):- d(X|Y=DXY),!.
construct(X.Y):- diffe(X,Y),asserta(nd(flag)),
               asserta(d(X|Y=X|Y)),asserta(d(Y|X=Y|X)).
construct(X.Y):- diffe(X,Y),erase(c),fail.

erase(c):- retract(d(X)),retract(d(Y)),!,erase_flag.
erase(nc).

erase_flag:- retract(nd(X!Y)),erase_flag,asserta(nd(X!Y)).
erase_flag:- retract(nd(flag)).

congr(T1!T2):-
    eval(trace(seek_congruency_in_ds)&
          con1(T1=T2,dbas.dbas.dbas)
    #trace(seek_congruency_in_general)&con1(T1=T2,W1).

con1(T,W1):- con2(T,W1).
con1(X.Y.Z=U.V.W,W2.W1):- con2(Y.Z.X=V.W.U,W1).
con1(X.Y.Z=U.V.W,W1):- con2(Z.X.Y=W.U.V,W1).

con2(X.Y.Z=U.V.W,W2.W1):- esi(X.Y=U.V!RZ1!W2),
                           con3(X.Y.Z=U.V.W,W1,WHY1),!.
con3(X.Y.Z=U.V.W,W2,WHY1):-
    eai(Y.Z.X=V.W.U!WHY2!W2),confilter(X.Y.Z!U.V.W),
    eval(eai(Z.X.Y=W.U.V!WHY3!W3)&iden(P!ok)!)
    eai(X.Y.Z=U.V.W!WHY3!W3)&iden(P!ok)),
    ccon(X.Y.Z=U.V.W,saa,WHY1.P,WHY2.WHY3).

ccon(T1=T2,WHY,W1,W2):-
    assert_cong2(T1=T2,congruent_triangles,WHY,W1,W2),
    prtt,pc(T1=T2,WHY),cc.

confilter(T1!T2):- bup,dirconfilter(T1!T2).
confilter(T1!T2):- not(bup).

cc:- bup,fail.
cc:- not(bup).

dirconfilter(T1!T2):- notcollinear(T1),notcollinear(T2),
                     dcfilter(T1!T2).

dcfilter(T1!T2):- perm_3(T1,T2,X.Y.Z,X.Z.Y),
                 rejfcg,! ,fail.
dcfilter(T1!T2):- dircono(TR3=TR4),
                 perm_2((T1).T2)!TR1.TR2,
                 perm_3(TR1,TR2,TR3,TR4),
                 rejfcg,! ,fail.

dcfilter(T1!T2):- diff(T1,T2).

```

```

checks((X.Y).(U.V)!co):- diffe(X,va),
                           diffe(Y,va),
                           diffe(U,va),
                           diffe(V,va),!.
checks(S!va).

/* Equal angles routines */
eai(X.Y.Z=U.V.W!WHY!base):- checka(X.Y.Z.U.V.W!P),
                           emda(X.Y.Z=U.V.W!WHY),
                           (P=co,!; true).

/* Database access */
esdata(X.Y=U,identity):- perm_2(X.Y!U),d(X|Y=D).
eadata(X=X,identity).
eadata(X=U,WHY):- wemo(X,Z),wemo(U,Z),eadata1((X).(U),WHY).
emdata(X.Y.Z=U.V.W!WHY):-
    d(X|Y=D1),d(Y|Z=D2),opposite(D2,Od2),
    diffe(D1,OD2),diffe(D1,D2),d(U|V=D3),d(V|W=D4),
    opposite(D4,OD$),diffe(D3,OD4),diffe(D3,D4),
    eval(em1(D1,D2,D3,D4,Y.V,WHY)).
em1(D1,D2,D3,D4,Y.V,RZ):- perm_2(d1.D2!D3.D4),
                           emv1(Y.V,RZ),!.
em1(D1,D2,D3,D4,A,RZ):- can_ang(D1.D2,P),can_ang(D3.D4,Q),
                           eadata(P=Q,RZ),!.
em1(D1,D2,D3,D4,A,both_righ_angles):- pai2(D1.D2),
                                         pai2(D3.D4).

emv1(Y.V,identical).
eadata1(X,RZ):- perm_2(X!X1.U1),emo(X1=U1|RZ),!.
pai2(X.Y.Z):- d(X|Y=D1),d(Y|Z=D2),
              pai2(D1.D2).
pai2(D):- can_ang(D,P),can_comp(D,Q),
          wemo(P,R),wemo(Q,R).

/* Arithmetics */
diffe(X,X):- !,fail.
diffe(X,Y).

diff_abs(X,Y,Z):- inf_r(X,Y),!,sum_r(Y,-X,Z).
diff_abs(X,Y,Z):- sum_r(X,-Y,Z).

inf_r(A,B):- sun_r(B,-A,C),sign(C,1).

sum_r(A,B,C):- decomp(A,S1,A2,A2),decomp(B,S2,S3,B2),
               p_mmc(A2,B2,P,A12,B12),
               N1 is A1*A12,N2 is B1*B12,
               sum(S1,N1,S2,N2,S,N),
               p_mdc(N,P,D,H,K),
               decomp(C,S,H,K).

```

```

sum(S,A,S1,A,1,0):- diffe(S,S1),!.
sum(S,A,S,B,S,C):- !,C is A+B.
sum(S,A,S1,B,S1, ,C):- C is B-A.

sign(-A,-(1)):- !,sign(A,1).
sign(-A,1):- !,fail.
sign(A,-(1)):- !,fail.
sign(0,0):- !.
sign(0,1):- !,fail.
sign(A,1).

decomp(-A,s,A1,A2):- !,decomp(A,S,A1,A2).
decomp(A\B,1,A,B):- diffe(B,1),!.
decomp(A,1,A,1).

p_mmc(A,B,P,C,E):- p_mdc(A,B,D,E,C),Q is A*B,
P is Q/D.

p_mdc(A,B,C,D,E):- B<A,! ,mdc(A,B,C),D is A/C,
E is B/C.
p_mdc(A,B,C,D,E):- mdc(B,A,C),D is A/C,E is B/C.

mdc(A,0,A):- !.
mdc(A,B,D):- C is A mod B, mdc(B,C,D).

prod_r(A,B,C):- decompp(A,S1,A1,A2),decomp(B,S2,B1,B2),
sign_prod(S1,S2,S),C1 is A1*B1,C2 is A2*B2,
p_mdc(C1,C2,D,C11,C12),decomp(C,S,C11,C12).

sign_prod(S,S,1):- !.
sign_prod(S,S1,-(1)).

equal(A,A):- !.
equal(A,B):- sum_r(A,-B,C),sum_r(C,-(1\10),R1),
sum_r(C,1\10,R2),sign(R1,-(1)),sign(R2,1).

/* Cannonical naming */

alf_ord(X,Y,Y,X):- not(perm_2(X.Y!aba.Z)),inf(Y,X),!.
alf_ord(X,X).

can_ang(X\Y.Y\Z,X\Y.Y\Z):- inf_eq(X,Z),!.
can_ang(X\Y.Y\Z,Z\Y.Y\X):- !.
can_ang(X\Y.X\Z,Y\X.Z\X):- inf_eq(Y,Z),!.
can_ang(X\Y.X\Z,Z\X.Y\X):- !.
can_ang(X\Y.Z\Y,X\Y.Z\Y):- inf_eq(X,Z),!.
can_ang(X\Y.Z\Y,Z\Y.X\Y):- !.
can_ang(X\Y.Z\X,Y\X.X\Y):- inf_eq(Y,Z),!.
can_ang(X\Y.Z\X,Z\X.X\Y):- !.
can_ang(X\Y.U\V,X\Y.U\V):- min_4(X,Y,U,V),!.
can_ang(X\Y.U\V,Y\X.V\U):- min_4(Y,X,U,V),!.
can_ang(X\Y.U\V,U\V.X\Y):- min_4(U,X,Y,V),!.
can_ang(X\Y.U\V,V\U.Y\X):- min_4(V,X,Y,U).

can_comp(X\Y.U,Z):- can_ang(Y\X.U,Z).

```

```

/* Handling of equivalence classes */

/* Equal sides */

weso(X,Z):- reso(X=Y),weso(Y,Z),!.
weso(X,X):- not(perm_2(X!aba.Z)).

aeso(Z,X):- reso(Y=X),aeso(Z,Y),!.
aeso(X,X).

/* Equal angles */

wemo(X,Z):- remo(X=Y),wemo(Y,Z),!.
wemo(X,X).

aemo(Z,X):- remo(Y=X),aemo(Z,Y),!.
aemo(X,X).

/* Handling of points */

q_dist(X,SQD):- alf_ord(X,P,Q),qdr(P,Q,SQD).

qdr(P,Q,DSSQ):- coord(P,X1,Y1),coord(Q,X2,Y2),
               diff_abs(X1,X2,DX),diff_abs(Y1,Y2,DY),
               prod_r(DX,DX,DXSQ),prod_r(DY,DY,DYSQ),
               sum_r(DXSQ,DYSQ,DSSQ),
               asserta((qdr(P,Q,DSSQ):- !)).

coord(P,X,Y):- point(P), G=..[P,X,Y],G,
              asserta((coord(P,X,Y):- !)).

/* Asserting routines */

assert_if_desc(X):- X,!.
assert_is_desc(X):- assert(X).

assert_ea(X1,X2,Y1,Y2,WHY):- can_ang(X1,X2,X),
                           can_ang(Y1,Y2,Y),
                           wemo(X,Z),wemo(Y,W),
                           aemo(T,Y),can_comp(X,U),
                           aemo(S,U),can_comp(Y,V),
                           wemo(V,Z1),
                           asserta(emo(X=Y!WHY)),
                           asserta(emo(U=V!
                           supplementary_angles)),
                           asserta(remo(Z=T)),
                           asserta(remo(Z1=S)),!.

assert_es(X1,Y1,WHY):- alford(X1,X),alford(Y1,Y),
                      weso(X,Z),weso(Y,W),diff(Z,W),aeso(T,Y),
                      asserta(eso(X=Y!WHY)),
                      asserta(eso(reso(Z=T))),!.

assert_congr2(T1=T2,RZ,RZ1,RZs,RZA):-
    pet(T1=T2,RZ,RZ1,RZs,RZA),
    assert_congr(T1=T2,RZ).

```

```

assert_congr(X.Y.Z=U.V.W,RZ):-
    d(X|Y=D1), d(Y|Z=D2), d(Z|X=D3), d(U|V=D4),
    d(V|W=D5), d(W|U=D6),
    assert_ea(D1,D2,D4,D5,RZ), assert_ea(D2,D3,D5,D6,RZ),
    assert_ea(D3,D1.D6,D4,RZ), assert_es(X.Y,U.V,RZ),
    assert_es(Y.Z,V.W,RZ),
    assert_es(Z.X,W.U,RZ),
    asserta(congr(X.Y.Z=U.V.W)).

/* Printing routines */

pea(A1=A2!RZ):- nl,write(' '),write('<'),
              write_list(A1),write(' =<'),write_list(A2).

pes(S1=S2!RZ):- nl,write(' '),
              write_list(S1),write('='),write_list(S2),
              write('           '),write_r(RZ).

prtt:- nl,write('therefore: ').

pet(X.Y.Z=U.V.W,saa,W1.P.W2.W3):-
    pes(X.Y=U.V!W1), pea(Y.Z.X=V.W.U!W2),
    eval(iden(P!nok)&pea(Z.X.Y=W.U.V!W3)!)
        ident(P!ok)&pea(X.Y.Z=U.V.W!W3)). 

pc(T1=T2,RZ):- write('triangle '),
              write_list(T1),
              write(' congruent to triangle '),
              write_list(T2),write_r(RZ).

write_list(X.L):- !,write(X),write_list(L).
write_list(X):- write(X).

trace(X):- traceflag,! ,nl,write_t(X).
trace(_).

write_t(X-Y):- !,write(X),write('-'),write_t(Y).
write_t(X.Y):- !,write(X),write('.'),write_t(Y).
write_t(Y):- write(Y),nl.

write_r(X):- write('           '),write_t(X).

/* Utilities */

member(X,nil):- !,fail.
member(X,X.L).
member(X,Y.L):- !,member(X,L).
member(X,X).

separate(X,L):- name(X,L1),merge(L1,L).

merge([A|L],B,S):- name(B,[A]),merge(L,S).
merge([],nil).

perm_2(X!X).
perm_2(X.Y!Y.X).

```

```

opposite(X\Y, Y\X).

inf_eq(X,Z):- inf(X,Z),!.
inf_eq(X,X).

inf(X,Z):- integer(X),integer(Z),!,X<Z.
inf(X,Z):- name(X,X1),name(Z,Z1),X1<Z1.

min_4(X,Y,U,V):- inf(X,Y),inf(X,U),inf(X,V).

collinear1(X,Y,U,V):- d(X\Y=D),d(U\V=D).

distinct(D1,D2):- sen_eq_op(D1,D2),!,fail.
distinct(D1,D2).

sen_eq_op(X\Y,U\V):- perm_2(X.Y!U.V).

eval(X!Y):- eval(X),!.
eval(X#Y):- eval(X).
eval(X&Y):- eval(X),eval(Y).

eval(X!Y):- eval(Y).
eval(X#Y):- eval(Y).
eval(X):- X.

```

Execution:

?- go.

Which is the name of the problem file?

PR1.

Problem 1 Gelernter, 1

given lines: ba,bc,bd,da,dc

<abd = <cbd given

given <bad is a right angle

given <bcd is a right angle

ready

ad=cd to - be - proved

proof:

top-down search:

db=db	identify
<bad = <bcd	both - right - angles
<dba = <dbc	given

therefore: triangle DBA congruent to triangle DBC SAA

therefore:

ad=cd	by - congruent - triangles
	q.e.d.

Comments:

This program answers two basic questions: 1) how to formulate a problem (the choice of a representation scheme and the adoption of canonical naming); and 2) how to identify construction strategies from some knowledge domain. The first issue has a particular influence on the computation time.

The statement of a problem in geometry is done by its (optional) diagram, the hypotheses and the goal. The geometric diagram is a set of points, defined by their cartesian coordinates. Its declaration is optional for the user with minor changes to the program; when it is present it aids in the proof of the goal. The diagram, however, is only a particular case of a whole class of geometric figures for which the problem (theorem) in question must be true. The diagram works mostly as a source of counter-examples for pruning unprovable goals, and so proofs need not depend on it: a proof of a theorem can be carried out without the use of a diagram. However, the diagram may also be used in a guiding way.

A general and flexible representation for the three basic geometric primitives used – segments, directions and angles – is needed to encode them, since these primitives form the basis of any geometric knowledge to be added to the database. The flexibility and generality are achieved by the concept of equivalence class, which allows, for example, that one direction be represented by any other element of its equivalence class. Each angle segment becomes defined by two points and its direction: (A:B) means the direction from A to B in segment AB. One direction can be defined by any pair of points belonging to the same equivalence class.

The canonical naming routines are a set of rewrite rules, applied to an expression to get it into some standard format. They reduce the ambiguity resulting from the syntactic variations of the thing named. In fact, canonical naming is a technique for overcoming combinatorial problems and making the database inquiry easy and fast. This elimination of redundant searching is also achieved by the database organization.

In geometry, combinatorial problems are very common, partly because of transitivity, when equality and congruence relations are involved. For example, if triangle ABC is congruent to triangle DEF one can store this fact in 72 variations. When during the proof it becomes necessary to retrieve the fact that triangle EFD is congruent to triangle BCA, the fact can be just one of the possibilities of that set of variations. This is done with the use of canonical names in the geometric primitives for segments, directions and angles, i.e. a standard variation representing the thing named.

For segments, the endpoints are ordered alphabetically. In the following example, segment CA would be represented as AC. For segment AC, the representation would be AC itself.

The case of an angle, DEH for example, is dealt with in the following way: as $D < H$ the canonical name of angle DEH(D:E:E:H) is D:E:E:H

For angle HED, as $H > D$, the canonical name is not H:E:E:D, but it is D:E:E:H (there is an inversion of pairs and an inversion of each pair). In fact angles DEH and HED are the same and, thus, they have the same canonical name.

The case of an angle defined by two directions, D1.D2 (e.g. A:B.C:D) expressed by different points is dealt with in another way: as A is the least of four points A, B, C, D, i.e. $A < B$, $A < C$ and $A < D$, the canonical name of the angle A:B.C:D is A:B.C:D.

Chapter 5 Doing Arithmetic with Prolog

Prolog is not a suitable language for expressing arithmetic calculations, or number crunching, because is not efficient. However, the following exercises show that it is possible to program arithmetic expressions based upon the Prolog arithmetic built-in predicates.

Problem 53

Verbal statement:

Describe the arithmetic laws using the definition of successor.

Logic program:

```
addition(X, 0, X).  
addition(X, s(Y), s(Z)):- addition(X, Y, Z).  
  
nought(_, 0, 0).  
nought(X, s(Y), P):- nought(X, Y, W), addition(W, X, Z).
```

Execution:

```
?- addition(s(s(s(0))), s(s(0)), S).  
S=s(s(s(s(s(0)))))
```

Problem 54

Verbal statement:

Find the absolute value of an integer.

Logic program:

```
absolute(X, X):- X >= 0, !.  
absolute(X, Y):- Y is -X.
```

Problem 55

Verbal statement:

Write a program to calculate the Fibonacci numbers.

Logic program:

```
fibonacci(0,1).  
fibonacci(1,1).  
fibonacci(X,F):- Z is X-2, Y is Z+1, fibonacci(Y,Y1),  
                  fibonacci(Z,Z1), F is Y1+Z1.
```

Comments:

The Fibonacci number generator is usually taken as an example of data-flow computation.

Therefore, this problem suggests that Prolog is associated with a non Von Neumann computational model. Observe how the retention of some of the history of computation is captured within the program.

Problem 56

Verbal statement:

Define factorial.

Logic program 1:

```
factorial(0,1).  
factorial(N,F):- M is N-1, factorial(M,G),  
                 F is N*G.
```

Logic program 2:

```
/* With recursion on the right */
recursive_factorial(N,F):- fact(N,1,F).

fact(0,F,F).
fact(N,X,F):- M is N-1, Y is X*N,
             fact(M,Y,F).
```

Logic program 3:

Problem 57

Verbal statement:

Write a generator of integers.

Logic program:

```
gen_integer(0).
gen_integer(I):- gen_integer(X), I is X+1.
```

Execution:

```
?- gen_integer(X).
X=0;
X=1;
X=2;
.
.
.
```

Problem 58 [adapted from Bruynooghe 1978]

Verbal statement:

Generate the primes in the first N integers and call them primes(N,CP).

Logic program:

```
primes(N,[1|LP]):- M is N-1,
                  integers(2,M,LI),
                  sift(LI,LP).

integers(N,0,[N]).
integers(N,M,[N|LI]):- R is M-1,
                      Q is N+1,
                      integers(Q,R,LI).

sift([],[]).
sift([P|LI],[P|LP]):- filter(P,LI,NLI),
                     sift(NLI,LP).

filter(P,[],[]).
filter(P,[N|LI],[N|NLI]):- divide(P,N,false),
                           filter(P,LI,NLI).

filter(P,[N|LI],NLI):- divide(P,N,true),
                     filter(P,LI,NLI).

divide(P,N,true):- 0 is N-P*(N/P).
divide(P,N,false).
```

Comments:

The method is the “sieve of Eratosthenes”. It consists in generating the list of integers from 2 to M, and in deleting from that list all the integers which are a multiple of some element in the list.

Problem 59**Verbal statement:**

Define Euclid's algorithm: maximum common divisor and minimum common multiple.

Logic program:

```
mcd(N, 0, N).
mcd(N, M, D) :- R is N mod M, mcd(M, R, D).
mcm(N, M, E) :- mcd(N, M, D), E is (N*M)/D.
```

Problem 60**Verbal statement:**

Compute the maximum of two numbers.

Logic program:

```
max(X, X, X).
max(X, Y, Y) :- X < Y.
max(X, Y, X) :- Y < X.
```

Problem 61**Verbal statement:**

Write a counter for integers from 1 to N.

Logic program:

```
count_till(1, 1) :- !.
count_till(N, Result) :- N1 is N-1, count_till(N1, Result1),
                     Result is Result1+N.
```

Execution:

```
?- count_till(4, R).
R=10;
no
```

Comments:

The result is obtained by adding: $4 + 3 + 2 + 1 = 10$

Problem 62

Verbal statement:

Write a translator for arithmetic expressions without parentheses, from infix to postfix notation. Don't handle expressions which start with '-'.

Example:

the expression $a*b+c*d-e:f$ is to be translated into
 $ab*cd*+ef:-$

Suggestion: use three lists for the representation, one for the infix expression, the other for the result and the third one for saving the operators (auxiliary list).

Logic program:

```

inf_suf([],[],[]).
inf_suf([0|I],S,[]):- operator(0),!,inf_suf(I,S,[0]).
inf_suf([01|I],[02|S],[02|X]):- operator(01),priority(01,N1),
                                priority(02,N2),N1=<N2,!,
                                inf_suf([01|I],S,X).
inf_suf([01|I],S,[02|X]):- operator(01),priority(01,N1),
                                priority(02,N2),N1>N2,
                                inf_suf(I,S,[01,02|X]).
inf_suf([A|I],[A|S],X):- inf_suf(I,S,X).
inf_suf([],X,X).

operator(*).           priority(*,2).
operator(/).           priority(/,2).
operator(+).           priority(+,1).
operator(-).           priority(-,1).

```

Comments:

The program entails the following rules:

- 1) after reading the operator, a translation is followed
- 2) during the reading operation of any operator the list of previous operators is examined:
 - if the previous operator has a priority greater than or equal to the current operator, then it is translated to the output list (3rd clause)
 - if not, then it is added to the auxiliary list (2nd and 4th clauses)
- 3) at the end, the operators that are inside the auxiliary list are translated to the output list, in reverse order (6th clause).

Execution:

```
?- inf_suf([a,* ,b,+ ,c,* ,d,- ,e,/ ,f],L).
L= [a,b,* ,c,d,* ,+ ,e,f,/,-]
?- inf_suf([x,+ ,y,* ,z,/ ,w],L).
L= [x,y,z,* ,w,/ ,+]
```

Problem 63 [Baxter 1980]

Verbal statement:

Write a program to solve the following problem from recreational mathematics: what number ending in 5 can be multiplied by 5 by moving the last 5 to the front?

Illustrate the program by verifying whether half of 6318 is 3195, by calculating twice 9135 and half of 1984, and by finding the solution of the cryptarithmetic problem:

$$35a2b/2 = 1ab64.$$

Note that we use the symbol ‘/’ for the division operation.

Logic program:

```
:op(500,xfx,'-').

half(0,[]-[],[]-[]).
half(C,[A|X]-U,[B|Y]-V):- h(C,A,B,D),
                               half(D,X-U,Y-V).

h(0,0,0,0).
h(0,1,0,1).
h(0,2,1,0).
h(0,3,1,1).
h(0,4,2,0).
h(0,5,2,1).
h(0,6,3,0).
h(0,7,3,1).
h(0,8,4,0).
h(0,9,4,1).
h(1,0,5,0).
h(1,1,5,1).
h(1,2,6,0).
h(1,3,6,1).
h(1,4,7,0).
h(1,5,7,1).
h(1,6,8,0).
h(1,7,8,1).
h(1,8,9,0).
h(1,9,9,1).
```

Execution:

```
?- half(0,X-[ ],[0,5|X]-[5,0|U]).  
X=102040816326530612244897959183673469387755  
  
?- half(0,[6,3,1,8]-[],[3,1,9,5]-[]).  
yes  
  
?- half(C,X,[9,1,3,5]-[]).  
C=1  
X=8270  
  
?- half(0,[1,9,8,4]-[],X).  
X=992  
  
?- half(0,[3,5,A,2,B]-[],[1,A,B,6,4]-[]).  
A=7  
B=8
```

Comments:

The problem can be phrased in the following equation:

$$\begin{aligned} d_1d_2\dots d_n \times 5 &= 5d_1d_2\dots d_n \\ \text{or} \\ d_1d_2\dots d_n 50/2 &= 5d_1d_2\dots d_n \end{aligned}$$

Where d_i are the digits in a decimal representation.

The above program is a good example of the versatility of predicates in Prolog: the solution of this problem in a procedural language requires an algorithm specific to the problem, and in the declarative language Prolog the solution arises from a general predicate!

A number such as 1984 is represented as [1,9,8,4]. A predicate ‘half(C,X,Y)’ is used with the following meaning: with a carry of ‘C’, half of ‘X’ equals ‘Y’.

For example, with a carry of 1, half of 28 equals 64, or ‘half(1,[2,8]-[],[6,4]-[])’. Another predicate ‘h(C,A,B,C1)’ is used with the following meaning: with a carry of ‘C’, half of ‘A’ equals ‘B’ with a carry of ‘C1’.

Problem 64 [Abreu 1984, personal communication]

Verbal statement:

Write a program to optimize arithmetic expression trees taking into account commutative and associative laws, and not assuming non-trivial arithmetic identities (e.g. $a + 0 = a$). Write a code generator for your favorite computer.
(Hint: use a Sethi-Ullman register allocator).

Logic program:

```
%  
%  
% sethi  
%  
% Sethi-Ullman expression tree optimizer and code generator.  
%  
%  
%  
% Topmost user interface.  
%  
  
compile(Expression, NRegs):-  
    do(Expression, Tree),  
    code(Tree, NRegs).  
  
%  
%  
% Expression tree construction.  
%  
%  
%  
% do(Expression, Labeled_Tree).  
%  
  
do(Expression, Labeled_Tree):-  
    tree(Expression, Tree),  
    assoc_tree(Tree, Associative_Tree),  
    cost(Associative_Tree, Labeled_Tree).  
  
%  
%  
% tree(Expression, Tree).  
%  
  
tree(X, [leaf, X]):- atomic(X).  
  
tree(X + Y, [aop(+), A, B]):- tree(X, A), tree(Y, B).  
tree(X - Y, [aop(-), A, B]):- tree(X, A), tree(Y, B).  
tree(X * Y, [aop(*), A, B]):- tree(X, A), tree(Y, B).  
tree(X / Y, [aop(/), A, B]):- tree(X, A), tree(Y, B).  
  
%  
%  
% assoc_tree(Tree, Associative_Tree).  
%
```

```

assoc_tree([Op,[Op|Left],[Op|Right]],[assoc(Op)|List]) :-
    associative(Op),
    assoc_tree([Op | Left], [_ | NLeft]),
    assoc_tree([Op | Right], [_ | NRight]),
    concat(NLeft, NRight, List).

assoc_tree([Op, [Op | Left], Right], [assoc(Op) | List]) :-
    associative(Op),
    assoc_tree([Op | Left], [_ | NLeft]),
    assoc_tree(Right, NRight),
    concat(NLeft, [NRight], List).

assoc_tree([Op, Left, [Op | Right]], [assoc(Op) | List]) :-
    associative(Op),
    assoc_tree(Left, NLeft),
    assoc_tree([Op | Right], [_ | NRight]),
    concat([NLeft], NRight, List).

assoc_tree([Op, Left, Right], [Op, NLeft, NRight]) :-
    assoc_tree(Left, NLeft),
    assoc_tree(Right, NRight).

assoc_tree([leaf, X], [leaf, X]).  

%  

%  

% cost(Associative_Tree, Lableed_Tree).  

%  

% Evaluates the cost of some tree performing commutative swaps  

% wherever appropriate.  

%  

cost([leaf, Value], [1, leaf, Value]).  

cost([assoc(Op) | Operands], Tree) :-  

    sort_operands(Operands, Associative_Node),
    make_tree(Op, Associative_Node, Tree).

cost([Op, Left, [leaf, Right]], [Cost, Op, [Cost | NLeft],
    [0, leaf, Right]]):-  

    cost(Left, [Cost | NLeft]).  

cost([Op, [leaf, Left], Right], [Cost, Op, [Cost | NRight],
    [0, leaf, Left]]):-  

    commutative(Op),
    cost(Right, [Cost | NRight]).  

cost([Op, Left, Right], [Cost, Op, [CostL | NLeft],
    [CostR | NRight]]):-  

    cost(Left, [CostL | NLeft]),
    cost(Right, [CostR | NRight]),
    actual_cost(Op, CostL, CostR, Cost).  

%  

%  

% sort_operands(Node_List, Associative_Node).  

%  

% Sort a list of trees by decreasing value of costs.  

%

```

```

sort_operands(Node_List, Associative_Node) :-
    setup_costs(Node_List, Weighted_List),
    sort(Weighted_List, Sorted_Associative_Node),
    fix_initial_cost(Sorted_Associative_Node, Associative_Node).

%
%
% setup_costs(Node_List, Weighted_Node_List).
%
% Setup the costs for a list of tree nodes.
%

setup_costs([], []).
setup_costs([[leaf,X]|Tail], [[0,leaf,X]|Weighted_Tail]) :-
    setup_costs(Tail, Weighted_Tail).
setup_costs([Node|Tail], [Weighted_Node|Weighted_Tail]) :-
    cost(Node, Weighted_Node),
    setup_costs(Tail, Weighted_Tail).

%
%
% make_tree(Op, Associative_Node, Tree).
%
% Build an ordinary binary tree from an associative tree node.
%

make_tree(Op, [Tree], Tree).
make_tree(Op, [[CostL|Left] | Right],
          [Cost, Op, [CostR|NRight], [CostL|Left]]):-
    make_tree(Op, Right, [CostR|NRight]),
    actual_cost(Op, CostL, CostR, Cost).

%
%
% fix_initial_cost(Old, New).
%
% All this does is make sure that if the leftmost descendant
% in an associative node is a leaf,
% it will get a weight of 1 and not zero.
% Note: the leftmost descendant is the last element of the
% associative node list.
%

fix_initial_cost([], []).
fix_initial_cost([[0, leaf, Value]], [[1, leaf, Value]]).
fix_initial_cost([X|L], [X|T]) :- fix_initial_cost(L, T).

```

```

%
%
% actual_cost(Op, CostL, CostR, Cost).
%
% Evaluate the actual cost of a tree node, given the costs of
% its sub-trees and the operator itself.
%

actual_cost(Op, CostL, CostR, Cost):-
    op_cost(Op, X),
    CostL_1 is CostL + 1,
    CostR_1 is CostR + 1,
    max(CostL, CostR_1, Max1),
    max(CostL_1, CostL, Max2),
    min(Max1, Max2, Min),
    max(Min, X, Cost).

%
%
% Code generation.
%

code(Tree, NRegs):-
    code1(Tree, 0, NRegs).

%
%
% code1(Tree, Regs).
%

code1([1, leaf, Value], R, NRegs):-
    m_move(Value, reg(R)).
code1([1, Op, Left, [0, leaf, Value]], R, NRegs):-
    code1(Left, R, NRegs),
    m_op(Op, Value, reg(R)).

code1([Cost, Op, [LCost | Left], [RCost | Right]], R, NRegs):-
    LCOST >= NRegs, RCOST >= NRegs,
    code1([RCOST | Right], R, NRegs),
    m_move(reg(R), '-(sp)'), 
    code1([LCOST | Left], R, NRegs),
    m_op(Op, '(sp)+', reg(R)).
code1([Cost, Op, [LCOST | Left], [RCOST | Right]], R, NRegs):-
    RCOST > LCOST -> (
        code1([RCOST | Right], R, NRegs),
        R_1 is R+1
        code1([LCOST | Left], R_1, NRegs),
        m_op(Op, reg(R), reg(R_1)),
        m_move(reg(R_1), reg(R))
    )

```

```

;  (
    code1([LCost | Left], R, NRegs), (
        RCost = 0 -> (
            Right = [leaf, Value],
            m_op(Op, Value, reg(R))
        )
        ;
        (
            R_1 is R+1
            code1([RCost | Right], R_1, NRegs),
            m_op(Op, reg(R_1), reg(R))
        )
    )
).
%  

%  

% Database.  

%  

commutative(aop(+)).  

commutative(aop(*)).  

associative(aop(+)).  

associative(aop(*)).  

op_cost(aop(_), 1).  

%  

%  

% Machine code.  

%  

m_move(X, Y):-      m_op(mov,X,Y).  

m_push(X):-          m_op(mov,X,'-(sp)').  

m_pop(X):-          m_op(mov,'(sp)+',X).  

m_op(Op, X, Y):-     m_opc(Op), m_val(X), write(','), m_val(Y), nl.  

m_opc(mov):-          write(' mov      ').  

m_opc(aop(+)):-        write(' add      ').  

m_opc(aop(-)):-        write(' sub      ').  

m_opc(aop(*)):-        write(' mul      ').  

m_opc(aop(/)):-        write(' div      ').  

m_val(reg(N)):-        write('r'), write(N), !.  

m_val(I):-              integer(I), write('#'), write(I), !.  

m_val(X):-              write(X), !.  

%  

% Utilities.  

%

```

```

max(A,B,A):- A >= B, !.
max(A,B,B):- !.

min(A,B,B):- A >= B, !.
min(A,B,A):- !.

concat([], L, L).
concat([X|T], L, [X|R]) :- concat(T, L, R).

```

Execution:

```

| ?- compile((1+2)*(3+4),2).
    mov      #1,r0
    add      #2,r0
    mov      #3,r1
    add      #4,r1
    mul      r1,r0

yes
| ?- compile((1/2)-(3/4),2).
    mov      #1,r0
    div      #2,r0
    mov      #3,r1
    div      #4,r1
    sub      r1,r0

yes
| ?- compile(1*(2-3/(4+5*6)+7)*8-9,2).
    mov      #3,r0
    mov      #5,r1
    mul      #6,r1
    add      #4,r1
    div      r1,r0
    mov      #2,r1
    sub      r0,r1
    mov      r1,r0
    add      #7,r0
    mul      #8,r0
    mul      #1,r0
    sub      #9,r0

yes
| ?- compile(1*(2-3/(4+5*6)+7)*8-9,1).
    mov      #5,r0
    mul      #6,r0
    add      #4,r0
    mov      r0,-(sp)
    mov      #3,r0
    div      (sp)+,r0

```

```

mov    r0,-(sp)
mov    #2,r0
sub    (sp)+,r0
add    #7,r0
mul    #8,r0
mul    #1,r0
sub    #9,r0

```

Comments:

Observe the way this program was written. The style is not logic oriented. The influence of C is heavy!

However, the program is in pure Prolog (only the logic component of the algorithm was considered!).

The predicate ‘assoc_tree’ takes a binary tree, and changes it into an associative tree. The predicate ‘cost’ implements the labeling function defined in the Sethi-Ullman algorithm. The predicates ‘sort_operands’, ‘setup_costs’, ‘make_tree’ and ‘fix_initial_cost’ are used to convert associative nodes back to the binary tree form.

Problem 65**Verbal statement:**

Specify the distribution of “*” by “+”.

Logic program:

```

multiply(X+Y,Z,X1+Y1):- multiply(X,Z,X1),
                           multiply(Y,Z,Y1).

multiply(Z,X+Y,X1+Y1):- multiply(Z,X,X1),
                           multiply(Z,Y,Y1).

multiply(X,Y,X*Y).

```

Problem 66 [Warren 1975 a]**Verbal statement:**

Write a program to find out if a year has 365 or 366 days.

Logic program:

```

:-op(300,fx,'no_of_days_in').
:-op(500,xfy,'IS').

366 'IS' no_of_days_in Y:- 0 is Y mod 4,
                           not(0 is Y mod 100),!.
365 'IS' no_of_day_in Y.

```

Execution:

```
?-X 'IS' no_of_days_in 1975, write(X).
365
?-X 'IS' no_of_days_in 1976, write(X).
366
?-X 'IS' no_of_days_in 2000, write(X).
365
```

Problem 67

Verbal statement:

Define an arithmetic for rational expressions.
 (This problem was suggested by Henri Kanoui).

Logic program:

```
/* Package 1:A and B non negative integers */ [Kanoui 1975]
rinf(A,B):- somreel(B,-A,C),signal(C,1).
rdiff(A,B):- equal(A,B),!,fail.
rdiff(A,B).

absdiff(X,Y,Z):- rinf(X,Y),!,somreel(Y,-X,Z).
absdiff(X,Y,Z):- somreel(X,-Y,Z).

somreel(A,B,C):- decomp(A,S1,A1,A2),
               decomp(B,S2,B1,B2),
               ppcm(A2,B2,P,A12,B12),
               N1 is A1*A12,N2 is B1*B12,
               som(S1,N1,S2,N2,S,N),
               pgcd(N,P,D,H,K),
               decomp(C,S,H,K).

som(S,A,S1,A1,1,0):- S\==S1,!.
som(S,A,S,B,S,C):- !, C is A+B.
som(S,A,S1,B,S,C):- B<A,! ,C is A-B.
som(S,A,S1,B,S1,C):- C is B-A.

subreel(-A,-B,C):- !,somreel(-A,B,C).
subreel(-A,B,C):- !,somreel(-A,-B,C).
subreel(A,-B,C):- !,somreel(A,B,C).
subreel(A,B,C):- !,somreel(A,-B,C).

prodreel(A,B,C):- decomp(A,S1,A1,A2),
                 decomp(B,S2,B1,B2),
                 sigprod(S,S2,S),
                 C1 is A1*B1,C2 is A2*B2,
                 pgcd(C1,C2,D,C11,C12),
                 decomp(C,S,C11,C12).
```

```

divreel(A,B,C):- decomp(A,S1,A1,A2),
               decomp(B,S2,B1,B2),
               sigprod(S1,S2,S),
               N1 is A1*B2, D1 is A2*B1,
               pgcd(N1,D1,G,N,D),
               decomp(C,S,N,D).

sigprod(S,S,1):- !.
sigprod(S,S1,-(1)).

pgcd(A,B,C,D,E):- B<A,! ,gcd(A,B,C),
                  D is A/C, E is B/C.
pgcd(A,B,C,D,E):- gcd(B,A,C),
                  D is A/C, E is B/C.

gcd(A,0,A):- !.
gcd(A,B,D):- C is A mod B,gcd(B,C,D).

ppcm(A,B,P,C,E):- pgcd(A,B,D,E,C),
                  Q is A*B, P is Q/D.

signal(-A,-(1)):- !,signal(A,1).
signal(-A,1):- !,fail.
signal(A,-(1)):- !,fail.
signal(0,0):- !.
signal(0,1):- !,fail.
signal(A,1).

decomp(-A,-S,A1,A2):- !,decomp(A,S,A1,A2).
decomp(A\B,1,A,B):- B\==1,! .
decomp(A,1,A,1).

equal(A,A):- !.
equal(A,B):- somreel(A,-B,C),
            somreel(C,-(1\1000),R1),
            somreel(C,1\1000,R2),
            signal(R1,-(1)),
            signal(R2,1).

/* Package 2 */ [L.Pereira 1975, personal communication]
:-op(700,xfx,[isr,norms_to,denorms_to]).
:-op(300,xfy,'^').

X isr Y :- rat(Y,X),! .

rat(U+V,W):- integer(U),integer(V),W is U+V;
rat(U,X),rat(V,Y),
X norms_to X1/X2,Y norms_to Y1/Y2,
lcm(X2,Y2,P,X12,Y12),
Z is X1*X12+Y1*Y12,
gcd(Z,P,_,W1,W2),
W1/W2 denorms_to W.

```

```

rat(U*V,W) :- integer(U), integer(V), W is U*V;
rat(U,X), rat(V,Y),
X norms_to X1/X2, Y norms_to Y1/Y2,
Z1 is X1*Y1, Z2 is X2*Y2,
gcd(Z1,Z2,_,W1,W2),
W1/W2 denorms_to W.
rat(-U,V) :- integer(U), V is -U ;
rat(U,V),
U norms_to U1/U2, V1 is -U1,
V1/U2 denorms_to V.
rat(U-V,W) :- rat(-V,V1), rat(U+V1,W).
rat(U/1,V) :- rat(U,V).
rat(U/V,W) :- rat(U,X), rat(V,Y),
X norms_to X1/X2, Y norms_to Y1/Y2,
sign(Y1,S,A),
Z1 is X1*Y2*S, Z2 is X2*A,
gcd(Z1,Z2,_,W1,W2),
W1/W2 denorms_to W.
rat(U^V,W) :- rat(V,Y), integer(Y),
rat(U,X), X norms_to X1/X2,
sign(Y,_,E), expt(X1,E,Z1), expt(X2,E,Z2),
(Y<0, sign(Z1,S,W1), W2 is S*Z2, W2/W1 denorms_to W;
Z1/Z2 denorms_to W).
rat(U,U) :- integer(U).

X norms_to X/1:- integer(X), !.
X/Y norms_to X/Y:- integer(X), integer(Y), Y>0.

X/1 denorms_to X:- !.
X denorms_to X.

number(X) :- X norms_to _ .

sign(X,S,A) :- X<0, !, S is -1, A is -X.
sign(X,1,X).

lcm(A,B,P,C,E) :- gcd(A,B,D,E,C), P is A*B/D.

gcd(A,B,D,E) :- sign(A,_,A1), sign(B,_,B1),
(B1<A1, !, gcd1(A1,B1,C);
gcd1(B1,A1,C)),
D is A/C, E is B/C.

gcd1(A,0,A) :- !.
gcd1(A,B,D) :- C is A mod B, gcd1(B,C,D).

expt(X,1,X) :- !.
expt(_,0,1) :- !.
expt(1,_,1).

expt(X,E,Y) :- even(E), E1 is E+1, expt(X,E1,Y1), Y is Y1*Y1;
E1 is E-1, expt(X,E1,Y1), Y is X*Y1.

```

Problem 68 [L. Pereira 1975, personal communication]

Verbal statement:

Consider Spencer Brown's 'The Laws of Form' (Bantam Books 1973) and write a program to implement his primary arithmetic (pg. 12-24).

Logic program:

```

:-op(500, xfy, ':').

read:- repeat, read(X),
       (X = end ; value(X,Y,Why),
        ttynl,ttynl,display('one has proved '),
        value_is(X,Y,Why),
        ttynl,ttynl,ttynl,fail).

value([],[],cancel):- !.
value([],[],condense):- !.

value(0:0,0,_):- !.
value(0:[],[],_):- !.
value([],0,[],_):- !.
value([],[],_):- !.
value([0],[],_):- !.
value(0,0,_):- !.

value(X:Y,Z,_):- value(X,XX,Why),
               value(Y,YY,Why2),
               value(XX:YY,Z,_),
               since,value_is(X,XX,Why),
               and,value_is(Y,YY,Why2),
               value_is(X:Y,Z,_).

value([X],Y,_):- value(X,Z,Why),value([Z],Y,_),
               since,value_is(X,Z,Why),
               value_is([X],Y,_).

since:- ttynl,ttynl,display('since ').

and:- ttynl,display('and ').

value_is(X,Y,Why):- ( Why=not_given;
                      Why=cancel,display('by cancellation');
                      Why=condense,display('by condensation') ),
                      !,ttynl,
                      show(X),display(' '),show(Y).

show(0):- !.
show(X:Y):- !,show(X),display(':'),show(Y).
show([X]):- !,display('['),show(X),display(']').
show(X):- display(X).

```

Problem 69 [L. Pereira 1975, personal communication]

Verbal statement:

Define the roman to arabic number conversion.

Logic program:

```
convert:- repeat, read(R), (R==stop; roman(R)), !, fail.  
roman(R):- roman1(D, R, []), !, nl,  
          write(D), nl, nl.  
  
/* DCG syntax */  
  
roman1(R) --> value(X), value(Y), {X<Y},  
                  roman1(R1), {R is Y-X+R1}.  
roman1(R) --> value(X), roman1(R1), {R is X+R1}.  
  
roman1(0)    --> [].  
  
value(1)     --> "I".  
value(5)     --> "V".  
value(10)    --> "X".  
value(50)    --> "L".  
value(100)   --> "C".  
value(500)   --> "D".  
value(1000)  --> "M".
```

Comments:

This program adopts the DCG grammar rule notation for ease of clarity.

Chapter 6 Doing Algebra with Prolog

As Richard Hamming stated, “the purpose of computing is insight, not numbers”. Insight is obtained by evaluating a mathematical expression. As a matter of fact, relations on quantities can be made clearer by algebraic means, and abstract symbols as well as numbers can be manipulated by a computer. One reason the algebraic capabilities of the computer have not been fully exploited is the lack of simple programming languages. This chapter gives evidence of the use of Prolog for algebraic processing, enhancing how a library of mathematical knowledge can be declared. These simple problems contribute to a better understanding of how algebraic processing can reduce numerical-processing time and how mathematical methods, such as those for solving differential equations, can be included in knowledge-based programs.

Problem 70 [L. Pereira 1976, personal communication]

Verbal statement:

Specify the differentiation function.

Logic program:

```
:op(700,xfx,:).
:op(500,xfy,[+,-]).
:op(400,xfy,[*,/]).
:op(300,xfy,@).
:op(300,fy,[-,exp,log,sin,cos,tg]).  
dv:-repeat,read(T),( T=stop ; derive(T),fail).  
derive(T:X):- T : X, !.  
X : [V] :- X : V.  
X : [U|V] :- d(X,U,W), simplify(W,Z), Z : V.  
X : Y :-d(X,Y,W), simplify(W,Z),  
       ttynl,output(Z),ttynl,ttynl.  
d(X,X,1).  
d(T,X,0):- atom(T),T=/=X ; number (T).
```

```

d(U+V,X,A+B):- d(U,X,A), d(V,X,B).

d(U-V,X,A+(-B)):- d(U,X,A), d(V,X,B).

d(-T,X,-R):- d(T,X,R).

d(K*U,X,K*W):- number (K), d(U,X,W).

d(U*V,X,B*U+A*V):- d(U,X,A), d(V,X,B).

d(U/V,X,W):- d(U*V@(-1),X,W).

d(U@V,X,V*W*U@(V+(-1))):- number V, d(U,X,W).

d(U@V,X,Z*log(U)*U@V+V*W*U@(V+(-1))):- d(U,X,W), d(V,X,Z).

d(log (T),X,R*T@(-1)):- d(T,X,R).

d(exp T,X,R*exp T):- d(T,X,R).

d(sin T,X,R*cos T):- d(T,X,R).

d(cos T,X,-R*sin T):- d(T,X,R).

d(tg T,X,W):- d(sin T/cos T,X,W).

simplify(X,X):- atomic(X).

simplify(X,Y):- X =.. [Op,Z], simplify(Z,Z1), rewrite(Op,Z1,Y).

simplify(X,Y):- X =.. [Op,Z,W], simplify(Z,Z1),
               simplify(W,W1),
               rewrite(Op,Z1,W1,Y).

rewrite(exp,K*log X,W):- binary(@,X,K,W).

rewrite(-,A+B,C+D):- unary(-,A,C), unary(-,B,D).

rewrite(X,Y,Z):- unary(X,Y,Z).

rewrite(*,X,X,W):-binary(@,X,2,W).

rewrite(*,-X,X,W):- binary(@,X,2,Z), unary(-,Z,W).

rewrite(*,X,-X,W):-binary(@,X,2,Z), unary(-,Z,W).

rewrite(*,-Y,Z):- binary(*,X,Y,Z).

rewrite(*,X,-Y,W):- binary(*,X,Y,Z), unary(-,Z,W).

rewrite(*,-X,Y,W):- binary(*,X,Y,Z), unary(-,Z,W).

rewrite(*,A@X,A@Y,W):- binary(+,X,Y,Z), binary(@,A,Z,W).

rewrite(*,X@A,Y@A,W):- binary(*,X,Y,Z), binary(@,Z,A,W).

rewrite(@,X*Y@(-1),-Z,W):- binary(@,Y*X@(-1),Z,W).

rewrite(X,Y,Z,W):- binary(X,Y,Z,W).

unary(-,-X,X).

unary(-,0,0).

unary(-,X,Y):- Y isr -X.

unary(log,1,0).                                unary(log,exp X,X).
unary(exp,0,1).                                unary(exp,log X,X).
unary(sin,0,0).                                 unary(sin,-X,- sin X).
unary(cos,0,1).                                 unary(cos,-X,cos X).

unary(Op,X,Y):- Y =.. [Op,X].

```

```

binary(+,X,0,X).
binary(+,X,-X,0).
binary(+,A,B+C,R+C):- R isr A+B.
binary(+,A+B,C,R+B):- R isr A+C.
binary(+,(sin X)@2,(cos X)@2,1).
binary(+,(cos X)@2,(sin X)@2,1).
binary(+,X,Y,Z):- Z isr X+Y.
binary(+,A,B,B+A):- number B.

binary(*,X,1,X).
binary(*,0,_,0).
binary(*,-1,X,-X).
binary(*,X,X@(-1),1).
binary(*,A,B*C,R*C):- R isr A*B.
binary(*,A*B,C,R*B):- R isr A*C.
binary(*,X,Y,Z):- Z isr X*Y.
binary(*,A,B,B*A):- number B.

binary(@,1,_,1).
binary(@,X,1,X).
binary(@,_,0,1).

binary(Op,X,Y,Z):- Z =.. [Op,X,Y].
output(X):- clean(X,Y), write(Y).

clean(X,X):- atom(X) ; number X.
clean(X,Y):- X =.. [Op,Z], clean(Z,Z1), Y =.. [Op,Z1].
clean(X,Y):- X =.. [Op,Z,W], clean(Z,Z1), clean(W,W1),
            U =.. [Op,Z1,W1], change(U,Y).

change(-1+X,X-1).
change(X+(-Y),X-Y).
change(X*Y@(-1),X/Y).
change(X@(-1)*Y,Y/X).
change(X,X).

```

Execution:

```

?- d((y+a)*(y-a),y,R).
R=(1+0)*(y-a)+(y+a)*(1-0)

```

Comments:

The result 'R' is $2*y$.

Problem 71 [Emden 1976 c]

Verbal statement:

Define the exponentiation function:

$$\text{exp}(x, y, z) \quad \text{iff} \quad x^y = z$$

The following properties of integer exponentiation are taken as the defining ones:

$$x^0 = 1 \quad \text{i.e.} \quad \forall x \quad \text{exp}(x, 0, 1)$$

and

$$x^y = x * x^{y-1} \quad \text{i.e.} \quad \forall x, y, z \quad \text{exp}(x, y-1, z) \Rightarrow \text{exp}(x, y, x * z)$$

It is taken for granted that the following defining property can be added without altering the relation defined and yet making the definition more “efficient”:

$$x^{2y} = (x^2)^y \quad \text{i.e.} \quad x^y = (x^2)^{y/2} \quad \text{i.e.} \\ \forall x, y, z \quad (\text{exp}(x * x, y/2, z) \wedge \text{even}(y)) \Rightarrow \text{exp}(x, y, z)$$

Logic program:

```
exp(X, 0, 1).
exp(X, Y, Z) :- even(Y), R is Y/2, P is X*X,
               exp(P, R, Z).
exp(X, Y, Z) :- T is Y-1, exp(X, T, Z1), Z is Z1*X.
even(Y) :- R is Y mod 2, R=0.
```

Execution:

?- `exp(2, 10, Z).`

`Z=1024`

Chapter 7 Playing with Prolog

Computer games consist of logic, strategy and rules. And, of course, there is a need for graphics to capture the imagination. All these crucial elements are available with the Prolog systems or can be easily written.

Games are difficult to understand. People know what they like but cannot explain them. This chapter emphasizes good programming techniques, including the so-called “structured programming”. Some ready-to-go examples cover games of logic, board games and puzzles. This topic proves to be an interesting and entertaining example of symbol manipulation in Prolog.

Problem 72 [Sebelik 1983]

Verbal statement:

There are five hats. Three of them are white and two are black. Three men are standing one behind the other, i.e. the second sees the first, the third sees the first, and the third sees the second. Each has one hat on his head. Only two hats are black, so it is impossible that each man has a black hat. The third says that he does not know what hat he has on his head. Then the second says the same. So neither the third, nor the second knows what hat he has himself. The question is what hat the first man has.

Write a program that solves the above logic puzzle.

Logic program:

```
: - op(100,xf , 'holds').  
:- op(150,xfy, '.' ).  
:- op(200,xfx,'if').  
:- op(220,xfy,'and_').  
:- op(300,xfx,'can_verify').  
:- op(300,xfx,'sees').  
:- op(300,xfx,'is_deducible_from').  
:- op(300, fx,'it_is_deducible_that').  
:- op(300,xf , 'is_false').  
:- op(320,xfy,'and').  
:- op(340,xfx,'knows_that').  
:- op(340,xf , 'knows_what_hat_he_has_himself').  
:- op(360,xfx,'has').  
:- op(380, fx,'the').  
:- op(380,xf , 'hat').
```

```

X holds:- fact(X).
X holds:- fact(X if Y), Y holds.
X and_ Y holds:- X holds, Y holds.

fact(X):- ( the_problem_is_stated_as(Text)      ;
            inference_rules_are_stated_as(Text) ),
            member(X,Text)

question(X):- it_is_deducible_that X holds, output(X).

the_problem_is_stated_as(
    there_are_five_hats. three_of_them_are_white_two_are_black.
    three_men_are_standing_one_behind_the_other_i.e. the second sees
    the first. the third sees the second. the third sees the first.
    each_has_one_hat_on_his_head. ie_the_statement. the third has
    black hat and the second has black hat and the first has black
    hat is_false.

    the_third_said_that_he_does_not_know_what_hat _he_has_on_his_h-
    ead.

    then_the_second_said_the_same. it_means_that. the third
    knows_what_hat_he_has_himself is_false. and_similarly. the second
    knows_what_hat_he_has_himself is_false.

    the_question_is_what_hat _the_first_man_has).

inference_rules_are_stated_as(
    it_is_deducible_that A_man has white hat if Statement is_false
    and_ Statement is_deducible_from A_man has black hat. similarly.
    A_man has white hat is_deducible_from An_assumption if Statement
    is_false and_Statement is_deducible_from A_man has black hat and
    An_assumption.

    A_man knows_what_hat_he_has_himself is_deducible_from An_assump-
    tion if A_man can_verify An_assumption and_
    A_man has

    Certain hat is_deducible_from An_assumption.

    Man1 can_verify Man2 has Some hat if Man1 sees Man2. A_man can-
    _verify Fact1 and Fact2 if A_man can_verify Fact1 and_ A_man
    can_verify Fact2.

    Everything is_deducible_from An_assumption if An_assumption
    is_false.

    and_it_is_all_for_solving_the_problem).

```

Execution:

```
?- question( the first has What hat ).
```

the program will answer:

```
the first has white hat.
```

Comments:

The above Prolog program solves a logic puzzle. The program consists of three parts. In the first part, the syntax of a problem oriented language is defined by a suitable choice of operators and their precedences. The second part is a simple interpreter for this problem oriented language. The third part is the main program, which solves the problem. It is written in the problem oriented language and, in fact, it is a formulation of the problem at the same time.

Problem 73

Verbal statement:

There are three pegs, A,B, and C, and three disks of different sizes a,b, and c. The disks have holes in their centers so that they can be stacked on the pegs.

Initially all the disks are on peg A; the largest, disk c, is on the bottom, and the smallest, disk a, is on the top.

It is desired to transfer all of the disks to peg C by moving disks one at a time. Only the top disk on a peg can be moved, but it can never be placed on top of a smaller disk (Towers-of-Hanoi puzzle).

Logic program 1 [L. Pereira 1975, personal communication]:

```
hanoi(1,A,B,C):- !, write('Move disk '), write(A),
                  write(' to '), write(B), nl.
hanoi(N,A,B,C):- M is N-1,
                  hanoi(M,A,C,B), hanoi(1,A,B,C),
                  hanoi(M,C,B,A).
```

Logic program 2 [Roussel 1975]:

```
hanoi(N):- hanoi(N,L,[]), nl, write(L), nl.
hanoi(N) --> put(N,a,b,c).
put(0,_,_,_) --> [].
put(N,A,B,C) --> {M is N-1},
                     put(M,A,C,B), move(A,B), put(M,C,B,A).

move(X,Y) --> [from,X,to,Y].
```

Execution of the first logic program:

```
?- hanoi(3,a,c,b).
```

Move disk	a	to	c
Move disk	a	to	b
Move disk	c	to	b
Move disk	a	to	c
Move disk	b	to	a
Move disk	b	to	c
Move disk	a	to	c

Comments:

Please note that the letters represent the pegs between which the movements are made.

Problem 74 [Emden 1978]

Verbal statement:

Consider the abstract game of Mastermind with 6 colors and 4 positions in which a combination can be placed. Consider that, in the score, a black key peg means strong match (right color in the right place) and a white key peg means weak match (right color in the wrong place). Write a program able to perform the Codemaker when you play the Codebreaker and vice-versa.

Suggestion: Consider that the Codebreaker constructs a sequence p_1, \dots, p_n of probes with $p_i = / = C$ for $i = / = n$ and $p_n = C$, where C means the key code. The Codemaker constructs a sequence s_1, \dots, s_n such that $s_i = f(p_i, C)$ where f is the score function. The selection of p_i by the Codebreaker may depend on $(p_i, s_i), \dots, (p_{i-1}, s_{i-1})$. It is the Codebreaker's objective to make n as small as possible.

Logic program

```
/* Program for the Mastermind game */
:-op(300, xfy, '.').
:-op(150, fyx, 's').

/* Interactive manager */

play:- write('Mastermind at your service.'), nl,
      write('Enter an arbitrary number between 0 and 164.'), nl,
      read(X), assert(seed(X)),
      write('Example format for entering code : '),
      write('[yellow,blue,white,black]'), nl,
      write('Example format for the score : '),
      write('[s s s 0,s 0]'), nl,
      write('Meaning 3 blacks and 1 white.'), nl,
      repeat, play1.

play1:- write('Do you want to make or to break codes ?'), nl,
        write('Answer make or answer break.'), nl,
        read(X), start(X).

ask:- write('Do you want another game ?'), nl,
      write('Answer yes or answer no.'), nl,
      read(X), answer(X).

answer(yes):- !, fail.
answer(no):- retract(seed(_)), nl,
            write('Mastermind was pleased to serve you.'), nl, nl.
```

```

/* User performs the Codemaker */

start(make):- write('Enter code; I promise not to look.'), nl,
            read(C), assert(code(C)),
            generate_code(P), score(P,S),
            write('My first probe is : '), write(P), nl,
            write('Score : '), write(S), nl,
            extendable((P.S).[],S S S S 0),
            retract(code(_)), !, ask.

extendable((P.(S S S _._))._,_):- write('The code must be: '),
                                         write(P), nl.

extendable(Cs,Ns):- can_code(Cs,Cc),
                  write('My next probe is: '), write(Cc), nl,
                  write('Score : '),
                  score(Cc,S), write(S), nl,
                  extendable((Cc.S).Cs,N).

can_code([],_).
can_code((P1.S1).Ps,P):- mm(P1,P,S1), can_code(Ps,P).

/* User performs the Codebreaker */

start(break):- generate_code(C), assert(code(C)),
              write('Enter first probe.'), nl,
              read(P), score(P,S), continue_break(S),
              !, ask.

continue_break(S S S _._):- write('You got it.'), nl,
                           retract(code(_)).

continue_break(S):- write('Your score : '), write(S), nl,
                   write('Enter next probe or type stop.'), nl,
                   read(X), respond_to(X).

respond_to(stop):- !, write('I assume you give up;'),
                  write(' the code is: '),
                  retract(code(C)), write(C), nl.
respond_to(P):- score(P,S), continue_break(S).

/* Score function */

score(Probe,Score):- code(C), mm(Probe,C,Score).

mm(P,C,S1,S2.[]):- blacks(P,C,P1,C1,S1),
                  whites(P1,C1,S2,C1).

blacks([],[],[],[],0).
blacks(U.P,U.C,P1,C1,S):- blacks(P,C,P1,C1,S).
blacks(U.P,V.C,U.P1,V.C1,S):- diff(U,V),
                               blacks(P,C,P1,C1,S).

```

```

whites([],C,0,Ms):- tuple(C,Ms).
whites(U,P,C,s,S,Ms1,Ms):- del(U,C,C1,Ms),
                           whites(P,C1,S,Ms).
whites(U,P,C,S,Ms):- nonmem(U,C,Ms), whites(P,C,S,Ms).

/* Code generator */

generate_code(U,V,W,X,[]):- random_colour(U),
                           random_colour(V),
                           random_colour(W),
                           random_colour(X).

random_colour(X):- random_number(R), N is R mod 6,
                  N1 is N+1, colour(N1,X).

random_number(R1):- retract(seed(R)), X is R*125,
                  Y is X+1, R1 is Y mod 165,
                  assert(seed(R1)).

colour(1,black).           colour(2,blue).
colour(3,green).          colour(4,red).
colour(5,white).          colour(6,yellow).

/* Miscellaneous */

/* del(U,Y,Y1) is the result of deleting U from list Y */

del(U,U,Y,Y,Ms):- tuple(Y,Ms).
del(U,V,Y,V,Y1,Ms1,Ms):- diff(U,V), del(U,Y,Y1,Ms).

/* nonmem(U,V) if U is not a member of list V */

nonmem(_,[],[]).
nonmem(U,V1,V,W1,W):- diff(U,V1), nonmem(U,V,W).

tuple([],[]).
tuple(U1,U,V1,V):- colour(_,U1), tuple(U,V).

diff(C1,C2):- colour(_,C1), colour(_,C2), not(C1=C2).

```

Execution:

```

| ?-play.

Mastermind at your service.
Enter an arbitrary number between 0 and 164.
| 123.
Example format for entering code : [yellow,blue,white,black]
Example format for the score : [s s s 0,s 0]
Meaning 3 blacks and 1 white.
Do you want to make or to break codes ?
Answer make or answer break.
| make.

```

```
Enter code; I promise not to look.  
| [blue,white,blue,white].  
My first probe is : [blue,red,blue,black]  
Score : [s s 0,0]  
My next probe is : [blue,red,green,green]  
Score : [s 0,0]  
My next probe is : [blue,black,black,black]  
Score : [s 0,0]  
My next probe is : [blue,blue,blue,blue]  
Score : [s s 0,0]  
My next probe is : [blue,white,blue,white]  
Score : [s s s s 0,0]  
The code must be : [blue,white,blue,white]  
Do you want another game ?  
Answer yes or answer no.  
| yes.  
Do you want to make or to break codes ?  
Answer make or answer break.  
| break.  
Enter first probe.  
| [blue,blue,red,red].  
Your score : [0,s 0]  
Enter next probe or type stop.  
| [white,white,white,white].  
Your score : [s s 0,0]  
Enter next probe or type stop.  
| [white,white,blue,black].  
Your score : [s s 0,s 0]  
Enter next probe or type stop.  
| [white,white,green,blue].  
Your score : [s 0,s 0]  
Enter next probe or type stop.  
| [white,red,white,black].  
You got it.  
Do you want another game ?  
Answer yes or answer no.  
| no.  
Mastermind was pleased to serve you.
```

Problem 75 [Bruynooghe 1978]

Verbal statement:

Write a program to solve the N-queens problem. The goal consists in finding all the ways of placing N queens on an $N \times N$ chess board so that no queen attacks another, where two queens are said to attack each other if they are positioned along a common row, column or diagonal.

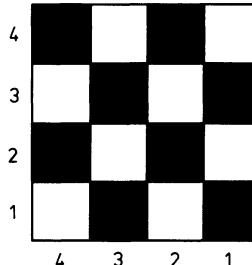


Fig. 1

Logic program 1:

```

begin:- permutation([4,3,2,1],L),pair([4,3,2,1],L,Q),
        safe(Q),write(Q),nl,nl.

/* Generator of permutations */

permutation([],[]).
permutation([X|Y],[U|V]) :- delete(U,[X|Y],W),permutation(W,V).

delete(X,[X|Y],Y).
delete(U,[X|Y],[X|V]) :- delete(U,Y,V).

/* Pairing lines with columns */

pair([],[],[]).
pair([X|Y],[U|V],[spot(X,U)|W]) :- pair(Y,V,W).

/* Check if the queen positions are compatible */

safe([]).
safe([P|Q]) :- check(P,Q),safe(Q).

check(P,[]).
check(P,[Q|R]) :- diagonals(P,Q),check(P,R).

/* non_attacking condition on diagonals */

diagonals(spot(X1,Y1),spot(X2,Y2)) :- Dx is X1-X2,
                                         Dy is Y1-Y2,
                                         Dx =\= Dy,
                                         Dx =\= -Dy.
```

Logic program 2: (more efficient version) [L.Pereira 1978]

```

queens(L,[C1|RC],R) :- delete(L1,L,RL),safe(p(L1,C1),R),
                     queens(RL,RC,[p(L1,C1)|R]).  

queens([],[],R).  

/* Delete and save as in logic program 1 */
```

Execution:

For the case of four queens, we have a 4×4 chess board:

```
?- queens([1,2,3,4],[1,2,3,4],[]).
```

Comments:

The N-queens problem is one of more commonly discussed puzzles in programming texts. In 1972, Dijkstra published its solution for conventional programming languages.

This problem is a good pedagogic example to open the debate about the selection of the most appropriate data structure to represent the solutions. The obvious solution will be a list, where the first element gives the column number of the queen in the first row, the second element gives the column number for the second row and so on. However, this data structure does not simplify the task of checking the validity of a particular configuration of queens. In stead of a list of column number, the logic program presents a list of terms ‘spot(NL,NC)’, where ‘NL’ characterizes the number of the line and ‘NC’ the number of the column. Each term ‘spot’ represents a square of the chess board.

See, also, problem 180.

Problem 76 [Emden 1980]

Verbal statement:

Write a program for the game of Nim defined as follows.

A position of the game of Nim can be visualized as a set of heaps of matches. Two players, called US and THEM, alternate making a move. As soon as a player, whose turn it is to move, is unable to make a move, the game is over and that player has lost; the other player is said to have won. A move consists of taking at least one match from exactly one heap.

Logic program:

```

:- op(300,xf,+).

s(X,Y):- move(X,Y),not(t(Y,Z)).
t(X,Y):- move(X,Y),not(s(Y,Z)).

append([],Y,Y).
append([U|X],Y,[U|Z]) :- append(X,Y,Z).

move(X,Y) :- append(U,[X1|V],X),
            takesome(X1,X2),
            append(U,[X2|V],Y).

takesome(X+,X).
takesome(X+,Y) :- takesome(X,Y).

```

Problem 77 [Abreu 1980, personal communication]

Verbal statement:

Write a program to play solitaire (one-player game) with pieces that behave like checkers (a piece can take adjacent ones if and only if there is an empty hole next to the other piece). The game starts with all but one hole filled, and the objective is to end with only one piece on the board. In the following figure a graph describes the board.

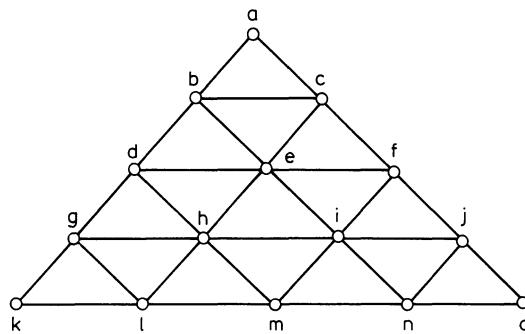


Fig.2

Logic program 1:

```
/* Compiler directive for C-Prolog */
:- op(300,xfy,:).
:- op(300,xfx,'in').

solitaire(Ea):- all(Nodes),remove(Ea,Nodes,Full),
              findlist(Full,[Ea],Moves),print(Moves),nl.
solitaire(_):- nl,write('Sorry but I cannot solve this.'),nl.

findlist([X],_,[]).
findlist(Full,Empty,[X|Moves]):- select(X,Full,NFull,Empty,
                                         NEmpty),
                                findlist(NFull,NEmpty,Moves).

select(A:B:C,F,NF,E,NE):- A:B:C,
                           remove(A,F,NF1), remove(B,NF1,NF2),
                           remove(C,E,NE1), add(C,NF2,NF),
                           add(A,NE1,NE2), add(B,NE2,NE).

/* Utilities */
H in [H|L].
H in [_|L] :- H in L.

add(X,[],[X]).
add(X,[L|U],[L|V]) :- add(X,U,V).
```

```

remove(X,[X|Y],Y):- !.
remove(X,[Y|U],[Y|V]):- remove(X,U,V).

print([A:B:C]):- write('From '),write(A),write(' to '),write(C),
        write(' eating '),write(B),nl.
print([X|Xs]):- print([X]),print(Xs).

a:c:f.    a:b:d.    b:d:g.    b:e:i.    c:e:h.    c:f:j.    d:g:k.
d:h:m.    d:e:f.    d:b:a.    e:h:l.    e:i:n.    f:c:a.    f:e:d.
f:i:m.    f:j:o.    g:d:b.    g:h:i.    h:e:c.    h:i:j.    i:e:b.
i:h:g.    j:f:c.    j:i:h.    k:g:d.    k:l:m.    l:h:e.    l:m:n.
m:n:o.    m:i:f.    m:h:d.    m:l:k.    n:i:e.    n:m:l.    o:j:f.
o:n:m.

all([a,b,c,d,e,f,g,h,i,j,k,l,m,n,o]).
```

Execution:

```
?- solitaire(f).
From a to f eating c
From d to a eating b
From f to d eating e
From g to b eating d
From a to d eating b
From l to e eating h
From d to f eating e
From j to c eating f
From n to l eating m
From k to m eating l
From m to f eating i
From c to j eating f
From o to f eating j
```

Comments:

The argument of ‘solitaire’ designates the initially empty place (the hole).

Logic program 2:

```

:- public solitaire/1. /* Compiler directives for C-Prolog */
:- mode remove(-,-,+), add(-,-,+), select(+,-,+,-,+),
   findlist(-,-,-,+), alter_lists(-,-,+,-,+), show(-).

:- op(300,xfy,:).

solitaire(Ea):- all(N,Nodes), M is N-1, remove(Ea,Nodes,Full), !,
              findlist(M,Full,[Ea],Moves), show(Moves),nl.
solitaire(_):- nl,write('Sorry but I cannot solve this.'),nl.
```

```

findlist(_, [X], _, []).
findlist(N, Full, Empty, [X|Moves]):-
    \+ deadend(N, Full), !,
    check(N, Full),
    select(X, Full, NFull, Empty,
           NEmpty),
    M is N-1,
    findlist(M, NFull, NEmpty, Moves).

select(A:B:C, F, NF, E, NE):-
    (A:B:C ; C:B:A),
    alter_lists(A:B:C, F, NF1, E, NE1),
    add(C, NF1, NF),
    add(A, NE1, NE2), add(B, NE2, NE).

alter_lists(A:B:C, F, NF, E, NE):-
    remove(A, F, NF1), !,
    remove(B, NF1, NF), !,
    remove(C, E, NE).

check(_,_).
check(N, Full):-
    assert(deadend(N, Full)), !, fail.

add(X, [], [X]):- !.
add(X, [Y|L], [X, Y|L]):- X @=< Y, !.
add(X, [L|U], [L|V]):- add(X, U, V).

remove(X, [X|Y], Y).
remove(X, [Y|U], [Y|V]):- remove(X, U, V).

show([A:B:C]):-
    write('From '), write(A), write(' to '), write(C),
    write(' eating '), write(B), write('.'), nl.
show([X|Xs]):-
    show([X]), show(Xs).

all(15, [a,b,c,d,e,f,g,h,i,j,k,l,m,n,o]).  

a:c:f.    c:f:j.    f:j:o.    o:n:m.    n:m:l.    m:l:k.  

k:g:d.    g:d:b.    d:b:a.    b:e:i.    c:e:h.    d:e:f.  

d:h:m.    e:i:n.    g:h:i.    h:i:j.    e:h:l.    f:i:m.

```

Comments:

The second program is more efficient and clever than the first one! In a second game it will not repeat moves that have already failed at some earlier time.

Problem 78

Verbal statement:

There are two numbers, M and N, such that $1 < M < N < 100$. Mister S is told the sum of the two numbers, Mister P is told their product, and they both know they were told so (Bramer 1980). The following dialogue takes place:

Mr. P: I don't know the numbers.
 Mr. S: I knew you didn't know them;
 I don't know them either.
 Mr. P: Now I know the numbers!
 Mr. S: Now I know them too!

What are the numbers M and N?

(This dialogue game was posed by John McCarthy during the Expert Systems Summer School in Edinburgh, 1979).

Logic program 1 [Porto 1981]:

```
/* Definition of infix operators */
:- op(800,xfy,[one_and_only_one, allows, verifying, every]).
:- op(800,xfy,[implies, more_than_one, have]).
:- op(750,xf,:).
:- op(700,xfx,given).
/* The problem */

sp(M,N) :- the_sum_is(S),
           sentence4 :
           one_and_only_one product(P) given sum(S)
           allows sentence3 verifying Sx=S :
           one_and_only_one sum(Sx) given product(P)
           allows sentence2 :
           every product(Px) given sum(Sx)
           implies sentence1 :
           more_than_one sum(_) given product(Px),
           the_numbers(M,N) have sum_and_product(S,P).

/*
The following subproblems make use of the DEC-10 Prolog      */
/* recording mechanism which works as follows:             */
/*   'recorda(K,T,R)' records term T under key K,        */
/*   using reference R.                                     */
/*   'recorded(K,T,R)' accesses term T recorded under key K, */
/*   its reference being R.                                */
/*   'erase(R)' erases the term recorded with reference R. */
/* */

S : one_and_only_one X allows Y :- call((X,Y)),( recorded(S,_,R),
                                             erase(R), !,fail ;
                                             recorda(S,X allows Y,_),
                                             fail );
                                             recorded(S,X allows Y,R) erase(R).

S verifying P : one_and_only_one X allows Y :-
               call((X,Y)),( not(P),
                             no_record(S), !,fail ;
                             recorded(S,_,R),
                             erase(R), !,fail ;
```

```

recorda(S,X allows Y,_),
          fail ) ;
recorded(S,X allows Y,R), erase(R).

S : more_than_one X :- call(X),( recorded(S,_,R), erase(R), ! ;
                                recorda(S,1,_), fail ) ;
                                recorded(S,_,R), erase(R), fail.

S : every X implies Y :- call(X),( not(Y),
                                no_record(S), !,fail ;
                                update_record(S), fail ) ;
                                recorded(S,N,R), erase(R), N=2.

no_record(S) :- recorded(S,_,R), erase(R) ;
              true.

update_record(S) :- ( recorded(S,2,_) ;
                        recorded(S,1,R), erase(R), recorda(S,2,_) ;
                        recorda(S,1,_), !.

not(X) :- call(X), !,fail ;
          true.

the_sum_is(S) :- integer(S,4,198).
the_numbers(M,N) have sum_and_product(S,P) :- integer(M,2,99),
                                             N is S-M,
                                             P is M*N, !.

product(P) given sum(S) :- S2 is S/2, integer(M,2,S2), P is M*(S-M).
sum(S) given product(P) :- factor(P,2,M), S is M+(P/M).
integer(I,I,_).

integer(I,Low,Up) :- New_low is Low+1, New_low =< Up,
                   integer(I,New_low,Up).

factor(P,M,M) :- 0 is P mod M.

factor(P,Guess,M) :- New_guess is Guess+1, P >= New_guess*New_guess,
                   factor(P,New_guess,M).

```

Comments:

By reading the statement of the problem it seems there is insufficient information to solve it, but in fact the solution requires no more than a limited knowledge of number theory. The analysis of the problem was done for the first time by (Bramer 1980) and followed by an answer (Nilsson; Campbell 1980) with a program written in Prolog.

This solution runs in the same way as the solution proposed by David Warren in the following program. It takes 4 seconds (Logic program 2 takes approximately 9 seconds), basically because this implementation is more efficient by not using 'setof'. The predicate 'one_and_only_one' fails as soon as a second solution is found, and 'every' also checks that there are several of them.

Logic program 2 [Warren 1981]

```
% Mr S and Mr P Problem.
%
% There are two numbers M and N such that 1 < M & N < 100.
% Mr S is told their sum S and Mr P is told their product P. The
% following dialogue takes place:
%
% Statement-1:
% Mr P: I don't know the numbers.
%         (There are several sum values S that are compatible with
%         the product value P).

statement1(P) :- several(S, compatible(S,P)).
```

```
% Statement-2:
% Mr S: I knew you didn't know them;
%         I don't know them either.
%         (For every product value P that is compatible with the
%         sum value S, statement-1 is true of P; and
%         there are several product values P that are compatible
%         with the sum value S).

statement2(S) :- every(P, compatible(S,P), statement1(P)),
               several(P, compatible(S,P)).
```

```
% Statement-3:
% Mr P: Now I know the numbers!
%         (There is just one sum value S compatible with the product
%         value P for which statement-2 is true of S, and that value
%         is S1).

statement3(P,S1) :- one(S,
                        (sumvalue(S), statement2(S),
                         compatible(S,P)),
                        S1).
```

```
% Statement-4:
% Mr S: Now I know them too!
%         (There is just one product value P compatible with the
%         sum value S for which statement-3 is true of P and S,
%         and that value is P1).

statement4(S,P1) :- one(P, (statement3(P,S), compatible(S,P)), P1).
```

```
% Question: What are the numbers?
%           (For which sum value S and product value P is state-
%           ment-4 true?)
```

```

answer(S,P) :- statement4(S,P).
% [The single solution S = 17, P = 52 is produced in about 9 seconds].
% Definitions of the quantifiers 'one', 'several' and 'every'.
one (X,P,X1) :- setof(X,P,[X1]).
several(X,P) :- setof(X,P,Xs), length(Xs,N), N > 1.
every(X,P,Q) :-\+ (P,\+Q).
% The remaining definitions are compiled:
:-compile(sandp1).
%
% Supporting definitions for the Mr S and Mr P Problem.
% Sum values range from 4 to 198.
sumvalue(S) :- range(S,4,198).

% The next two clauses are logically equivalent to the third clause,
% but are more efficient in the cases that S or P are already known.

compatible(S,P) :- nonvar(S), !,
    Mmax is S/2, S99 is S-99, max(2,S99,Mmin),
    range(M,Mmin,Mmax),
    P is M*(S-M).

compatible(S,P) :- nonvar(P), !,
    sqroot(P,Mmax), P99 is P/99, max(2,P99,Mmin),
    range(M,Mmin,Mmax),
    N is P/M, P is M*N,
    S is M+N.

% Sum value S is compatible with product value P if there are
% numbers M and N in the range 2 to 99 such that S is the sum
% of M and N, and P is the product of M and N. (See above.)

compatible(S,P) :-
    range(M,2,99),
    range(N,2,99),
    S is M+N,
    P is M*N.

% Finally, definitions of the predicates 'range', 'max' and 'sqroot'.
range(I,L,M) :- nonvar(I), !, L =< I, I =< M.
range(I,I,_).
range(I,L,M) :- L < M, L1 is L+1, range(I,L1,M).

max(X,Y,X) :- X >= Y, !.
max(X,Y,Y) :- X < Y, !.

sqroot(N,RN) :- N < 181, !, N1 is N*4, N2 is N*2,
    sqroot(N1,RN1,0,N2), RN is RN1/2.

```

```

sqroot(N,RN) :- N < 32768, !, N1 is N*4,
              sqroot(N1,RN1,0,363), RN is RN1/2.
sqroot(N,RN) :- sqroot(N,RN,0,363).

% 'sqroot' expanded to include upper and lower limits
sqroot(N,RN,RN,_) :- N =:= RN*RN, !.
sqroot(N,RN,RN,RN1) :- RN1-RN < 2, !.
sqroot(N,RN,LL,UL) :- ML is (LL+UL+1)/2, M is ML*ML,
                     sqrootn(N,RN,LL,UL,ML,M).

% 'sqrootn' sets up for next invocation of sqroot
sqrootn(N,RN,LL,UL,ML,M) :- M > N, !, sqroot(N,RN,LL,ML).
sqrootn(N,RN,LL,UL,ML,M) :- M =< N, sqroot(N,RN,ML,UL).

```

Comments:

This program modifies the solution to the “S–P problem” by António Porto (Logic program 1). The first four statements correspond exactly to the four statements of the dialogue. The built-in predicate ‘retract’ is dispensed with in favor of ‘setof’, which is really what is needed. The quantifiers “one”, “several”, “every” are very close to the treatment of determiners in the natural language question-answering system Chat, implemented by Fernando Pereira and David Warren. The aim of David Warren was to produce a clearer version than the one by António Porto and solve the “S–P Problem” in very much the same way as Max Bramer outlined in AISB Quarterly No. 37.

Problem 79

Verbal statement:

A puzzle that is frequently adopted to illustrate problem solving concepts is the 15-puzzle. The 15-puzzle consists of 15 numbered, movable tiles set in a 4×4 frame. One cell of the frame is always empty, making it possible to move an adjacent numbered tile into the empty cell, thus moving the empty cell also.

Consider a simpler configuration, the 8-puzzle, and write a Prolog program for playing it.

Logic program:

```

/* 8-PUZZLE */

/* Main module of the program: */
/*      construction of a state list */
/*      defining the path from */
/*      initial state X to final V */

```



```

change([A,B,X3],[A,B,Y3],N):- member(b,X3),
                                ((N=:=1, change1(X3,Y3));
                                 (N=:=2, change2(X3,Y3))).

change1([H1,H2|L],[H2,H1|L]). 
change2([A|B],L):- change1(B,B1), conc([A],B1,L).

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

not(P):- P,! ,fail.
not(_).

right(X,Y):- (member([_,b,_],X),change(X,Y,2));
              (member([b,_,_],X),change(X,Y,1)).

up(X,Y):- trans(X,Z),left(Z,W),trans(W,Y).

down(X,Y):- trans(X,Z),right(Z,W),trans(W,Y).

trans([I],[],[]).
trans([N1|L1],[H2|L2],[H3|L3]),[[H1,H2,H3]]:- trans([L1,L2,L3],L).

/* Computation of value U of actual state Z:      */
/*   evaluation through the heuristic function.    */

calc(A,B,V,Level):- comp(A,B,W),U is W+Level.

comp(A,B,0).
comp(A,B,W):- simplify(A,A1),simplify(B,B1),misplace(A1,B1,W).

simplify([L],L).
simplify([L|L1],NL):- conc(L,L2,NL),simplify(L1,L2).

conc([],L,L).
conc([N|T],L,[N|U]) :- conc(T,L,U).

misplace([],[],0).
misplace([H1|L1],[H2|L2],Z):- misplace(L1,L2,Z1),
                                ((H1==H2,Z is Z1);
                                 (H1\==H2,Z is Z1+1)). }W

/* Insert sorting */

insort([H|T],S):- insort(T,L),plugin(H,L,S).
insort([],[]).

plugin(suc
        (Matrix,U,N),[suc(Matrix2,U2,N2)|T],[suc(Matrix2,U2,N2)|L]):-
        V2(U,! ,plugin(suc(Matrix,U,N),T,L)).
plugin(X,L,[X|L]). 

/* Search */

go1([suc(H1,_,Level1)|R],V,T,N):- N==Level1,go(H1,Y,T,R,Level1).
go1([suc(H1,V,Level1)|LMX|T1],N):- N\==Level1,

```

```

M is N-1,
delete(N,L,L1),
go1([suc(H1,U,Level1)|L1],Y,
     T1,M).

delete(_,[],[]).
delete(N,[suc(X,Y,N)|L],M):- !, delete(N,L,M).
delete(N,[suc(X,Y,I)|L1],[suc(X,V,I)|L2]):- delete(N,L1,L2).

reverse([],[]).
reverse([H|T],L):- reverse(T,R), conc(R,[H],L).

/* Messages. */

print1(X,Y):- nl,nl,write('      8-PUZZLE'),nl,nl,nl,
             write('Initial configuration:'),nl,nl,
             write(X),nl,nl,
             write('Final configuration:'),nl,nl,
             write(Y),nl,nl.

print2([]).
print2([X|T]):- write(X),nl,
              print2(T).

```

Execution:

```

?- go([[1,2,3],[4,5,6],[b,7,8]],[[1,2,3],[4,5,6],[7,8,b]],[],[],0).

      8-PUZZLE

Initial configuration:
[[1,2,3],[4,5,6],[b,7,8]]

Final configuration:
[[1,2,3],[4,5,6],[7,8,b]]

List of successor nodes:
[suc([[1,2,3],[4,5,6],[7,b,8]],3,1),suc([[1,2,3],[b,5,6],[4,7,8]],5,1)]

Unordered list of generated nodes not expanded:
[suc([[1,2,3],[4,5,6],[7,b,8]],3,1),suc([[1,2,3],[b,5,6],[4,7,8]],5,1)]

Unordered list of generated nodes not expanded:
[suc([[1,2,3],[4,5,6],[7,b,8]],3,1),suc([[1,2,3],[b,5,6],[4,7,8]],5,1)]

List of visited nodes:
[[[1,2,3],[4,5,6],[b,7,8]]]

List of successor nodes:
[suc([[1,2,3],[4,5,6],[7,8,b]],2,2),suc([[1,2,3],[4,5,6],[7,8,b]],2,2),
suc([[1,2,3],[4,b,6],[7,5,8]],5,2)]

```

Unordered list of generated nodes not expanded:

```
[suc([[1,2,3],[b,5,6],[4,7,8]],5,1),suc([[1,2,3],[4,5,6],[7,8,b]],2,2),
suc([[1,2,3],[4,5,6],[7,8,b]],2,2),suc([[1,2,3],[4,b,6],[7,5,8]],5,2)]
```

Unordered list of generated nodes not expanded:

```
[suc([[1,2,3],[4,5,6],[7,8,b]],2,2),suc([[1,2,3],[4,5,6],[7,8,b]],2,2),
suc([[1,2,3],[b,5,6],[4,7,8]],5,1),suc([[1,2,3],[4,b,6],[7,5,8]],5,2)]
```

List of visited nodes:

```
[[[1,2,3],[4,5,6],[7,b,8]],[[1,2,3],[4,5,6],[b,7,8]]]
```

Solution:

```
[[1,2,3],[4,5,6],[b,7,8]]
[[1,2,3],[4,5,6],[7,b,8]]
[[1,2,3],[4,5,6],[7,8,b]]
```

Comments:

The program is triggered on by the predicate ‘go(X,Y,T,D,Level)’ where ‘X’ describes the initial configuration, ‘Y’ stands for the final configuration, ‘T’ stores all the state transitions between ‘X’ and ‘Y’, ‘D’ is the list of all generated nodes that were not expanded, and ‘Level’ corresponds to the depth of the search tree. For example, when we want a change:

	2 8 3	1 2 3		
from	1 6 4	to	8 4	b=blank
	7 5		7 6 5	

we write:

```
go([[2,8,3],[1,6,4],[7,b,5]],
    [[1,2,3],[8,b,4],[7,6,5]],
    [ ],
    [ ],
    0).
```

The predicate ‘findall’ computes the list ‘D1’ of all successors of the node to be expanded and the values of the evaluation function ‘suc(Z,V)’. These values are the result of all possible moves and, therefore, are not placed yet in ‘T’. As a result of applying ‘goodnode’, the node to be expanded is placed in ‘T’. The predicate ‘conc(D,D1,D2)’ allows the list of successor nodes, which are sorted by the predicate ‘insort’, to be expanded. The predicate ‘go1’ is in charge of going on the search. The goal state is attained when it is the same as the current node. Therefore, ‘go(X,X,T,_,_)’ is satisfied and the execution stops.

Problem 80 [Townsend 1982, personal communication]

Verbal statement:

Write a program to play tic-tac-toe (noughts and crosses).

Logic program:

```

start:- play('x','o',
            ['*','*','*',
             '*','*','*',
             '*','*','*']),
        start.

play(X,Y,Board):- won(Y,Board),nl,write('Win for '),write(Y),nl.
play(X,Y,Board):- full(Board),nl,write('Draw'),nl.
play(X,Y,Board):- move(X,Board,Board1),play(Y,X,Board1),
                nl,write('I resign'),nl.

move('x',Board,Board1):- show(Board),nl,
                        write('Enter col and row'),
                        read(Col),read(Row),
                        update('X',Col,Row,Board,Board1),nl,
                        write('Your move'),nl,
                        show(Board1).

move('x',Board,Board1):- nl,write('Invalid move'),nl,
                        move('x',Board,Board1).

move('o',Board,Board1):- amove('o',Board,Board1),
                       okmove('o','k',Board1).

show(Board):- nl,rshow(Board),nl,nl.

rshow([]).
rshow([A,B,C|Rest]):- nl,write(A),write(B),write(C),rshow(Rest).

amove(X,Board1,Board2):- update(X,1,1,Board1,Board2);
                        update(X,1,2,Board1,Board2);
                        update(X,1,3,Board1,Board2);
                        update(X,2,1,Board1,Board2);
                        update(X,2,2,Board1,Board2);
                        update(X,2,3,Board1,Board2);
                        update(X,3,1,Board1,Board2);
                        update(X,3,2,Board1,Board2);
                        update(X,3,3,Board1,Board2).

okmove(X,Y,Board):- won(x,Board).
okmove(X,Y,Board):- amove(Y,Board,Board1),
                  won(Y,Board1),!,fail.
okmove(_,_,_).

full([]).
full(['*' | _]):- !,fail.
full([_|T]):- full(T).

won(X,[X,X,X,-,-,-,-,-,-]). 
won(X,[-, -, -, X,X,X,-,-,-]). 
won(X,[-, -, -, -, -, X,X,X]).
```

```

won(X,[X,_,_,X,_,_,X,_,_]) .
won(X,[_,X,_,_,X,_,_,X,_]) .
won(X,[_,_,X,_,_,X,_,_,X]) .
won(X,[X,_,_,_,X,_,_,_,X]) .
won(X,[_,_,X,_,X,_,X,_,_]) .

update(X,1,1,['*',A2,A3,A4,A5,A6,A7,A8,A9],
[X,A2,A3,A4,A5,A6,A7,A8,A9]) .
update(X,2,1,[A1,'*',A3,A4,A5,A6,A7,A8,A9],
[A1,X,A3,A4,A5,A6,A7,A8,A9]) .
update(X,3,1,[A1,A2,'*',A4,A5,A6,A7,A8,A9],
[A1,A2,X,A4,A5,A6,A7,A8,A9]) .
update(X,1,2,[A1,A2,A3,'*',A5,A6,A7,A8,A9],
[A1,A2,A3,X,A5,A6,A7,A8,A9]) .
update(X,2,2,[A1,A2,A3,A4,'*',A6,A7,A8,A9],
[A1,A2,A3,A4,X,A6,A7,A8,A9]) .
update(X,3,2,[A1,A2,A3,A4,A5,'*',A7,A8,A9],
[A1,A2,A3,A4,A5,X,A7,A8,A9]) .
update(X,1,3,[A1,A2,A3,A4,A5,A6,'*',A8,A9],
[A1,A2,A3,A4,A5,A6,X,A8,A9]) .
update(X,2,3,[A1,A2,A3,A4,A5,A6,A7,'*',A9],
[A1,A2,A3,A4,A5,A6,A7,X,A9]) .
update(X,3,3,[A1,A2,A3,A4,A5,A6,A7,A8,'*'],
[A1,A2,A3,A4,A5,A6,A7,A8,X]) .

```

Comments:

When there are a number of clauses whose heads all have the same predicate they form a procedure for that predicate, and the construction is equivalent to a CASE statement. The procedure 'play' is a deterministic program in the sense that the alternative implications are assumed to be mutually exclusive. If the attempt to prove an alternative fails after partial evaluation then this attempt is abandoned and the next alternative tried. The relation 'show' is responsible for showing the board of the game. The relation 'rshow' is defined recursively on a list as triples, with each show on a new line.

Chapter 8 Searching with Prolog

Exploring and finding alternative paths through a network or a tree structure is often necessary while solving problems. Two issues become relevant: search, focusing on the methods for exploring the structures that describe the problem domains, and control, focusing on how the problem solver selects those methods and shifts attention among its subprocesses.

In general, control is regarded as a special case of search. In the following examples, the methods are described in logic, and they are separated as much as possible from the control machinery.

Problem 81

Verbal statement:

A road map is sketched as follows:

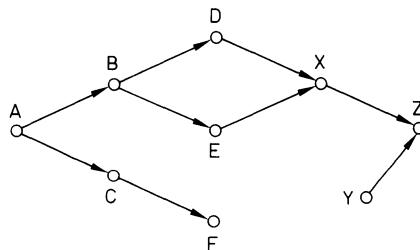


Fig. 1

Present the most appropriate representation for the map. Show the generation of the top-down and the bottom-up search spaces.

(This problem was suggested by Robert Kowalski).

Logic program:

go(a).	go(z):- go(y)
go(b):- go(a)	go(c):- go(a)
go(d):- go(b)	go(f):- go(c).
go(e):- go(b)	go(x):- go(d).
go(z):- go(x).	go(x):- go(e)

Execution:

The top-down search space is triggered by the following query:

```
?- go(z).
```

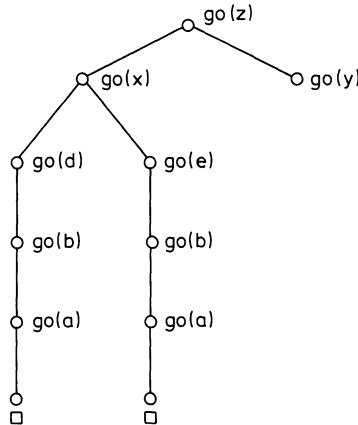


Fig. 2

The bottom-up search space is triggered by the following query:

```
?- go(a).
```

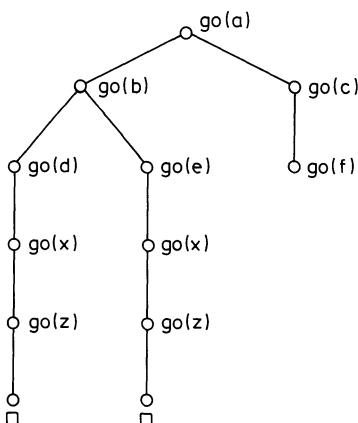


Fig. 3

Problem 82

Verbal statement:

Natural numbers can be defined on the notion of successor. Show the search space generation for the number 2 when triggering the logic program either in the top-down or in the bottom-up direction.

(This problem was suggested by Robert Kowalski).

Logic program:

```
natural(0)
natural(s(X)):- natural(X).
```

Execution:

```
?- natural(s(s(0))).
```

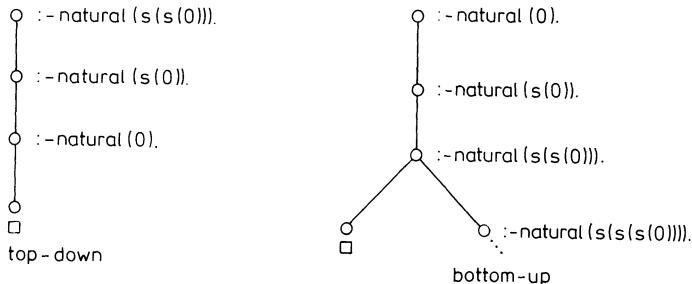


Fig. 4

Comments:

The representation for the natural number 2 is $s(s(0))$.

Problem 83 [Clocksin; Mellish 1981]

Verbal statement:

Describe the algorithm for integer division based upon the operations of addition and multiplication.

Logical program:

```
integer_division(N1,N2,Result):- integer(Result),
                                Product1 is Result*N2,
                                Product2 is (Result+1)*N2,
                                Product1=<N1,
                                Product2>N1, !.
```

Comments:

This program exemplifies the method “generate and test”. The predicate ‘integer’ is the generator device, and the other four conditions are the testing device.

The generator can produce all the possible candidates, but only one can be filtered by the conditions. When the candidate is discovered the mechanism for the integer division stops. The cut symbol ‘!’ is in charge of controlling the backtracking, and therefore it stops the generator when the candidate is obtained.

In brief, the “generate and test” paradigm is achieved by inserting a non-deterministic predication among the concatenated goals of the ‘integer_division’ relation. The test will repeatedly fail causing the non-deterministic relation to be re-defined until a useful value is produced enabling the overall relation to be established.

Problem 84

Verbal statement:

There were three departures for a Monte Carlo rally:

London: John (BMW), Tommy (FORD), Fred (BMW), Anne (FORD) and Teddy (BMC)

Paris: Guy (FIAT), Claude (FORD), Jean (FIAT), and Brigitte (BMC)

Lisbon: Luis (FIAT), Carlos (BMW), Lucas (BMC), Pedro (FORD), and Nuno (FIAT).

and only four drivers, Tommy, Fred, Pedro and Nuno, arrived at Monte Carlo.

Which cars did arrive at Monte Carlo?

From where did the drivers that arrived at Monte Carlo leave ?

Logic program:

```
rally:- drivers(A,london),
        drivers(B,lisbon),
        drivers(C,paris),
        drivers(D,bmw),
        drivers(E,fiat),
        drivers(F,ford),
        drivers(G,bmc),
        drivers(H,monte_carlo),
        intersection(A,H,Lon),
        intersection(B,H,Lis),
        intersection(C,H,Par),
        intersection(D,H,Bmw),
        intersection(E,H,Fiat),
        intersection(F,H,Ford),
        intersection(G,H,Bmc),nl,nl
        write('Arrived:'),nl,nl,
        write('From London: '),write(Lon),nl,
        write('From Lisbon: '),write(Lis),nl,
        write('From Paris: '),write(Par),nl,nl,
        write('in BMW: '),write(Bmw),nl,
        write('in FIAT: '),write(Fiat),nl,
        write('in FORD: '),write(Ford),nl,
        write('in BMC: '),write(Bmc),nl.
```

```

intersection([],X,[]).
intersection([X|R],Y,[X|Z]):- member(X,Y),!
                                intersection(R,Y,Z).
intersection([X|R],Y,Z):- intersection(R,Y,Z).

member(X,[X|_]). 
member(I,[_|T]):- member(I,T).

drivers([john,tommy,fred,anne,teddy],london).
drivers([guy,claude,jean,brigitte],paris).
drivers([luis,carlos,lucas,pedro,nuno],lisboa).
drivers([fred,carlos,john],bmw).
drivers([luis,nuno,jean,guy],fiat).
drivers([anne,tommy,claude,pedro],ford).
drivers([teddy,brigitte,lucas],bmc).
drivers([tommy,fred,pedro,nuno],monte_carlo).

```

Execution:

The execution of:

?-rally.

gives us the following results:

Arrived:

```

From London : [tommy,fred]
From Lisbon : [nuno,pedro]
From Paris  : []
in BMW    : [fred]
in FIAT   : [nuno]
in FORD   : [pedro,tommy]
in BMC    : []

```

Problem 85 [Baxter 1979, personal communication]

Verbal statement:

A situation involve five houses in a row amongst which are distributed five colors, five drinks, five nationalities, five cigarettes, and five pets subject to the following constraints:

- The Englishman lives in the red house.
- The Spaniard owns a dog.
- The Norwegian lives in the first house.
- Kools are smoked in the yellow house.
- Chesterfields are smoked next to where the fox is kept.

The Norwegian lives next to the blue house.
 The Old Gold smoker owns snails.
 The Lucky Strike smoker drinks orange juice.
 The Ukrainian drinks tea.
 The Japanese smokes Parliaments.
 The Kools smoker lives next to where the horse is kept.
 Coffee is drunk in the green house.
 The green house is to the immediate right of the ivory house.
 Milk is drunk in the middle house.

The problems to solve are: Who drinks water? Who keeps the zebra?

Logical program 1:

```
/* Choose and verify method */
:- op(800,xfx,':').

solve:- 
    candidate(Colours,Drinks,Nationalities,Cigarettes,Pets),
    constraints(Colours,Drinks,Nationalities,Cigarettes,Pets),
    member(water: House1,Drinks),
    member(Who1: House1,Nationalities),write(Who1),nl,
    member(zebra: House2,Pets),
    member(Who2: House2,Nationalities),write(Who2),nl.

candidate
([red:C1,yellow:C2,blue:C3,green:C4,ivory:C5],
 [orange_juice:D1,tea:D2,coffee:D3,milk:D4,water:D5],
 [englishman:N1,spaniard:N2,norwegian:N3,ukrainian:N4,
  japanese:N5],
 [kools:S1,chesterfield:S2,old_gold:S3,lucky_strike:S4,
  parliaments:S5],
 [dog:P1,fox:P2,snails:P3,horse:P4,zebra:P5]):-
    permutation([C1,C2,C3,C4,C5],[1,2,3,4,5]),
    permutation([D1,D2,D3,D4,D5],[1,2,3,4,5]),
    permutation([N1,N2,N3,N4,N5],[1,2,3,4,5]),
    permutation([S1,S2,S3,S4,S5],[1,2,3,4,5]),
    permutation([P1,P2,P3,P4,P5],[1,2,3,4,5]).

constraints(C,D,N,S,P):-
    member(englishman:House1,N),member(red:House1,C),
    member(spaniard:House2,N),member(dog:House2,P),
    member(norwegian:1,N),
    member(kools:House3,S),member(yellow:House3,C),
    member(chesterfield:House4,S),
    next(House4,Next4),member(fox:Next4,P),
    member(norwegian:House5,N),next(House5,Next5),
    member(blue:Next5,C),
```

```

member(old_gold:House6,S),member(snails:House6,P),
member(lucky_strike:House7,S),member(orange_juice:House7,D),
member(ukrainian:House8,N),member(tea:House8,D),
member(japanese:House9,N),member(parliaments:House9,S),
member(kools:House10,S),next(House10,Next10),
    member(horse:Next10,P),
member(coffee:House11,D),member(green:House11,C),
member(green:House12,C),left(Left12,House12),
    member(ivory:Left12,C),
member(milk:3,D).

permutation([],[]).
permutation([A|X],Y):- delete(A,Y,Y1),permutation(X,Y1).

delete(A,[A|X],X).
delete(A,[B|X],[B|Y]) :- delete(A,X,Y).

member(A,[A|_]). 
member(A,[B|X]) :- member(A,X).

next(X,Y) :- left(X,Y).
next(X,Y) :- left(Y,X).

left(1,2).
left(2,3).
left(3,4).
left(4,5).

```

Logic program 2:

```

/* Verify and choose method */
:- op(800,xfx,':').

solve:- 
    constraints(Colours,Drinks,Nationalities,Cigaretts,Pets),
    candidate(Colours,Drinks,Nationalities,Cigaretts,Pets),
    member(water:House1,Drinks),
    member(Who1:House1,Nationalities),write(Who1),nl,
    member(zebra:House2,Pets),
    member(Who2:House2,Nationalities),write(Who2),nl.

candidate([_:C1, _:C2, _:C3, _:C4, _:C5],
          [_:D1, _:D2, _:D3, _:D4, _:D5],
          [_:N1, _:N2, _:N3, _:N4, _:N5],
          [_:S1, _:S2, _:S3, _:S4, _:S5],
          [_:P1, _:P2, _:P3, _:P4, _:P5]):-

permutation([C1,C2,C3,C4,C5],[1,2,3,4,5]),
permutation([D1,D2,D3,D4,D5],[1,2,3,4,5]),
permutation([N1,N2,N3,N4,N5],[1,2,3,4,5]),
permutation([S1,S2,S3,S4,S5],[1,2,3,4,5]),
permutation([P1,P2,P3,P4,P5],[1,2,3,4,5]).
```

```

/* The constraints are the same as in the "Choose and Verify"
program */

/* The permutation definition is the same */

member(A,[A|_]):- !.
member(A1:A2,[B1:B2|X]):-
    atomic(A1),atomic(A2),atomic(B1),atomic(B2),
    (A1==B1,A2=\=B2; A2==B2,A1=\=B1),!,fail.
member(A,[_|X]):- member(A,X).

next(X,Y):- left(X,Y);left(Y,X).

left(1,2).
left(2,3).
left(3,4).
left(4,5).

```

Execution:

```
?- solve.
```

Comments:

This is a typical combinatorial problem where Prolog presents a good answer by enhancing two key characteristics: program size and efficiency.

Humans require about half an hour to solve this problem using essentially a trial and error method. The above programs adopt the “choose and verify” and “the verify and choose” methods. The program generates $(5!)^{**}5$ possibilities, and for each choice of a candidate solution it verifies whether the constraints are met. A program written in a procedural language based on operational semantics takes a microsecond to verify the constraints, and the computation time is about 5 hours. The Prolog program solves this problem in about 5 seconds!

The chosen representation for the data is based upon the infix operator ‘:’, e.g. ‘blue:5’ means that the 5th house is blue.

The first logic program adopts the “choose and verify method”,

```
candidate(X),constraints(X)
```

where a possible instance of ‘X’ is first found, and only afterwards is it verified whether it satisfies certain constraints.

But this requires excessive computation. The second program adopts the “verify and choose” method,

```
constraints(X),candidate(X)
```

where a possible instance of ‘X’ satisfying the constraints is partially specified. This approach takes full advantage of Prolog, because a computation may produce a partial result which becomes complete after some future computation. This second program is very efficient. Using the DEC-10 (KI-10) Prolog compiler (Edinburgh), a solution is obtained in 3.8 seconds.

Problem 86 [Clocksin; Mellish 1981]

Verbal statement:

A house has six rooms and seven doors as shown in the following figure. A telephone is in room g.

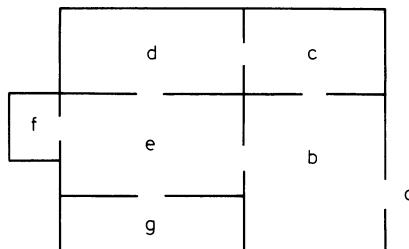


Fig.5

Find the shortest way to reach the phone, and find another way without entering rooms d and f. Consider a depth-first strategy.

Logic program:

```

door(a,b).  door(b,c).  door(b,d).  door(d,e).  door(c,d).
door(e,f).  door(g,e).

has_phone(g).

go(X,X,T).
go(X,Y,T):- (door(X,Z);door(Z,X)),not(member(Z,T)),
            not(member(Y,T)),go(Z,Y,[Z|T]).

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

not(X):- X,! ,fail.
not(_).

```

Execution:

```

?- go(a,X,[]),has_phone(X).
?- go(a,_,[d,f]).

```

Comments:

The predicate ‘go’ defines a path between rooms ‘X’ and ‘Y’, avoiding the rooms in ‘T’. The second clause of ‘go’ declares that to go from ‘X’ to ‘Y’, avoiding the rooms in ‘T’, it is necessary to find a door from room ‘X’ to room ‘Z’. After checking the candidate ‘Z’ is not in ‘T’, go from ‘Z’ to ‘Y’ adding ‘Z’ to ‘T’.

The program is based upon the depth-first strategy which states that only one of the neighbors of each node of the graph is visited. The other neighbors are ignored till a failure provokes backtracking.

Problem 87 [Clocksin; Mellish 1981]

Verbal statement:

A maze is described by a set of nodes of a graph:

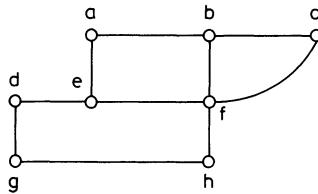


Fig. 6

Write a program able to travel through a maze by choosing one neighbor to visit, after considering all the immediate neighbors of each node (neighborhood-first strategy). The node g is the starting point and the node a is the goal to be reached.

Logic program:

```

good_node(X,Y,T) :- (a(X,Y); a(Y,X)), not(member(Y,T)).
go(X,X,_).
go(X,Y,T) :- find_all(Z,good_node(X,Z,T),D),
            go_all(D,Y,T).
go_all([H|_],Y,T) :- go(H,Y,[H|T]).
go_all([_|L],Y,T) :- go_all(L,Y,T).
find_all(X,G,_) :- asserta(found(mark)), G,
                  asserta(found(X)), fail.
find_all(,-,L) :- collect_found([],M), !, L=M.
collect_found(S,L) :- get_next(X), !, collect_found([X|S],L).
collect_found(L,L).
get_next(X) :- retract(found(X)), !, X=\=mark.
member(X,[X|_]). 
member(X,[_|Y]) :- member(X,Y).
not(X) :- X,!, fail.
not(_).
  
```

Execution:

```
?- go(g,a).
[a,f,d,b,e,h,c]
[f,e,g]
[d,h]
```

Comments:

The predicate ‘good_node’ checks whether a node was visited previously. The predicate ‘find_all’ is able to build up a list of known objects with a given property. It finds all the immediate neighbors that are able to pass the good node test, and it places them in ‘D’. The built-in predicate ‘asserta’ is in charge of storing the objects in an internal database, controlled by a flag called mark. The predicate ‘collect_found’ collects all the objects stored, through the predicate ‘get_next’, and it places them in a list ‘L’.

Observe the control installed in ‘collect_found’ and ‘get_next’ through ‘!’. When a failure occurs, the flag is found, ‘collect_found’ is solved by the second clause, where the second argument is instantiated (the result) with the list already built.

Problem 88 [Clocksin; Mellish 1981]

Verbal statement:

A road map is represented by the following graph:

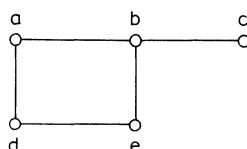


Fig. 7

Write a program to find the shortest path between two towns based upon the best-first strategy. Write another program based upon the breadth-first strategy.

Logic program 1:

```
/* Best-first strategy */
```

```
go1(X,X,_).
go1(X,Y,T):- find_all(desc(Z,Dist),good_node(X,Z,Dist,T),D),
in_sort(D,D1),write(D),nl,write(D1),nl,
go1_all(D1,Y,T),write(T).

good_node(X,Y,D,T):-(road(X,Y,D);road(Y,X,D)),not(member(Y,T)).
```

```

go1_all([desc(H,_)|_],Y,T):- go1(H,Y,[H|T]).  

go1_all([_|L],Y,T):- go1_all(L,Y,T).  

in_sort([H|T],S):- in_sort(T,L), plug_in(H,L,S).  

in_sort([],[]).  

plug_in(desc(Town,Dist),[desc(Town2,Dist2)|T],  

        [desc(Town2,Dist2)|L]):-  

        Dist2<Dist,!,  

        plug_in(desc(Town,Dist),T,L).  

plug_in(X,L,[X|L]).  

road(a,b,5).  

road(b,e,2).  

road(d,a,4).  

road(e,d,5).  

road(c,b,3).  

road(c,e,3).

```

Execution:

```
?- go(a,e,[]).  
[d]
```

Comments:

The program is not searching for the shortest path. In fact, after building up a list of neighbors of a node, the list is sorted and only the closest neighbor is searched in the first place. The predicate ‘desc(X,Y)’ represents the list of structures corresponding to the neighbors, where ‘X’ stands for the name of the town and ‘Y’ for the distance of this town from the current position. The predicate ‘good_node(X,Y,T)’ avoids roads with only one direction, and tests whether a node was already visited. This predicate is successful when there is an arc from ‘X’ to ‘Y’ or from ‘Y’ to ‘X’ and when ‘Y’ is not a member of ‘T’. The second clause of ‘go(X,Y,T)’ finds the immediate neighbors that pass the examination of ‘good_node’, and places them in list ‘D’. Afterwards, it goes on searching through the closest neighbors. The predicate ‘go_all’ is designed to fail when it reaches the end of the list of nodes to be searched. The first clause of ‘go_all’ applies ‘go’ to the head of the list, and the second clause applies ‘go_all’ to the tail of the list in a recursive way. The predicate ‘in_sort’ is a sort method. Each member of the list is considered separately and inserted in a new list, in the right spot. The sorting of the immediate neighbors is achieved in order to start the search through the closest one.

Logic program 2:

```
/* Breadth-first search */  

go(X,Y):- go1([X],Y).  

go1(L,D):- member(D,L),!.  

go1(L,D):- go2(L,[],M),go1(M,D),write(M),nl.
```

```

go2([],L,L):- !.
go2([N|Ns],Sf,L):- find_all(D,(r(N,D);r(D,N)),Ds),
add_on(Ds,Sf,Sf1),go2(Ns,Sf1,L).

add_on([],L,L).
add_on([N|Ns],L,M):- member(N,L),!, add_on(Ns,L,M).
add_on([N|Ns],L,M):- add_on(Ns,[N|L],M).

```

Problem 89 [Baxter 1980]

Verbal statement:

Write the alpha-beta pruning procedure in Prolog. This procedure consists of two aspects: 1) whenever something is found about the best that can be hoped for at a given node, test what is known about the parent node. It may be that no further work is sensible below the given node; 2) whenever the exact game value of a node is established, test what is known about the nodes above. It may be that the best that can be hoped for at the parent node can be revised or fixed exactly. (See the specification in Artificial Intelligence Vol. 6, No. 4, 1975 and consult Artificial Intelligence by Patrick Winston, Addison-Wesley, 1977).

Logic program:

```

evaluate:- game_tree(T),prune(T,-999,999,V),write(V),nl.

prune([P|Ps],A,B,V):-
prune(P,-B,-A,V1),maximum(A,-V1,A1),
(A1>=B,V=A1; prune(Ps,A1,B,V)).

prune([],A,B,A).
prune(V,A,B,V).

maximum(X,Y,X):- X>=Y.
maximum(X,Y,Y). /* Y>X */

game_tree(
[
[[[3,1,4],[1,5,9],[2,6,5]],[[3,5,8],[9,7,9],[3,2,3]],[[8,4,6],
[2,6,4],[3,3,8]]],
[[[3,2,7],[9,5,0],[2,8,8]],[[4,1,9],[7,1,6],[9,3,9]],
[[9,3,7],[5,1,0],[5,8,2]]],
[[[0,9,7],[4,9,4],[4,5,9]],[[2,3,0],[7,8,1],[6,4,0]],
[[6,2,8],[6,2,0],[8,9,9]]]]).

```

Chapter 9 Learning with Prolog

This chapter concerns learning problem solving, and in particular adding knowledge and clause modification. The last issue involves modifying the logic of clauses and the conditions under which the individual clauses may be applied. The method of discrimination, used in various learning systems, is adopted to modify a given clause so that it would be applicable in some contexts and fail in others. This modification process enables a system to adapt its knowledge to specific requirements.

Problem 90 [Grumbach 1983]

Verbal statement:

Write a program to tell whether a sentence is semantically correct or not. Expand the program so that it can accept correct sentences (learn) for which it has previously given a wrong answer. The final program provides the user with a tool that allows him to both ask questions and enter information in the same natural declarative way, i.e. questions and assertions.

Logic program:

```
/* Dictionary */
class_sem(henry,human).
class_sem(lenina,human).
class_sem(eats,eat).
class_sem(drinks,drink).
class_sem(cakes,food).
class_sem(coffee,beverage).
class_sem(tea,beverage).

/* Knowledge base */
link_sem(human,eat).
link_sem(eat,food).
link_sem(human,drink).

is_true(P):- learn(sem(P,yes),link_sem(X,Y)).
```

```

learn(Q,R):- assertz(R:-assert(R)),
            Q,
            retract(R:-assert(R)).

sem(P,ANSWER):- sem_class(P,PSC),
               sem_verify(PSC,ANSWER).

sem_class([],[]).
sem_class([X|T],[Y|R]) :- class_sem(X,Y), sem_class(T,R).

semi_verify([_|[]],yes).
sem_verify([A,B|T],yes) :- link_sem(A,B),
                         sem_verify([B|T],yes).
sem_verify(_,no).

```

Execution:

```

?- sem([henry,eats,cakes],ANSWER).
ANSWER=yes

?- sem([henry,drinks,coffee],ANSWER).
ANSWER=no

?- is_true([henry,drinks,coffee]).
```

(Will add the fact: link_sem(drink, beverage).
Now sem([lenina,drinks,tea],yes) is also true.)

A trace of the execution of the first goal is:

```

CALL sem([henry,eats,cakes],ANSWER)
CALL sem_class([henry,eats,cakes],C)
...
EXIT sem_class([henry,eats,cakes],[human,eat,food])
CALL sem_verify([human,eat,food],ANSWER)
    CALL link_sem(human,eat)
    EXIT link_sem(human,eat)
    CALL link_sem(eat,food)
    EXIT link_sem(eat,food)
    EXIT sem_verify([human,eat,food],yes)
EXIT sem([henry,eats,cakes],yes)
```

Comments:

The term ‘sem – class’ replaces each word by its semantic equivalent which is read in the dictionary. Then the sentence is divided into groups beginning and ending with “semantic words” (nouns, verbs), and the program verifies whether each group holds in the knowledge base.

The updating is based on something like a view of the knowledge base. The learning technique, implemented in the above program, consists in automatizing the adding of information. A good way to add the information by hand is to modify the very program, to integrate the new information in the same way as the pre-

existent one. The programmer, when building his program, knows that the procedure to be argumented is ‘link – sem(X,Y)’, but he does not know the corresponding values of ‘X’ and ‘Y’, without tracing the execution. Observe that ‘link – sem(drink,beverage)’ is missing.

In the above program, all this is solved automatically by adding an auxiliary clause ‘is – true’. The learning facility is obtained by a first term that adds one catch-all clause ‘link – sem’ at the end of the knowledge base. The second term (sem(P,yes)) parses the sentence, in the normal way. During the execution of a goal ‘is – true’, the catch-all clause is activated with ‘X’ and ‘Y’ becoming instantiated: it adds the new information (new ‘link – sem’) to the knowledge base. Finally, the third term of ‘learn’ deletes the catch-all clause.

Problem 91 [Grumbach 1983]

Verbal statement:

You are requested to provide the user of some database with tools such that updating can be done in the same natural declarative manner as the queries.

Take an example of a relational database describing courses and assume you want to make the following updating:

1. henry will lecture course – 3.
2. course – 2 lecture will be given on wednesday 18h in the same room.
3. course – 3 lecture on tuesday 16h is cancelled.
4. tuesday henry’s lecture is cancelled.
5. course – 1 will take place in room – 3 instead of room – 1.

Logic program:

```

teacher(linda,course_1).
teacher(lenina,course_1).
teacher(lenina,course_2).

room_day(course_1,room1,monday,14).
room_day(course_2,room1,tuesday,14).
room_day(course_3,room1,monday,16).
room_day(course_3,room2,tuesday,16).

is_true(P):- add_catch_all(teacher(T,C)),
            add_catch_all(room_day(C,R,D,H)),
            P.

add_catch_all(P):- assertz((P:- retract((P:- retract(_),
                                         !,constraints(P),
                                         assert(P))),
                           !,constraints(P),
                           assert(P))),!.
```

```

is_false((P,Q)):- P,
    is_false (Q).          /* to hold predicates */
                           /* sequences */
is_false(P):- P,retract(P).      /* the main clause */
is_false(_).                   /* to get true anyway */

constraints(room_day(C,R,D,H)):- 8<H,H<20,
    not(room_day(_,R,D,H)).
```

```

not(X):- X,! ,fail.
not(_).
```

Execution:

For the first updating:

```
?- is_true(teacher(henry,course_3)).
```

For the second updating:

```
?- is_true((room_day (course_2,R,_,_),
            room_day(course_2,R,wednesday,18))).
```

For the 3rd updating:

```
?- is_false(room_day(course_3,_,tuesday,16)).
```

For the 4th updating:

```
?- is_false((teacher(henry,C),room_day(C,R,tuesday,16))).
```

For the 5th updating:

```
?- is_false(room_day(course_1,room1,D,H)),
   is_true(room_day(course_1,room3,D,H)).
```

Problem 92 [Grumbach 1983]

Verbal statement:

You are requested to provide the user of some knowledge base with tools such that updating can be done in the same natural declarative manner as the queries, in situations where there are several possible modifications.

Take the example of a knowledge base with animal classification, with rules such as:

r1: if the animal has hair, then it is a mammal

The user enters characteristics of an animal, for example it has hair, it eats meat, it has black stripes, and the system deduces the corresponding animal, by application of successive rules.

Logic program:

```

/* Operators necessary to interpret rules */

:- op(101, xfy, 'and').
:- op(102, xfy, 'which').
:- op(103, fx, 'a').
:- op(104, xfy, 'is').

/* Rules */

rule(1, a animal which 'has hair' is a mammal).
rule(2, a animal which 'gives milk' is a mammal).
rule(3, a animal which 'has feathers' is a bird).
rule(5, a mammal which 'eats meat' is a carnivorous).
rule(10, a carnivorous which 'has black stripes' is a tiger).

/* Rules interpreter */

family(N1, X, Y) :- rule(N2, a X which C is a Z),
    char(C),
    family(N2, Z, Y).
family(_, X, X) :- not(rule(_, a X which_ is a_)).

/* ('char' for 'characteristics') */

char((C1 and C2)) :- char(C1), char(C2).
char(C) :- ask(C, ANSWER), ANSWER=yes, !.

/* Requests with learning features */

query(LIST_CHAR, ANSWER) :- assert_char(LIST_CHAR),
    family(_, animal, ANSWER).

is_true(LIST_CHAR, ANIMAL) :-
    assertz((rule(0, a X which C is a Y) :- true_update(X, C, Y))),
    query(LIST_CHAR, ANIMAL),
    retract((rule(0,_) :- true_update(_,_,_))). 

is_false(LIST_CHAR, ANIMAL) :-
    assertz((rule(0, a X which C is a Y) :- false_update(X, C, Y))),
    non(query(LIST_CHAR, ANIMAL)),
    retract((rule(0,_) :- false_update(_,_,_))). 

/* Updating */

true_update(X, C, Y) :- make_new_number_rule(N),
    list_all_char(LC1),
    not_already_tested_char(LC1, LC),
    assert(rule(N, a X which LC2 is a Y)).

```

```

false_update(X,C,Y):- used_rules(LIST_RULES),
                     suspected_rule(LIST_RULES,SUSPECTED_RULE),
                     no_more_deducible(SUSPECTED_RULE),
                     retract(rule(SUSPECTED_RULE)),
                     !, fail.

/* Other procedures */

non(P):- P,fail.
non(_).

```

Execution:

```

?- query(['has hair','eats meat','h.bl.st.'],ANIMAL).
?- is_true(['has hair','eats meat','h.bl.st.'],tiger).
?- is_false(['has hair','eats meat','h.bl.st.'],rabbit).

```

The deduction sequence involves three rules: r1, r5 and r10. The 'is_false' request tells us that 'has_hair' + 'eats meat' + 'has black stripes + rabbit + r1 + r5 + r10 is false. It's assumed user's arguments are correct; so, r1 + r5 + r10 is false. The program could have asked the user for the accuracy of each of these rules, but it prefers to provide him with external views.

Comments:

The knowledge base contains rules; its updating will then consist in the addition and/or retrieval of rules. The problem is to locate bad rules and/or create new correct ones.

The method proposed by Davis within the Teiresias system is based upon three steps: 1) showing back the deduction sequence to the user; 2) tracking down the bug interactively, asking for guilty rules, and deleting them, asking for new rules, and interpreting them; and, 3) checking the correctness of the new knowledge base (see Davis & Lenat, Knowledge-based Systems in Artificial Intelligence, McGraw-Hill, 1982).

The program implements this method: the 'fail' at the end of the deduction is recovered by a catch-all rule; the deduction sequence is reachable through the resolution tree (by an 'ancestors' predicate); the user is asked for complementary information; a resolution with the new rules is started to check the modification; and, a final check (for consistency and completeness) and consequent actions, maybe involving another interactive session, is performed.

The default tracking and the knowledge base modification implemented in the program is close to the above method. But, it differs widely in the conception of the interactive session. The system queries do not mention rules, but views of these rules through examples.

The catch-all rule is performed by 'r(0, a X which C is a Y)'. The 'used_rules' looks for the rules that were used in the deduction sequence, with an 'ancestors' predicate, and gathers them in a list. The 'suspected_rule' chooses one rule in the list, beginning with the first one. The 'no_more_deducible' looks for another deduction sequence involving the suspected rule, builds the corresponding exam-

ple and asks the user whether the assertion is also false. If it is, the suspected rule is the guilty one and it will be deleted; otherwise, the suspected rule is considered to be correct, and the next rule is tried. The catch-all rule is needed because within this rule we can look at the deduction sequence, memorized by the Prolog interpreter.

Problem 93 [Brazdil 1984]

Verbal statement:

One of the common errors in learning is overgeneralization: the given term Q is applicable in certain contexts instead of failing. The purpose of the following program is to correct this. Two contexts need to be supplied: an application context A in which the proof of Q should succeed, and a rejection context R in which the proof of Q should fail. All clauses determining what Q is and how it is related to the contexts A and R should also be given. The expression generated which is applicable in the application context only is referred to as the discriminant, and the process of generating the discriminant as discrimination.

Logic program:

```
/* Operators */
:- op(150,yfx,:).
:- op(145,xfx,<-).
:- op(140,xfy,&).
:- op(135,xfx,:=).

/* For example 1 */

ex1(Disc):- exc1,
    der(q<-u&v,app),      /* Generate derivation(s) Da */
    der(q<-w&v,rej),       /* Generate derivation(s) Dr */
    disc(q,Disc,Disr).      /* Generate the discriminant(s) */

exc1:- retractall(_:_),
    assertz(c1:q<-s&r),   /* Domain specific knowledge used */
    assertz(c2:s<-t),       /* in this example */
    assertz(c3:s<-w),
    assertz(c4:t<-u),
    assertz(c5:r<-v).

/* For example 2 */

ex2(Disc):- exc2,
der(term(t1:t2)<-const(t1)&const(t2),app),
    /* Generate derivation(s) Da */
der(term(t1:t2)<-termv(t1)&termc(t2),rej),
    /* Generate derivation(s) Dr */
disc(term(t1:t2),Disc,Disr),
    /* Generate discriminant(s) */
```

```

exc2:- retractall(_:_),      /* Domain specific knowledge */
       assertz(c1:term(X)<-(X1:X2):=X&term(X1)&term(X2)),
       assertz(c2:term(X)<-termc(X)),
       assertz(c3:term(X)<-termv(X)),
       assertz(c4:termc(X)<-(X1:X2):=X&termc(X1)&termc(X2)),
       assertz(c5:termc(X)<- const(X)),
       assertz(c6: termv(X) <- (X1:X2):=X&termv(X1)),
       assertz(c7: termv(X) <- (X1:X2):=X&termv(X2)),
       assertz(c8: termv(X) <- var(X)),
       assertz(c9: X1:=X2 <- true):- X1:=X2).

/* Generation of derivation trees */

/* The generation of derivation trees is accomplished by */
/* 'der(P1<-C,Typ)' where 'P1<-C' is the expression whose */
/* truth/falsity is to be established, and 'Typ' is the */
/* type of the context (app/rej). */

der(P1<-C,Typ):-
    name(a1,Name),
    addcontx(Name,C),
    idgoals(P,Id1,1,I1),
    derx(P1,P2,Id1,Id2,I1:I1,I2:Der),
    assertz(Typ:Der), write(Typ:Der), nl, fail.

der(_,_):-
    name(a1,[N1,N2]),
    delcontx([N1,_]). 

/* Addition and deletion of context */

/* The addition and the deletion of contexts is accomplished */
/* by 'addcontx([N1,N2],P)' where '[N1,N2]' is the clause */
/* name represented as a list of two characters, and 'P' */
/* is the conjunction of predicates to be asserted. */

addcontx([N1,N2],P1&P2):- !,
    name(C,[N1,N2]),
    assertz(C:P1<-true),
    N3 is N2+1,
    addcontx([N1,N3],P2).

addcontx([N1,N2],P1):-
    name(C,[N1,N2]),
    assertz(C:P1<-true).

delcontx([N1,N2]):-
    C:P1<-true,
    name(C,[N1,_]),
    retract(C:P1<-true),
    fail.

delcontx(_).

```

```

/* Generation of goal identifiers */

/* The generation of goal identifiers is accomplished      */
/* by 'idgoals(P,Is,I1,I4)' where 'P' is the given      */
/* conjunction of goals, 'Is' is the conjunction          */
/* of goal identifiers to be generated, 'I1' is the      */
/* last identifier to be used, and 'I4' is the new value */
/* of the last identifier.                                */

idgoals(P1&P2, I1&I2, I1, I4):- !,
    I3 is I1+1,
    idgoals(P2, I2, I3, I4).

idgoals(P1, I1, I1, I4):- !,
    I4 is I1+1.

/* Expansion of derivation trees */

/* The expansion of the derivation trees is done          */
/* by 'derx(P1,P2,Id1,Id2,D1,D2)' where 'P1' is the     */
/* expression whose truth/falsity is to be established,   */
/* 'P2' is the expression obtained from P1, 'Id1' and    */
/* 'Id2' are conjunctions of goal identifiers and 'D1'   */
/* and 'D2' are derivations.                            */

derx(true,true,Id1,Id1,D1,D1):-!.
derx(true&P3,P3,Id1&Id3,Id3,D1,D1):- !.
derx(P1&P3,P5,Id1&Id3,Id5,D1,D3):-
    derx1(P1,P2,Id1,Id2,D1,D2),
    joingoals(P2&P3,P4,Id2&Id3,Id4),
    derx(P4,P5,Id4,Id5,D2,D3).

derx(P1,P5,Id1,Id5,D1,D3):-
    not(P1=_&_),
    derx(P1,P2,Id1,Id4,D1,D2),
    derx(P2,P5,Id4,Id5,D2,D3).

derx1(P1,P2,Id1,Id2,I1:D1,I2:D2):-
    C:P1<-P2,
    idgoals(P2,Id2,I1,I2),
    D2=(D1:(C:Id1<-Id2)).

/* Joining goals */

/* The joining of goals is done by 'joingoals(P1,P2,Id1,Id2)', */
/* where 'P1' and 'P2' are goals, and 'Id1' and 'Id2' are goal */
/* identifiers.                                              */

joingoals((P1&P2)&P3,P1&P5,(Id1&Id3)&Id3,Id1&Id5):- !,
    joingoals(P2&P3,P5,Id2&Id3,Id5).

joingoals(true&P3,P3,Id1&Id3,Id3):- !.
joingoals(P1,P1,Id1,Id1).

```

```

/* Predicate T1:=T2 */

T1:=T2 :- not(not(T1=T2)), not(not(spec(T1, T2)), T1=T2,
spec(T1, T2):-ground(T2, 1,_), T1=T2.

ground(qq(N1), N1, N2):- !, N2 is N1+1.
ground(T, N1, N2):- T=..[_|Ts], Ts==[], !.
ground(T, N1, N2):- T=..[_|Ts], grounds(Ts, N1, N2).

grounds([T|Ts], N1, N3):- ground(T, N1, N2),
                           grounds(Ts, N2, N3).
grounds([], N1, N1).

/* Comparison of derivation trees */

/* The comparison of derivation trees is done by      */
/* 'disc(P,Pa,Pr)' where 'P' is the expression to      */
/* be specialized and 'Pa' and 'Pr' are the expressions */
/* obtained.                                         */

disc(P,Pa,Pr):-
    idgoals(P,Id,1,_),
    disc(P:Id:P:Id,Pa:Ia:Pr:Ir), assertz(disc:Pa),
    write(disc:Pa), nl, fail.
disc(P,Pa,Pr).

discr(Pa1:Ia1:Pr1:Ir1,P3):- !,
    not(Pa1=Pr1),
    P3=(Pa1:Ia1:Pr1:Ir1).

discr((true&Pa3):(_&I3):(true&Pr3):(_&Ir3),P3):- !,
    P3=(Pa3:Ia3:Pr3:Ir3).

discr((Pa1&Pa3):(Ia1&Ia3):(Pr1&Pr3):(Ir1&Ir3),P3):-
    discr(Pa1:Ia1:Pr1:Ir1,Pa2:Ia2:Pr2:Ir2),
    joingoals(Pa2&Pa3,Pa5,Ia2&Ia3,Ia5),
    joingoals(Pr2&Pr3,Pr5,Ir2&Ir3,Ir5),
    P3=(Pa5:Ia5:Pr5:Ir5).

discr((Pa1&Pa3):(Ia1&Ia3):(Pr1:Pr3):(Ir1&Ir3)),P3):-
    discr(Pa3:Ia3:Pr3:Ir3,Pa4:Ia4:Pr4:Ir4),
    P3=(Pa1&Pa4):(Ia1:Ia4):(Pr1&Pr4):(Ir1&Ir4)).

discr(P1,P3):-
    not(P1=(_&:_&:_&:_&_)),
    discr1(P1,P2),
    discr(P2,P3).

discr(P1,P3):-
    P1=(Pa1:Ia1:Pr1:Ir1),
    Ca:Pa1<-Pa2,
    app:Da,
    inder(Ca:Ia1<-Ia2, Da)

```

```

Cr:Pr1<-Pr2,
rej:Dr,
inder(Cr:Ir1<-Ir2,Dr),
P3=(Pa2:Ir2:Pr2:Ir2).

/* Searching through a derivation */
inder(C,Der:C):- !.
inder(C,Der:_):- inder(C,Der).

```

Execution:

For example 1:

```
?- ex1(D).
```

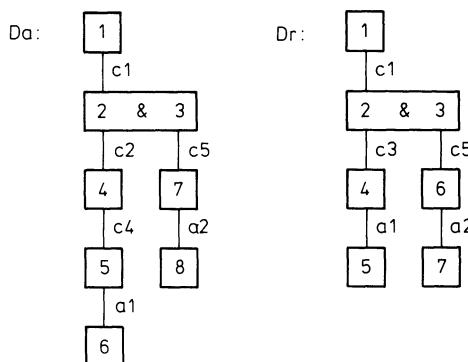


Fig. 1

Discriminant:

t & r

For example 2:

```
?- ex2(D).
```

Derivation trees Da1 & Da2

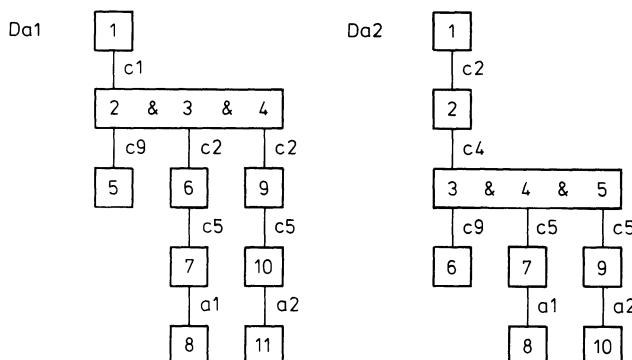


Fig. 2

Derivation trees Dr1 & Dr2

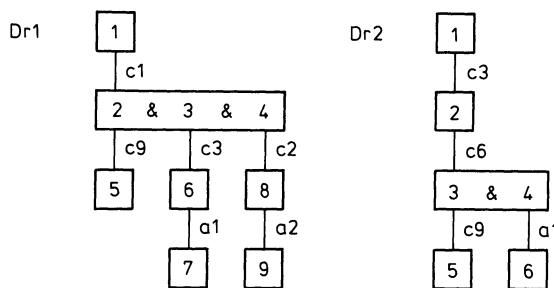


Fig.3

Discriminants:

D1 X1:X2:=t1:t2 & termc(X1) & term(X2)	Da1 x Dr1
D2 X1:X2:=t1:t2 & term(X1) & term(X2)	Da1 x Dr2
D3 term(t1:t2)	Da2 x Dr1
D4 termc(t1:t2)	Da2 x Dr2

Comments:

The required discriminant is formal on the basis of comparison of two kinds of derivation trees:

- (1) the derivation tree Da associated with the proof of $Q <- A$, and
- (2) the derivation tree DR associated with the proof of $Q <- R$.

The derivation trees mentioned will be generated and stored (using assert) using `der(Q <- A,app)` and `der(Q <- R,rej)` respectively. The constant 'app' indicates that 'A' is an application context, and 'rej' indicates that 'R' is a rejection context. Call `disc(Q,Disc, _)` will initiate the search for the right discriminant on the basis of the derivation trees stored.

In example 2 more than one discriminant is obtained due to the fact that `der(Q <- A,app)` has more than one solution. The corresponding derivation trees are referred to as Da1 and Da2. The predicate `der(Q <- R, rej)` has also two solutions Dr1 and Dr2, but the expression generated should be specific enough so that it would fail in all rejection contexts. As we can see, only discriminants D1 and D4 satisfy this criteria.

Chapter 10 Modeling with Prolog

A representation is a set of conventions for describing objects. Experience has shown that the choice of a suitable formalism is the key to turning hard problems into simple ones.

The value of an abstract representation (such as semantic nets or logic formalisms) is to separate and clarify the modeling aspects from computation oriented considerations. One of the consequent challenges to programming language design is to provide an implementation framework which maintains this clarity and focus as far as possible. This chapter concentrates on problems that enhance Prolog as a tool for logical domain modeling.

Problem 94

Verbal statement:

In a group of women, some are supposed to be beautiful, other just female. Describe the knowledge behind this scenario, in order to query about the attributes of these women.

Suppose that every blond-haired woman with brown eyes is beautiful (a latin criterion of beauty . . .).

Logic program:

```
woman(W):- human(W), female(W).  
beautiful(B):- blond(B), brown_eyes(B).  
  
human(helder).      female(susan).      blonde(susan).  
human(john).        female(alice).       blonde(elisabeth).  
human(susan).        female(kathy).       .  
human(alice).        female(mary).        brown_eyes(alice).  
human(kathy).        female(elisabeth).   brown_eyes(elisabeth).  
human(mary).         .  
human(elisabeth).    .
```

Execution:

```
?- woman(X), beautiful(X).  
X=elisabeth
```

Problem 95 [Warren 1981 c]

Verbal statement:

In a pub regulars drink beer. But there are exceptions. Some fellows don't like alcoholic drinks and are used to milk. How many are they?

Logic program:

```
:- op(500,yfx,'drinks').
john      drinks      whisky.
martin    drinks      whisky.
david     drinks      milk.
ben       drinks      milk.
helder    drinks      beer.
laurence  drinks      beer.
chris     drinks      beer.
carlos    drinks      brandy.
```

Execution:

```
?- setof(X,X drinks milk,S),length(S,N).
S=[ben,david], N=2.
```

Comments:

The built-in predicate ‘length’ computes the dimension of a list of objects. The extra facility ‘setof’ provided by Prolog supports answers for the following question: “is it true of any S and N that the set of those that drink milk is S and the size of the set S is N?”

Problem 96

Verbal statement:

In a newspaper was published the following plot: “Helder, a poor scientist, was in love with the daughter of an admiral. One day, a general captured the girl. Helder rode to the general’s barrack and killed the general. The girl was very grateful and fell in love with Helder. The admiral was so happy to have his daughter back he gave Helder half of his boats”. After listening the plot, a child asked some questions, such as:

“Who is the father of the girl?”, “Who is rich?”, “Who loves who?”, “Who is poor?”, “Who captured who?”, “Who killed who?”, and “Who lived happily ever after?”.

Write a model for the plot and ask the child to rephrase in Prolog his questions.

Logic program 1:

```

:- op(500,xfy,'is').
:- op(500,yfx,'captures').
:- op(500,yfx,'rides_to').
:- op(500,yfx,'gives').
:- op(500,yfx,'is_father_of').

helder is poor.
helder is scientist.
admiral is happy.
admiral is_father_of girl.
helder loves girl.
girl loves holder.
general captures girl.
helder kills general.
admiral gives half_boats to holder.

```

Execution:

```

?- X is poor.
X=helder

?- Z loves Y.
Z=helder
Y=helder;
Z=girl
Y=helder;
no

?- W captures Y.
W=general
Y=girl

?- Z kill W.
Z=helder
W=general

```

Logic program 2:

```

poor(helder).
scientist(helder).
father(admiral,girl).
happy(admiral)
loves (helder,girl).
captures(general,girl).
kills(helder,general)
gives(admiral,holder,half_boats).

not(X):- X,! ,fail.
not(_).

```

Execution:

```
?- not(poor(X)).  
?- father(X,Y).  
X=admiral  
Y=girl
```

Comments:

The story summary is a collection of database assertions: “Helder is poor”, “Helder is a scientist”, “Helder loves the girl”, “the general captures the girl”, “Helder rides to the general’s barrack”, “Helder kill the general”, “the girl loves Helder”, “the admiral is happy” and “the admiral gives Helder half his boats”.

Problem 97

Verbal statement:

Consider a fact deduction machine which takes a collection of known facts and makes new conclusions, over a domain of animals in a zoo.

Suppose the machine knowledge is composed of 15 recognition productions:

```
P1  
If the animal has hair,  
then it is a mammal.  
  
P2  
If the animal gives milk  
then it is a mammal.  
  
P3  
If the animal has feathers  
then it is a bird.  
  
P4  
If the animal flies,  
and it lays eggs,  
then it is a bird.  
  
P5  
If the animal is a mammal,  
and it eats meat,  
then it is a carnivore.  
  
P6  
If the animal is a mammal,  
it has pointed teeth  
it has claws,  
and its eyes point forward,  
then it is a carnivore.
```

P7

If the animal is a mammal,
and it has hoofs,
then it is an ungulate.

P8

If the animal is a mammal,
and it chews cud,
then it is an ungulate,
and it is even toed.

P9

If the animal is a carnivore,
it has a tawny color,
and it has dark spots,
then it is a cheetah.

P10

If the animal is a carnivore,
it has a tawny color,
and it has black stripes,
then it is a tiger.

P11

If the animal is an ungulate,
it has long legs and a long neck,
it has a tawny color,
and it has dark spots,
then it is a giraffe.

P12

If the animal is an ungulate,
it has a white color,
and it has black stripes,
then it is a zebra.

P13

If the animal is a bird,
it does not fly,
it has long legs and a long neck,
and it is black and white,
then it is an ostrich.

P14

If the animal is a bird,
it does not fly,
it swims,
and it is black and white,
then it is a penguin.

P15
 If the animal is a bird
 and it is a good flyer
 then it is an albatross.

Write a program (simulating that machine) able to discover a certain animal by means of asking its characteristics using the recognition productions, and to describe the properties of any animal in its knowledge base.

(This problem was suggested from “Artificial Intelligence” by Patrick Winston).

Logic program:

```
/* Knowledge base */

rule(1,animal,mammal,[c1]).  

rule(2,animal,mammal,[c2]).  

rule(3,animal,bird,[c3]).  

rule(4,animal,bird,[c4,c5]).  

rule(5,mammal,carnivore,[c6]).  

rule(6,mammal,carnivore,[c7,c8,c9]).  

rule(7,mammal,ungulate,[c10]).  

rule(8,mammal,ungulate_toed,[c11]).  

rule(9,carnivore,cheetah,[c12,c13]).  

rule(10,carnivore,tiger,[c12,c14]).  

rule(11,ungulate,giraffe,[c15,c16,c12,c13]).  

rule(12,ungulate,zebra,[c17,c14]).  

rule(13,bird,ostrich,[c18,c15,c16,c19]).  

rule(14,bird,penguin,[c18,c20,c19]).  

rule(15,bird,albatross,[c21]).  

/* Recognition process : discover animal's name */

recognition(X):- rule(N,X,Y,Z),discover(Z),found(rule),
               conclusion(X,Y,N), recognition(Y),
               retractall(fact(_,_)).  

recognition(_):- retract(rule),write('Done.'),nl.  

recognition(_):- write('Don''t know this animal.'),nl.  

found(X):- X,!.  

found(X):- assert(X).  

/* Discovering process */

discover([]).  

discover([X|Y]):- ask(X),discover(Y).  

ask(X):- fact(X,yes),!.  

ask(X):- fact(X,no),!,fail.  

ask(c1):- write('has it hair?')nl,! ,complete(c1).  

ask(c2):- write('does it give milk?')nl,! ,complete(c2).  

ask(c3):- write('has it feathers?')nl,! ,complete(c3).
```

```

ask(c4):- write('does it fly?'),nl,! ,complete(c4).
ask(c5):- write('does it lay eggs?'),nl,! ,complete(c5).
ask(c6):- write('does it eat meat?'),nl,! ,complete(c6).
ask(c7):- write('has it pointed teeth?'),nl,! ,complete(c7).
ask(c8):- write('has it claws?'),nl,! ,complete(c8).
ask(c9):- write('has it eyes pointing forward?'),
    nl,! ,complete(c9).
ask(c10):- write('has it hoofs?'),nl,! ,complete(c10).
ask(c11):- write('does it chew cud?'),nl,! ,complete(c11).
ask(c12):- write('has it a tawny color?'),nl,! ,complete(c12).
ask(c13):- write('has it dark spots?'),nl,! ,complete(c13).
ask(c14):- write('has it black stripes?'),nl,! ,complete(c14).
ask(c15):- write('has it long legs?'),nl,! ,complete(c15).
ask(c16):- write('has it a long neck?'),nl,! ,complete(c16).
ask(c17):- write('has it a white color?'),nl,! ,complete(c17).
ask(c18):- write('does it not fly?'),nl,! ,complete(c18).
ask(c19):- write('is it black and white?'),nl,! ,complete(c19).
ask(c20):- write('does it swim?'),nl,! ,complete(c20).
ask(c21):- write('is it a good flyer?'),nl,! ,complete(c21).

complete(X):- read(Y),assert(fact(X,Y)),Y=yes.

/* Conclusion of the recognition process */

conclusion(X,Y,N):- nl,tab(4),write('--- the '),write(X),
                  write(' is a '),write(Y),write(' by rule '),
                  write(N),nl,nl.

/* Description process: discover animal's properties */

description(X):- rule(N,Y,X,Z),description1(Y,L,[ ]),
                conclusion1(X,L,Y,Z,N).
description(_):- nl,write('Don''t know this animal.'),nl.

description1(Y,L,Ls):- rule(_,X,Y,_),description1(X,L,[X|Ls]).
description1(_,L,L).

/* Conclusions of the description process */

conclusion1(X,L,Y,Z,N):- nl,write('a '),write(X),
                        write(' is an '),
                        output(L),write(Y),
                        write('satisfying conditions: '),
                        nl,
                        output(Z),nl,write('by rule '),
                        write(N),write('.'). 

output([]).
output([A|B]):- write(A),tab(1),output(B).

```

Execution:

When discovering the animal's name

1) Example of a successful recognition

?-recognition(animal).

has it hair?

| yes.

---the animal is a mammal by rule 1

does it eat meat?

| yes.

---the mammal is a carnivore by rule 5

has it a tawny color?

| yes.

has it dark spots?

| no.

has it black stripes?

| yes.

---the carnivore is a tiger by rule 10

Done.

2) Example of a unsuccessful recognition

?-recognition(animal).

has it hair?

| no.

does it give milk?

| no.

has it feathers?

| no.

does it fly?

| no.

Don't know this animal.

When discovering the animal's properties

1) Example of a successful description

?- description(tiger).

A tiger is an animal mammal carnivore satisfying conditions:

c12 c14

by rule 10

2) Example of a unsuccessful description

?- description(horse).

Don't know this animal.

Comments:

The program answers only with the numbers of the conditions for convenience of the programmer. A procedure may be written to translate these numbers into the sentences of the conditions.

Problem 98

Verbal statement:

Model the world of cooking recipes and write a question-answering program to be used at home.

(Suggestion: adopt the model of a recipe book).

Logic program:

```
/*-----*/
/*          Cooking Recipes           */
/*-----*/
:- op(700, xfy, '^').
/*      COMMUNICATION WITH THE USER      */
/* How recipe X is manufactured */
manufacture(T):-
    recipe(T,L), write_nl('ingredients:'),
    write_list(L), ingr_med(L,U), exec(U), nl, write('OK.').

ingr_med([[_,_]], [H]). 
ingr_med([[_,_]|L], [H|T]):- ingr_med(L,T).

exec([_,_,_]):- omsimp, display_nl(roll(result)), !.
exec([_,_,_,rasped_cheese]):-
    omsimp, display_list([grasp(cheese,result,result),
    roll(result)]), !.
exec([_,_,_|T]):-
    T=[onion,parsley], cut1(T), display_nl(shake(eggs)),
    mixture([salt,parsley],eggs,result),
    stew(result),
    mixture(result,result,result),
    display_list([warm(result,2),roll(result)]), !.
exec([eggs,margarine,salt|L]):-
    cut1(L), bas, mixture(L,result,result),
    display_nl(warm(result,1)),
    mixture(eggs,result,result),
    display_list([warm(result,2),roll(result)]), !.
```

```

exec([sugar_beet|_]):-  

    boil1([sugar_beet,eggs]), cut1([sugar_beet,onion]),  

    mixture([onion,salt,pepper,oil,vinager,sugar],  

            sugar_beet,result),  

    display_list([cut(eggs),go_on(result,eggs)]), !.  

exec([_,_,bacon|_]):-  

    display_list([wash(lettuce),peel(carrot)]),  

    cut1([lettuce,carrot,bacon]),  

    mixture(lettuce,carrot,result),  

    display_list([grasp(bacon,oil,result),  

                warm(result,2)]),  

    mixture([bacon,salt,pepper],result,result), !.  

exec([A,B,C|T]):-  

    wash1([A,B]), cut1([A,B,C]),  

    mixture_all([A,B,C|T],result).  

omsimp:- bas, mixture(result,margarine,result),  

        display_nl(warm(result,2)).  

bas:- display_nl(shake(eggs)), mixture(salt,eggs,result),  

      display_nl(warm(margarine,1)).  

stew(X):- mixture(onion,margarine,X), display_nl(warm(X,2)).  

/* Find all recipes of a certain category */  

find_all(X):-  

    (call(recipe([X|T],L)),  

     write_nl([X|T]), write_nl('ingredients:'),  

     write_list(L), fail); true.  

/* If ingredient X is not available what recipes can I not do? */  

not_do(X):- recipe(T,L), ingredient_med(L,U), ((member(X,U),  

                                                write('you can not do'), write_nl(T));  

                                                not(member(X,U))), fail.  

/* When ingredients I1,I2,...,In are not available is it  

   possible to do recipe L? */  

can_do(L,T):-  

    (recipe(L,M), ingr_med(M,U), intersection(T,U,V),  

     are_the_same(U,V), write('yes.'));  

     write('no.').  

are_the_same([X|L],T):- member(X,T), are_the_same(L,T),  

are_the_same([],_).  

/* Which are the basic ingredients for a certain type of  

   recipes (omelet, salads, etc)? */
```

```

ingr_bas(X):-
    ((recipe([X|_],T), ingr_med(T,U),
    nomap(U), call(map0(L), intersection(U,L,M),
    retract(map0(L)), asserta(map0(M)), fail); (true,
    write('the basic elements for '), write(X),
    write_nl('are:'), call(map0(M)), write(M)),
    retract(map0(M))).

nomap(T):- not(map0(_)), asserta(map0(T)), !.
nomap(_).

/* Which are the ingredients of recipe X? */

ingredients(X):-
    recipe(X,T),
    write('the ingredients for the recipe '), write(X),
    write_nl(' are:'), write_list(T).

/* Is there recipe X? */

exists(L):- recipe(L,_), write('yes.'), !.
exists(L):- not(recipe(L,_)), write('no.').

/* Increase the quantities of the recipe */

increase(M,L):-
    recipe(L,T), multiply(M,T,U),
    write('increase ingredients '), write(M),
    write_nl(' times:'), write_list(U).

multiply(M,[],[]).
multiply(M,[[X,Y^U]|T],[[X,Z^U]|L]):-
    Z is M*X, multiply(M,T,L), !.
multiply(M,[[X,qb]|T],[[X,qb]|L]):-
    multiply(M,T,L), !.
multiply(M,[X,Y]|T),[[X,Z]|L]):-
    Z is M*X, multiply(M,T,L), !.

/* PRINTING ROUTINES */

write_list([]):- nl.
write_list([H|T]):- write_nl(H), write_list(T).

write_nl(M):- write(M), nl.

display_list([]):- nl.
display_list([H|T]):- display_ml(H), display_list(T).

display_ml(H):- display(H), nl.

/* UTILITIES */

intersection([],X,[]).
intersection([X|R],Y,[X|Z]):- member(X,Y), !,
    intersection(R,Y,Z).

```

```

intersection([X|R],Y,Z) :- not(member(X,Y)), !,
    intersection(R,Y,Z).

member(H,[H|_]). 
member(H,[_|T]) :- member(H,T).

not(P) :- P, !, fail.
not(_).

/*          DATA HANDLING          */
mixture(X,Y,Z) :- grasp1(X,Y,Z), display_nl(shake(Z)).

mixture_all([X,Y|L],Z) :- 
    display_nl(grasp(X,Y,result)),
    mixture(L,result,Z).

cut1([H]) :- display_nl(cut(H)), !.
cut1([H|L]) :- display_nl(cut(H)), cut1(L).

grasp1(X,Y,Z) :- atom(X), display_nl(grasp(X,Y,Z)), !.
grasp1([H],Y,Z) :- display_nl(grasp(H,Y,Z)), !.
grasp1([H|L],Y,Z) :- 
    display_nl(grasp(H,Y,result)),
    grasp1(L,result,Z).

boil1([H]) :- display_nl(boil(H)), !.
boil1([H|L]) :- display_nl(boil(H)), boil1(L).

wash1([H]) :- display_nl(wash(H)), !.
wash1([H|L]) :- display_nl(wash(H)), wash1(L).

/*          KNOWLEDGE BASE          */
/* Recipes in the database */

recipes([omelet,simple],[[eggs,3],[margarine,1*s_soup],
           [salt,1*s_tea]]).
recipe([omelet,of,cheese],[[eggs,3],[margarine,1*s_soup],
           [salt,1*s_tea],[rasped_cheese,2*s_soup]]).
recipe([omelet,of,potatoes, and,ham],[[eggs,3],
           [margarine,1*s_soup],
           [salt,1*s_tea],[ham,2*slices],
           [baked_or_fried_potatoes,qb]]).
recipe([omelet,of,parsley, and,onion],[[eggs,3],
           [margarine,1*s_soup],
           [salt,1*s_tea],[onion,1],[parsley,qb]]).
recipe([omelet,of,bacon],[[eggs,3],[margarine,1*s_soup],
           [salt,1*s_tea],[bacon,2*slices]]).
recipe([salad,of,lettuce, and,tomato],[[lettuce,1],[tomato,2],
           [onion,1],[oil,qb],[salt,qb],[vinegar,qb]]).
recipe([salad,of,sugar_beet],[[sugar_beet 750*gr],[onion,1],
           [oil,qb],[vinegar,qb],[salt,qb],[pepper,qb],
           [sugar,1*s_tea],[eggs,2]]).

```

```

recipe([salad,with,bacon],[[lettuce,1],[carrot,1],
    [bacon,120^gr],[oil,2^s_soup],[salt,qb],[pepper,qb]]).

/* Atomic actions */

warm(X,H).           /* warm X during H minutes */

boil(X).

cut(X).

peel(X).

roll(X).

decorate_with(X,Y).   /* decorate X with Y */

grasp(X,Y,Z).         /* grasp X to Y and get Z */

wash(X).

shake(X).

rasp(X).

```

Execution:

```

| ?- manufacture([omelet,simple]).  

ingredients:  

[eggs,3]  

[margarine,1^s_soup]  

[salt,1^s_tea]  

shake(eggs)  

grasp(salt,eggs,result)  

shake(result)  

warm(margarine,1)  

grasp(result,margarine,result)  

shake(result)  

warm(result,2)  

roll(result)  

OK.  

yes  

| ?- manufacture([omelet,of,cheese]).  

ingredients:  

[eggs,3]  

[margarine,1^s_soup]  

[salt,1^s_tea]  

[rasped_cheese,2^s_soup]  

shake(eggs)  

grasp(salt,eggs,result)  

shake(result)

```

```
warm(margarine,1)
grasp(result,margarine,result)
shake(result)
warm(result,2)
grasp(cheese,result,result)
roll(result)

OK.

yes
| ?- manufacture([omelet,of,potatoes, and,ham]).  
ingredients:  
[eggs,3]  
[margarine,1^s_soup]  
[salt,1^s_tea]  
[ham,2^slices]  
[baked_or_fried_potatoes,qb]

cut(ham)
cut(baked_or_fried_potatoes)
shake(eggs)
grasp(salt,eggs,result)
shake(result)
warm(margarine,1)
grasp(ham,result,result)
grasp(baked_or_fried_potatoes,result,result)
shake(result)
warm(result,1)
grasp(eggs,result,result)
shake(result)
warm(result,2)
roll(result)

OK.

yes
| ?- manufacture([salad,of,lettuce, and,tomato]).  
ingredients:  
[lettuce,1]
[tomato,2]
[onion,1]
[oil,qb]
[salt,qb]
[vinegar,qb]

wash(lettuce)
wash(tomato)
cut(lettuce)
cut(tomato)
cut(onion)
grasp(lettuce,tomato,result)
```

```

grasp(onion,result,result)
grasp(oil,result,result)
grasp(salt,result,result)
grasp(vinegar,result,result)
shake(result)

```

OK.

yes

Comments:

The program is able to answer the following questions:

- A) Manufacture some recipe: ‘manufacture([omelet,of,bacon])’
- B) Find all recipes of a certain type: ‘find – all(salad)’
- C) Show the impossible recipes on account of the lack of some ingredient: ‘not – do(bacon)’
- D) List the possible recipes with the available ingredients: ‘can – do([salad,with,bacon],[salt,oil,bacon])’
- E) Know the basic ingredients for some recipe: ‘ingr – bas(omelet)’
- F) Know the ingredients and their quantities for a recipe: ‘ingredients([salad,of,sugar – beet])’
- G) Know whether a recipe is known: ‘exists([cake,of,chocolate])’
- H) Increase n times some recipe: ‘increase(3,[salad,of,lettuce, and,tomato])’

The available ground knowledge is organized around the available facts, i.e. the atomic actions such as ‘wash(X)’. Each recipe is represented by a binary relation where each argument is a list: the first one is the name of the recipe, and the second is the set of ingredients. The deep knowledge is attached to frames and it is embedded in the clauses, such as ‘manufacture’. Other clauses, such as ‘cut1(L)’ or ‘grasp1(X,Y,Z)’, aggregate repetitive basic actions over the ingredients contained in the list ‘L’. The search attached to the execution of each recipe is done by the clause ‘exec(T)’ over the list ‘T’ of ingredients.

Chapter 11 Designing with Prolog

The purpose of this chapter is to exhibit some representative exercises of knowledge engineering. The first exercise has to do with civil engineering, the second deals with architecture, the third with pharmacology, and the fourth with sequences of numbers. Behind all the exercises there is a combinatorial problem.

Problem 99

Verbal statement:

An engineering department is in charge of removing 5 cubic meters of garbage per day, but there are only two means: a digging machine and manual workers. It is known that the machine removes 4 cubic meters per day and one worker 1 cubic meter per day. Which are the alternative solutions faced by that department?

Logic program:

```
remove(0, []).
remove(X, [Y]) :- means(Y, Z), X = < Z.
remove(X, [Y|L]) :- means(Y, Z), T is X - Z, T > 0,
    remove(T, L).

means(digging_machine, 4).
means(worker, 1).
```

Execution:

```
?- remove(5, L).
L=[digging_machine, digging_machine];
L=[digging_machine, worker];
L=[worker, digging_machine];
L=[worker, worker, digging_machine];
L=[worker, worker, worker, digging_machine];
L=[worker, worker, worker, worker, digging_machine];
L=[worker, worker, worker, worker, worker];
no
```

Comments:

There are various combinations of machine and labor force with which to carry out this task. The program has no rules for recognizing those likely to be more economical than others, and the equal combinations.

This example may also be viewed as a simple planning case.

Problem 100 [Markusz 1977]**Verbal statement:**

Write a program for designing an architectural unit obeying the following specifications:

- Two rectangular rooms.
- Each room has a window and interior door.
- Rooms are connected by interior door.
- One room also has an exterior door.
- A wall can have only one door or window.
- No window can face north.
- Windows cannot be on opposite sides of the unit.

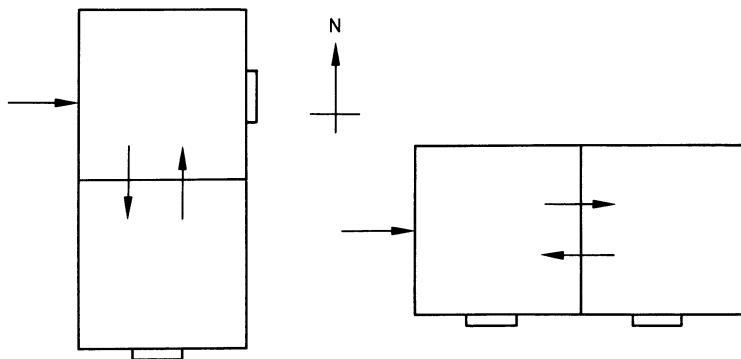


Fig.1. Sample Plans

Logic program:

```

plan(FD,D1,W1,D2,W2):- frontroom(FD,D1,W1),
    opposite(D1,D2),
    room(D2,W2),
    notopposite(W1,W2).

frontroom(FD,D,W):- room(D,W),
    direction(FD),
    FD=\=D, FD=\=W.

room(D,W):- direction(D),
    direction(W),
    D=\=W, W=\=north.

```

```

direction(north).
direction(south).
direction(east).
direction(west).

opposite(north,south).
opposite(south,north).
opposite(east,west).
opposite(west,east).

notopposite(D1,D2):- opposite(D1,D3),
D2=\=D3.
```

Execution:

Planning a unit entered from the west:

```
?-plan(west,D1,W1,D2,W2).

D1 = east,
W1 = south,
D2 = west,
W2 = south
```

Problem 101 [Darvas et al. 1978]

Verbal statement:

A drug never exerts its effect under the well-controlled conditions of primary screening of preclinical tests, since the activity spectrum and toxicity of the compound can be influenced unpredictably by the possible drug interactions, environmental factors, other diseases of the patient, etc. To predict the effect of these factors on the activity may be a very significant tool in the planning of clinical tests of a drug to be released or in the design of control tests of drugs already in use.

Write a program for predicting drug interactions.

Logic program:

```
/* Drug properties */

characteristic(barbamid,amidopiridin,pirezolone).
characteristic(barbamid,barbitalsodium,barbiturate).
characteristic(tromexan,pelentan,coumarin).
characteristic(seconal,recobarbital,barbiturate).
characteristic(peritol,periactin,base).

active_characteristic(tromexan,pelentan,haemostat,circulation).
active_characteristic(seconal,secobarbital,hypnot,cns).
active_characteristic(barbamid,amidopirin,antipyret,cns).
```

```

active_characteristic(barbamid,barbitalsodium,hypnot,cns).
active_characteristic(peritol,periactin,antihistamine,cns).

/* Facts about chemistry */

contains(periactin,amino_group).
contains(procaine,amino_group).

base(X):- contains(X,amino_group).

/* Data on varions drugs */

ingredient(aspirin,salicylic_acid).
ingredient(whisky,ethanol).

/* General rules about drug interactions */

interaction(D1,F1,acty_decr,D2,F2):-
    active_characteristic(D1,F1,haemostat,U),
    characteristic(D2,F2,barbiturate).

interaction(Drug1,Drug2,
    increased_absorption_by_ion_pairFormation(Agent1,Agent2)):-  

    ingredient(Drug1,Agent1),
    ingredient(Drug2,Agent2),
    quaternary_ammonium_salt(Agent1),
    anti_arrhythmic(Agent1),
    salicylate(Agent2).

increase_absorption(K1,F1,ion_pairFormation,K2,F2):-
    quaternary_ammonium_salt(K1,F1),
    anti_arrhythmic(K1,F1),
    salicylate(K2,F2).

decrease(K1,F1,plasma_half_life,K2,F2):-
    preparation(K1,F1),
    F1=dexamethasone,
    preparation(K2,F2),
    F2=phenobarbital,
    patient(asthmatic).

/* Database */

preparation(tromexan,pelentan;coumarin+pelentan;haemostat;
    circulation!p.a.! 1s).
preparation(second,secobarbital;hypnot;cus!p.o!1).
preparation(barbanid,amidopirin; pirazolone+amidopirin;
    antipyret;cus!
    barbitalsodium;barbiturate+barbitalsodium;hyprot;
    cus!p.o!S8).
preparation(peritol,periactin;base+periaction;antihistamine;
    cus!p.o!60).

```

Execution:

Find the interaction of pelagent and recobarbital:

```
?- interaction(D1,pelentan,X,D2,recobarbital).  
D1=tromexan  
X=acty_decr  
D2=seconal
```

Find the interaction between aspirin and whisky:

```
?- interaction(aspirin,whisky,What).
```

Problem 102 [Baxter 1980]

Verbal statement:

Arrange 3 1's, 3 2's, ..., 3 9's in sequence so that for all i there are exactly i numbers between successive i 's.

Logic program:

Comments:

This is the so-called Lanford sequence.

Chapter 12 Drawing with Prolog

The use of knowledge brings the programmer to an understanding of concepts like feedback, recursion, state and so on. In this chapter the examples involve search through a large number of alternatives. This feature is especially well suited for problems that can be modelled as graph search ones. This is also the basis for an approach to planning and simulation models using logic programming (see Chapters 13 and 18). The purpose is also to support the programmer with graphics in order to put forward the underlying intelligent machinery and to show how programs reason.

Problem 103

Verbal statement:

Find a path from A to Z in the following directed graph (Fig. 1).

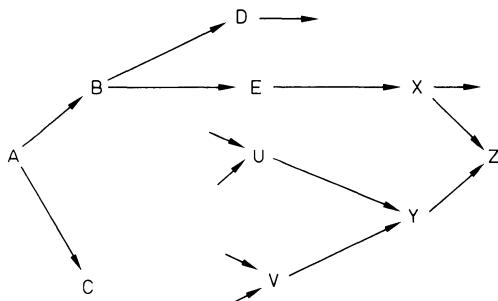


Fig. 1

Logic program:

```
go(X,Y):- go(X,Z), write('Path:'), nl, write((X,Z)),  
          go(Z,Y), write((Z,Y)), nl  
  
go(a,b).           go(x,z).  
go(a,c).           go(y,z).  
go(b,d).           go(u,y).  
go(b,e).           go(v,y).  
go(e,x).
```

Execution:

```
?- go(a,z).
```

Problem 104 [Emden 1976 e]

Verbal statement:

Let the interpretation I be

```
{G(A,B), G(A,C), G(A,D), G(B,F), G(C,F), G(E,F), G(F,C)}
```

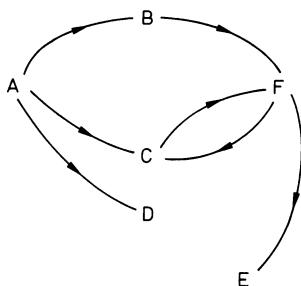


Fig. 2

Calculate the following queries:

- Q1=G(x₁,x₂) G(y₁,y₂)
- Q2=G(x,y) G(y,z)
- Q3=G(x,x)
- Q4= y.G(x,y)
- Q5= y.G(x,y) G(y,z)

Logic program:

```

q1(X1,X2,Y1,Y2):- g(X1,X2),g(Y1,Y2).
q2(X,Y,Z):- g(X,Y),g(Y,Z).
q3(X):- g(X,X).
q4(X):- g(X,_).
q5(X,Z):- g(X,Y),g(Y,Z).

g(a,b).   g(a,c).   g(a,d).
g(b,f).   g(c,f).   g(e,f).   g(f,c).

/* Computation of query Q1 */
answer1:- q1(X1,X2,X3,X4),
          write((X1,X2,X3,X4)),nl.
answer1.
  
```

Comments:

The clause ‘answer1’ implements the query Q1.

Problem 105 [Szeredi 1977]

Verbal statement:

Construct a program for drawing a picture (a graph), such as Fig. 3, using a continuous line, and without taking out the pencil from the surface of the sheet of paper.

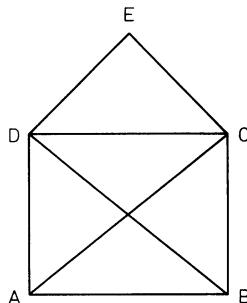


Fig. 3

Logic program:

```

:-op(600,xfy,'-').

draw(G,[P,Q|L]) :- choose(G,P-Q,G1),
                  draw(G1,[Q|L]). 

draw([],[]).

choose([P-Q|G],P-Q,G).
choose([P-Q|G],Q-P,G).
choose([E|G],F,[E|G1]) :- choose(G,F,G1).

```

Execution:

```

?-draw([a-b,a-c,a-d,b-c,b-d,c-d,c-e,d-e],L).
L=[a,b,c,d,e,c,a,d,b]

```

Comments:

The result is output on a line-printer or any other device (e.g. graphic display) by executing the output procedure specified for that device.

The relation ‘draw’ is responsible for choosing an arc and going on drawing by choosing another arc connected to the previous one. The relation ‘choose’ has three definitions. The first two allow arcs to be picked up, and the third one allows the graph ‘G’ to be reordered.

Problem 106

Verbal statement:

A graph generator is required able to:

- 1) generate all possible graphs with one-way edges from an initial given graph;
- 2) fix some nodes for each generated graph, i.e. erase the edges connecting to those nodes;
- 3) determine the end nodes;
- 4) determine which nodes (and how many) may attain the previous end nodes.

Logic program:

```

begin:- display('list of nodes ?'),nl,read(L),assert(nds(L)),
        display('list of edges ?'),nl,read(X),assert(ed(X)),
        display('list of relevant origin ?'),nl,read(Y)),
        assert(or(Y)),matrix.

matrix:- retract(ed(L)),compact(L,L1),calcul(L1),fail.
matrix:- display('Done.'),nl,nl,nl.

compact([],[]).
compact([m(X,Y)|B],R):- in(m(X,Y),B,D),!,
    (R=[m(X,Y)|C] ; R=[m(Y,X)|C]),
    compact(D,C).
compact([A|B],[A|C]) :- compact(B,C).

in(m(A,B),[m(B,A)|X],X).
in(m(A,B),[C|X],[C|R]) :- in(m(A,B),X,R).

calcul(L1) :- or(Y),in(A,Y),name(A,A1),
    X=[78,79,46|A1],name(X1,X),tell(X1),
    slash(out_of_list(A,L1,L2)),nds(L),
    slash(look(L,L2,L3)),calcul1(L2,L3).

calcul1(L2,L3) :- in(B,L3),
    test(B),complete(L2,B).

complete(L2,B) :- nds(L),orig(L2,B,L,O),length(O,No),put(" "),
    test(No),output(O),nl,!.

orig(L2,B,[N|Ns],[N|O]) :- path(N,B,L2).
orig(L2,B,[N|Ns],[N|O]) :- path(N,B,L2),
    orig(L2,B,Ns,O).
orig(L2,B,[_|Ns],O) :- orig(L2,B,Ns,O).
orig(_,_,[],[]).

slash(P) :- call(P),!.

path(A,B,L) :- in(m(A,B),L).
path(A,B,L) :- in(m(A,C),L),slash(out_of_list(C,L,Li)),
    path(C,B,Li).

```

```

out_of_list(A, [m(_,A)|X], Y):- out_of_list(A,X,Y).
out_of_list(A, [B|X], [B|Y]):- out_of_list(A,X,Y).
out_of_list(_,[],[]).

look([N|Ns],L2,Y):- in(m(N,_),L2),
    look(Ns,L2,Y).
look([N|Ns],L2,[N|Y]):- in(m(_ ,N),L2),
    look(Ns,L2,Y).
look([N|Ns],L2,Y):- look(Ns,L2,Y).
look([],_,[]).

in(A,[A|_]). 
in(A,[_|B]):- in(A,B).

output([A]):- put(" "),test(A).
output([A|B]):- put(" "),test(A),output(B).

test(A):- A<10,put(" "),write(A),!.
test(A):- write(A),!.

```

Problem 107

Verbal statement:

Write a program to test connections between nodes in a given graph. Apply your program to the graph in Fig.4 in order to test connections between A and D, and A and E.

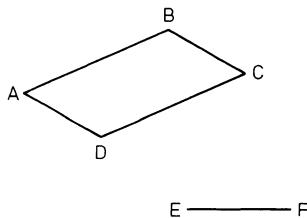


Fig. 4

Remark: This problem exemplifies the use of ‘ancestors’. Observe that your program must prevent loops caused by the circuits of the graph. Use the built-in procedure ‘ancestors’.

Logic program:

Program 1:

```

/* connections test */

connections(X,Y):- (connected(X,Y); connected(Y,X)),
    nl,write('yes.'),nl.
connections(_,_):- nl,write('no.'),nl.

```

```

connected(X,Y):- ancestors(L),
               in2(connected(X,Y),L),!,fail.
connected(X,Y):- edge(X,Y).
connected(X,Y):- edge(X,Z),connected(Z,Y).

in2(A,[A|B]):- in(A,B).
in2(A,[_|B]):- in2(A,B).

in(A,[A|_]). 
in(A,[_|B]):- in(A,B).

/* Graph description */

edge(a,b).          edge(b,c).          edge(e,d).
edge(d,a).          edge(e,f).

```

Program 2:

```

/* connection test */
connections(X,Y):-(connected(X,Y); connected(Y,X)),
                   nl,write('yes.'),nl.
connections(_,_):- nl,write('no.'),nl.

connected(X,Y):- edge(X,Y).
connected(X,Y):- edge(X,Z),
               (subgoal_of(connected(Z,Y)),!,fail;
                connected(Z,Y)).

/* Graph description */

edge(a,b).          edge(b,c).          edge(e,d).
edge(d,a).          edge(e,b).

```

Execution:

```

?-connections(a,d).
yes.

?-connections(a,e).
no.

```

Comments:

Note how the use of the built-in predicate ‘subgoal-of’ makes program 2 more readable than program 1. In fact, the use of ‘ancestors’ obliged us to define ‘in2’ and ‘in’.

Problem 108 [L. Pereira; Porto 1980]

Verbal statement:

Write a program for coloring any planar map with at most four colors, such that no two adjacent regions have the same color.

Logic program:

```
next(blue,yellow).
next(blue,red).
next(blue,green).
next(yellow,blue).
next(yellow,red).
next(yellow,green).
next(red,blue).
next(red,yellow).
next(red,green).
next(green,blue).
next(green,yellow).
next(green,red).
```

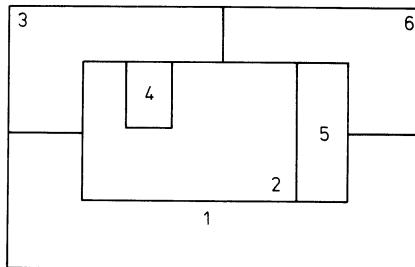


Fig.5

```
goal(R1,R2,R3,R4,R5,R6):- next(R1,R2),next(R1,R3),
next(R1,R5),next(R1,R6),
next(R2,R3),next(R2,R4),
next(R2,R5),next(R2,R6),
next(R3,R4),next(R3,R6),
next(R5,R6).
```

Comments:

The program consists of a complete list of pairs of different colors. These constitute the admissible pairs of colors for regions next to each other.

To obtain a coloring of the map above, we give as a goal to the program all the pairs of regions that are next to each other. This can be done systematically by first pairing region 1 with higher numbered regions next to it, then region 2, etc.

Problem 109

Verbal statement:

Write a program to solve the Soma cube, a three-dimensional puzzle designed by Piet Hein. He conceived of the cube during a lecture on quantum physics by Heisenberg. While the German physicist was speaking of a space sliced into cubes,

Piet Hein's supple imagination caught a fleeting glimpse of the following geometrical theorem. If you take all the irregular shapes that can be formed by combining no more than four cubes, all the same size and joined at their faces, these shapes can be put together to form a larger cube.

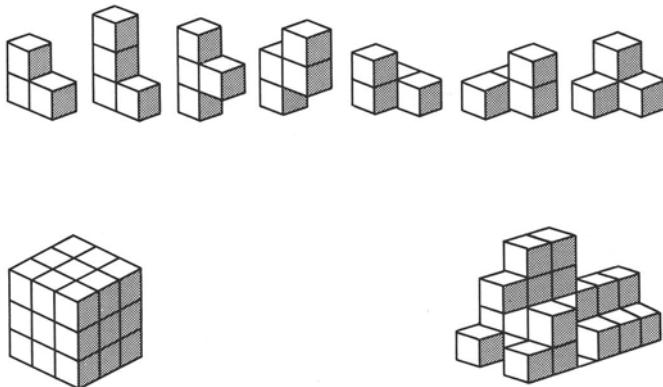


Fig. 6

In addition to the cube, the seven Soma pieces can be combined to construct a variety of three dimensional structures.

(The puzzle is described in Martin Gardner's book "More Mathematical Puzzles and Diversions", Penguin Books, 1966).

Logic program 1 [Swinson 1981]:

```

:- op(50,xfy,&).

/** shape(overall limits (X,Y,Z), missing list, ***/
/**      list of pieces to be used)           ***/

shape(Xl,Yl,Z,Missing,_) :-
    clear, piece(P,List,Xp&Yp&Zp),
    Xl >= Xp, Xr is Xl - Xp,
    Yl >= Yp, Yr is Yl - Yp,
    Zl >= Zp, Zr is Zl - Zp,
    nextn(0,Xr), nextn(0,Yr), nextn(0,Zr),
    store(P,List,X,Y,Z,Missing), fail.

shape(____,____,____,____,____):-
    write('Piece options set up.'), nl, fail.

shape(X,Y,Z,_,all):-
    fillit(X,Y,Z,[a,b,c,d,e,f,g],[]), more.

shape(X,Y,Z,_,Plist):-
    fillit(X,Y,Z,Plist,[]), more.

/** next number N between Start and End **/

```

```

nextn(Start,Start,End).
nextn(Start,N,End):- Start < End,
    Start1 is Start + 1, nextn(start1,N,End).

/* store(piece P filling List, shifted X,Y,Z, ***/
/* provided not into any Missing Unit) ***/
store(P,List,X,Y,Z,Missing):-
    move_piece(List,X,Y,Z,Mlist),
    check(Missing,Mlist),
    recordz(P,option(Mlist),_).

/* piece list moved X,Y,Z, is the list */
move_piece([],_,_,_,[]).
move_piece([A1&B1&C1|R1],X,Y,Z,[A2&B2&C2|R2]):-
    move_piece(R1,X,Y,Z,R2),
    A2 is A1 + X, B2, is B1 + Y, C2 is C1 + Z.

/* check that nothing in list 1 is in list 2 */
check([],Mlist).
check([A|Rest],Mlist):-
    noting(A,Mlist), check(Rest,Mlist).

/* term A is in list */
notin(A,[]).
notin(A,[B|Rest]):- A \== B,notin(A,Rest).

/* fillit(in X,Y,Z, List of pieces still to be used ***
/* list of pieces already placed ***/
fillit(X,Y,Z,[],Used):- show(X,Y,Z,Used), nl.
fillit(X,Y,Z,[P|Rest],Used):-
    option(P,List), check2(List,Used),
    fillit(X,Y,Z,Rest,[P,List|Used]).

/* check2 that nothing in list 1 is in list 2 ***
/* (cf "check" but every other in list 2) ***/
check2([],Used).
check2([A|Rest],Used):-
   notin2(A,Used), check2(Rest,Used).

/* term A is not in list (every other checked only) ***
notin2(A,[]).
notin2(A,[_,List|Rest]):- notin(A,List),notin2(A,Rest).

/* more question */
more:- write('Do you want me to try for another?'), nl,
      read(X), X == no.

/* show each layer of solution */

```

```

show(X,Y,Z,Used):- nl, nextn(1,Nz,Z),
    nl, nextn(1,Ny,Y), nl, Iy is Y-Ny+1,
    nextn(1,Nx,X), showit(Nx,Iy,Nz,Used),
    fail.

show(_,_,_,_):- nl, nl.

/* *** seek the piece at X,Y,Z (show ". " if none) ***/

showit(_,_,_,[]):- write('. ').

showit(X,Y,Z,[P,Rest]):-
    showit1(X,Y,Z,P,Rest), !.

showit(X,Y,Z,[_,_|Rest]):-
    showit(X,Y,Z,Rest).

/* *** seek piece at X,Y,Z and write "P " if found ***/

showit1(X,Y,Z,P,[X&Y&Z|_]):- write(P), write(' '), !.
showit1(X,Y,Z,P,[_|Rest]):- showit1(X,Y,Z,P,Rest).

/* *** get a recorded option ***/

option(P,List):- recorded(P,option(List),_).

/* *** clear old recorded options ***/

clear:- piece(P,_,_),
    recorded(P,option(_),R),
    erase(R),
    fail.

clear.

/* *** piece a (8 positions) ***/

piece(a,[1&1&1,1&2&1,1&1&2,2&1&1],2&2&2).
piece(a,[1&1&1,1&2&1,1&2&2,2&2&1],2&2&2).
piece(a,[1&2&1,1&1&2,1&2&2,2&2&2],2&2&2).
piece(a,[1&1&1,1&1&2,1&2&2,2&1&2],2&2&2).
piece(a,[1&1&1,2&1&1,2&2&1,2&1&2],2&2&2).
piece(a,[1&2&1,2&1&1,2&2&1,2&2&2],2&2&2).
piece(a,[1&2&2,2&2&1,2&1&2,2&2&2],2&2&2).
piece(a,[1&1&2,2&1&1,2&1&2,2&2&2],2&2&2).

```

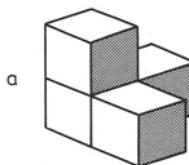


Fig. 7

```

/* *** piece b (12 positions) ***/

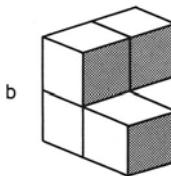
piece(b,[1&1&1,1&1&2,1&2&2,2&1&1],2&2&2).
piece(b,[1&1&1,1&2&1,1&1&2,2&2&1],2&2&2).

```

```

piece(b,[1&1&1,1&2&1,1&2&2,2&2&2],2&2&2).
piece(b,[1&2&1,1&1&2,1&2&2,2&1&2],2&2&2).
piece(b,[1&1&1,1&2&1,2&1&1,2&1&2],2&2&2).
piece(b,[1&2&1,1&2&2,2&1&1,2&2&1],2&2&2).
piece(b,[1&1&2,1&2&2,2&2&1,2&2&2],2&2&2).
piece(b,[1&1&1,1&1&2,2&1&2,2&2&2],2&2&2).
piece(b,[1&1&1,2&1&1,2&2&1,2&2&2],2&2&2).
piece(b,[1&2&1,2&2&1,2&1&2,2&2&2],2&2&2).
piece(b,[1&2&2,2&1&1,2&1&2,2&2&2],2&2&2).
piece(b,[1&1&2,2&1&1,2&2&1,2&1&2],2&2&2).

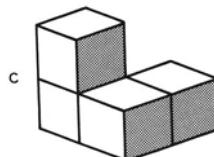
```

**Fig.8**

```

/*** piece c (12 positions) ***/
piece(c,[1&1&1,1&2&1,1&2&2,2&1&1],2&2&2).
piece(c,[1&2&1,1&1&2,1&2&2,2&2&1],2&2&2).
piece(c,[1&1&1,1&1&2,1&2&2,2&2&2],2&2&2).
piece(c,[1&1&1,1&2&1,1&1&2,2&1&2],2&2&2).
piece(c,[1&1&1,1&1&2,2&1&1,2&2&1],2&2&2).
piece(c,[1&1&1,1&2&1,2&2&1,2&2&2],2&2&2).
piece(c,[1&2&1,1&2&2,2&1&1,2&2&2],2&2&2).
piece(c,[1&1&2,1&2&2,2&1&1,2&1&2],2&2&2).
piece(c,[1&1&1,2&1&1,2&1&2,2&2&2],2&2&2).
piece(c,[1&2&1,2&1&1,2&2&1,2&1&2],2&2&2).
piece(c,[1&2&2,2&1&1,2&2&1,2&2&2],2&2&2).
piece(c,[1&1&2,2&2&1,2&1&2,2&2&2],2&2&2).

```

**Fig.9**

```

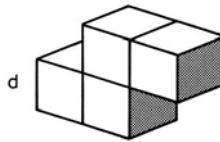
/*** piece d (12 positions) ***/
piece(d,[1&1&1,1&2&1,1&2&2,1&3&2],1&3&2).
piece(d,[1&2&1,1&1&2,1&2&2,1&1&3],1&2&3).

```

```

piece(d,[1&1&1,1&1&2,1&2&2,1&2&3],1&2&3).
piece(d,[1&2&1,1&3&1,1&1&2,1&2&2],1&3&2).
piece(d,[1&1&1,1&2&1,2&2&1,2&3&1],2&3&1).
piece(d,[1&1&1,1&1&2,2&1&2,2&1&3],2&1&3).
piece(d,[1&2&1,1&3&1,2&1&1,2&2&1],2&3&1).
piece(d,[1&1&2,1&1&3,2&1&1,2&1&2],2&1&3).
piece(d,[1&1&1,2&1&1,2&2&1,3&2&1],3&2&1).
piece(d,[1&1&1,2&1&1,2&1&2,3&1&2],3&1&2).
piece(d,[1&2&1,2&1&1,2&2&1,3&1&1],3&2&1).
piece(d,[1&1&2,2&1&1,2&1&2,3&1&1],3&1&2).

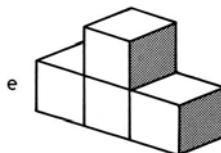
```

**Fig. 10**

```

/*** piece e (12 positions) ***/
piece(e,[1&1&1,1&2&1,1&3&1,1&2&2],1&3&2).
piece(e,[1&2&1,1&1&2,1&2&2,1&2&3],1&2&3).
piece(e,[1&2&1,1&1&2,1&2&2,1&3&2],1&3&2).
piece(e,[1&1&1,1&1&2,1&2&2,1&1&3],1&2&3).
piece(e,[1&1&1,1&2&1,1&3&1,2&2&1],2&3&1).
piece(e,[1&1&1,1&1&2,1&1&3,2&1&2],2&1&3).
piece(e,[1&2&1,2&1&1,2&2&1,2&3&1],2&3&1).
piece(e,[1&1&2,2&1&1,2&1&2,2&1&3],2&1&3).
piece(e,[1&1&1,2&1&1,2&2&1,3&1&1],3&2&1).
piece(e,[1&1&1,2&1&1,2&1&2,3&1&1],3&1&2).
piece(e,[1&2&1,2&1&1,2&2&1,3&2&1],3&2&1).
piece(e,[1&1&2,2&1&1,2&1&2,3&1&2],3&1&2).

```

**Fig. 11**

```

/*** piece f (24 positions) ***/
piece(f,[1&1&1,1&2&1,1&3&1,1&1&2],1&3&2).
piece(f,[1&1&1,1&2&1,1&2&2,1&2&3],1&2&3).

```

```

piece(f,[1&3&1,1&1&2,1&2&2,1&3&2],1&3&2).
piece(f,[1&1&1,1&1&2,1&1&3,1&2&3],1&2&3).
piece(f,[1&1&1,1&2&1,1&3&1,1&3&2],1&3&2).
piece(f,[1&2&1,1&2&2,1&1&3,1&2&3],1&2&3).
piece(f,[1&1&1,1&1&2,1&2&2,1&3&2],1&3&2).
piece(f,[1&1&1,1&2&1,1&1&2,1&1&3],1&2&3).
piece(f,[1&1&1,1&2&1,1&3&1,2&1&1],2&3&1).
piece(f,[1&1&1,1&1&2,1&1&3,2&1&1],2&1&3).
piece(f,[1&1&1,1&2&1,1&3&1,2&3&1],2&3&1).
piece(f,[1&1&1,1&1&2,1&1&3,2&1&3],2&1&3).
piece(f,[1&1&1,2&1&1,2&2&1,2&3&1],2&3&1).
piece(f,[1&1&1,2&1&1,2&1&2,2&1&3],2&1&3).
piece(f,[1&3&1,2&1&1,2&2&1,2&3&1],2&3&1).
piece(f,[1&1&3,2&1&1,2&1&2,2&1&3],2&1&3).
piece(f,[1&1&1,1&2&1,2&1&1,3&1&1],3&2&1).
piece(f,[1&1&1,1&1&2,2&1&1,3&1&1],3&1&2).
piece(f,[1&1&1,1&2&1,2&2&1,3&2&1],3&2&1).
piece(f,[1&1&1,1&1&2,2&2&1,3&2&1],3&2&1).
piece(f,[1&1&1,2&1&1,3&1&1,3&2&1],3&2&1).
piece(f,[1&2&1,2&2&1,3&1&1,3&2&1],3&2&1).
piece(f,[1&1&2,2&1&2,3&1&1,3&1&2],3&1&2).

```

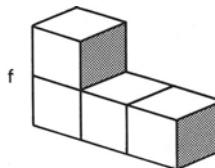


Fig. 12

```

/**** piece g (12 positions) ***/
piece(g,[1&1&1,1&2&1,1&1&2],1&2&2).
piece(g,[1&1&1,1&2&1,1&2&2],1&2&2).
piece(g,[1&2&1,1&1&2,1&2&2],1&2&2).
piece(g,[1&1&1,1&1&2,1&2&2],1&2&2).
piece(g,[1&1&1,1&2&1,2&1&1],2&2&1).
piece(g,[1&1&1,1&1&2,2&1&1],2&1&2).
piece(g,[1&1&1,1&2&1,2&2&1],2&2&1).
piece(g,[1&1&1,1&1&2,2&1&2],2&1&2).
piece(g,[1&1&1,2&1&1,2&2&1],2&2&1).
piece(g,[1&1&1,2&1&1,2&1&2],2&1&2).
piece(g,[1&2&1,2&1&1,2&2&1],2&2&1).
piece(g,[1&1&2,2&1&1,2&1&2],2&1&2).

```

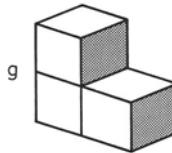


Fig. 13

standards:-

```

write('Some standard shapes other than the cube are:'),
nl, write('bath    bed    boat    castle'),
nl, write('dog     tower   well    zigzag'), nl).

bath:- shape(5,3,2,[2&2&2,3&2&2,4&2&2],all).

bed:- shape(6,3,2,[2&1&2,2&2&2,2&3&2,3&1&2,3&2&2,3&3&2,
4&1&2,4&2&2,4&3&2],all).

boat:- shape(5,5,3,[1&1&2,1&1&3,1&2&2,1&2&3,1&3&2,1&3&3,
1&4&1,1&4&2,1&4&3,1&5&1,1&5&2,1&5&3,2&1&2,2&1&3,
2&2&3,2&3&3,2&4&2,2&4&3,2&5&1,2&5&2,2&5&3,3&1&2,
3&1&3,3&2&3,3&4&3,3&5&2,3&5&3,4&1&1,4&1&2,4&1&3,
4&2&2,4&2&3,4&3&3,4&4&3,4&5&2,4&5&3,5&1&1,5&1&2,
5&1&3,5&2&1,5&2&2,5&2&3,5&3&2,5&3&3,5&4&2,5&4&3,
5&5&2,5&5&3],all).

castle:- shape(5,5,2,[1&1&1,1&1&2,1&2&2,1&3&2,1&4&2,
2&1&2,2&2&2,2&3&2,2&4&2,2&5&2,3&1&2,3&2&2,3&3&2,
3&4&2,3&5&2,4&1&2,4&2&2,4&3&2,4&4&2,4&5&2,
5&2&2,5&3&2,5&4&2],all).

cube:- shape(3,3,3,[],all).

dog:- shape(6,3,4,[1&2&1,3&1&1,3&3&1,1&1&2,1&2&2,1&3&2,
3&1&2,3&3&2,4&1&2,4&3&2,5&1&2,5&3&2,6&1&2,6&3&2,
1&1&3,1&3&3,2&1&3,2&3&3,3&1&3,3&3&3,4&1&3,4&2&3,
4&3&3,5&1&3,5&2&3,5&3&3,6&1&3,6&2&3,6&3&3,1&1&4,
1&2&4,1&3&4,2&1&4,2&3&4,3&1&4,3&3&4,4&1&4,4&2&4,
4&3&4,5&1&4,5&2&4,5&3&4,6&1&4,6&2&4,6&3&4],all).

tower:- shape(2,2,7,[1&1&7],all).

well:- shape(5,3,3,[2&2&1,2&2&2,2&2&3,4&1&1,4&1&2,4&1&3,
5&1&1,5&1&2,5&1&3,4&2&3,5&2&2,5&2&3,4&3&1,4&3&2,
4&3&3,5&3&1,5&3&2,5&3&3],all).

zigzag:- shape(5,5,3,[1&2&1,1&2&2,1&2&3,1&3&1,1&3&2,1&3&3,
1&4&1,1&4&2,1&4&3,1&5&1,1&5&2,1&5&3,2&3&1,2&3&2,
2&3&3,2&4&1,2&4&2,2&4&3,2&5&1,2&5&2,2&5&3,3&1&1,
3&1&2,3&1&3,3&4&1,3&4&2,3&4&3,3&5&1,3&5&2,3&5&3,
4&1&1,4&1&2,4&1&3,4&2&1,4&2&2,4&2&3,4&5&1,4&5&2,
4&5&3,5&1&1,5&1&2,5&1&3,5&2&1,5&2&2,5&2&3,5&3&1,
5&3&2,5&3&3],all).

```

Logic program 2 [F. Pereira 1982]:

```

shape(X,Y,Z,M,T) :-  

    table([X,Y,Z],T),  

    sculpt(M,T),  

    fill([a,b,c,d,e,f,g],T,p(X,Y,Z)).  
  

table([],_).  

table([N|Ns],T) :-  

    functor(T,t,N),  

    tables(N,Ns,T).  
  

tables(0,_,-) :- !.  

tables(N,Ns,T) :-  

    table(Ns,T0),  

    arg(N,T,T0),  

    N1 is N-1,  

    tables(N1,Ns,T).  
  

sculpt([],_).  

sculpt([P|Ps],T) :-  

    open(P,X,Y,Z),  

    mark(-,X,Y,Z,T),  

    sculpt(Ps,T).  
  

open(p(X,Y,Z),X,Y,Z).  
  

fill([],_,_,-).  

fill([P|Ps],T,B) :-  

    place(P,T,B),  

    fill(Ps,T,B).  
  

place(P,T,B) :-  

    free(P,B,T,X,Y,Z),  

    piece(P,L,Lo,Hi),  

    within(X,Y,Z,Lo,Hi,B),  

    uses(L,P,X,Y,Z,T).  
  

free(P,p(A,B,C),T,X,Y,Z) :-  

    in(X,A),  

    in(Y,B),  

    in(Z,C),  

    mark(P,X,Y,Z,T).  
  

uses([],_,_,_,_,_).  

uses([C|Cs],P,X0,Y0,Z0,T) :-  

    open(C,DX,DY,DZ),  

    X is DX+X0,  

    Y is DY+Y0,  

    Z is DZ+Z0,  

    mark(P,X,Y,Z,T),  

    uses(Cs,P,X0,Y0,Z0,T).

```

```

mark(P,X,Y,Z,ABC) :-  

    arg(X,ABC,BC),  

    arg(Y,BC,C),  

    arg(Z,C,P).  
  

in(X,X).  

in(X,X0) :-  

    X0>1,  

    X1 is X0-1,  

    in(X,X1).  
  

within(X,Y,Z,p(LX,LY,LZ),p(HX,HY,HZ),p(U,V,W)) :-  

    X+LX>0,  

    Y+LY>0,  

    Z+LZ>0,  

    X+HX=<U,  

    Y+HY=<V,  

    Z+HZ=<W.  
  

piece(a,[p(0,1,0),p(0,0,1),p(1,0,0)],p(0,0,0),p(1,1,1)).  

piece(a,[p(0,1,0),p(0,1,1),p(1,1,0)],p(0,1,0),p(1,1,1)).  

piece(a,[p(0,-1,1),p(0,0,1),p(1,0,1)],p(0,-1,1),p(1,0,1)).  

piece(a,[p(0,0,1),p(0,1,1),p(1,0,1)],p(0,0,1),p(1,1,1)).  

piece(a,[p(1,0,0),p(1,1,0),p(1,0,1)],p(1,0,0),p(1,1,1)).  

piece(a,[p(1,-1,0),p(1,0,0),p(1,0,1)],p(1,-1,0),p(1,0,1)).  

piece(a,[p(1,0,-1),p(1,-1,0),p(1,0,0)],p(1,-1,-1),p(1,0,0)).  

piece(a,[p(1,0,-1),p(1,0,0),p(1,1,0)],p(1,0,-1),p(1,1,0)).  

piece(b,[p(0,0,1),p(0,1,1),p(1,0,0)],p(0,0,0),p(1,1,1)).  

piece(b,[p(0,1,0),p(0,0,1),p(1,1,0)],p(0,0,0),p(1,1,1)).  

piece(b,[p(0,1,0),p(0,1,1),p(1,1,1)],p(0,1,0),p(1,1,1)).  

piece(b,[p(0,-1,1),p(0,0,1),p(1,-1,1)],p(0,-1,1),p(1,0,1)).  

piece(b,[p(0,1,0),p(1,0,0),p(1,0,1)],p(0,0,0),p(1,1,1)).  

piece(b,[p(0,0,1),p(1,-1,0),p(1,0,0)],p(0,-1,1),p(1,0,1)).  

piece(b,[p(0,1,0),p(1,1,-1),p(1,1,0)],p(0,1,-1),p(1,1,0)).  

piece(b,[p(0,0,1),p(1,0,1),p(1,1,1)],p(0,0,1),p(1,1,1)).  

piece(b,[p(1,0,0),p(1,1,0),p(1,1,1)],p(1,0,0),p(1,1,1)).  

piece(b,[p(1,0,0),p(1,-1,1),p(1,0,1)],p(1,-1,0),p(1,0,1)).  

piece(b,[p(1,-1,-1),p(1,-1,0),p(1,0,0)],p(1,-1,-1),p(1,0,0)).  

piece(b,[p(1,0,-1),p(1,1,-1),p(1,0,0)],p(1,0,-1),p(1,1,0)).  

piece(c,[p(0,1,0),p(0,1,1),p(1,0,0)],p(0,0,0),p(1,1,1)).  

piece(c,[p(0,-1,1),p(0,0,1),p(1,0,0)],p(0,-1,0),p(1,0,1)).  

piece(c,[p(0,0,1),p(0,1,1),p(1,1,1)],p(0,0,1),p(1,1,1)).  

piece(c,[p(0,1,0),p(0,0,1),p(1,0,1)],p(0,0,0),p(1,1,1)).  

piece(c,[p(0,1,0),p(1,1,0),p(1,1,1)],p(0,1,0),p(1,1,1)).  

piece(c,[p(0,0,1),p(1,-1,1),p(1,0,1)],p(0,-1,1),p(1,0,1)).  

piece(c,[p(0,1,0),p(1,0,-1),p(1,0,0)],p(0,0,-1),p(1,1,0)).  

piece(c,[p(1,0,0),p(1,0,1),p(1,1,1)],p(1,0,0),p(1,1,1)).
```

```

piece(c,[p(1,-1,0),p(1,0,0),p(1,-1,1)],p(1,-1,0),p(1,0,1)).
piece(c,[p(1,-1,-1),p(1,0,-1),p(1,0,0)],p(1,-1,-1),p(1,0,0)).
piece(c,[p(1,1,-1),p(1,0,0),p(1,1,0)],p(1,0,-1),p(1,1,0)).
piece(d,[p(0,1,0),p(0,1,1),p(0,2,1)],p(0,1,0),p(0,2,1)).
piece(d,[p(0,-1,-1),p(0,0,1),p(0,-1,2)],p(0,-1,1),p(0,0,2)).
piece(d,[p(0,0,1),p(0,1,1),p(0,1,2)],p(0,0,1),p(0,1,2)).
piece(d,[p(0,1,0),p(0,-1,1),p(0,0,1)],p(0,-1,0),p(0,1,1)).
piece(d,[p(0,1,0),p(1,1,0),p(1,2,0)],p(0,1,0),p(1,2,0)).
piece(d,[p(0,0,1),p(1,0,1),p(1,0,2)],p(0,0,1),p(1,0,2)).
piece(d,[p(0,1,0),p(1,-1,0),p(1,0,0)],p(0,-1,0),p(1,1,0)).
piece(d,[p(0,0,1),p(1,0,-1),p(1,0,0)],p(0,0,-1),p(1,0,1)).
piece(d,[p(1,0,0),p(1,1,0),p(2,1,0)],p(1,0,0),p(2,1,0)).
piece(d,[p(1,0,0),p(1,0,1),p(2,0,1)],p(1,0,0),p(2,0,1)).
piece(d,[p(1,-1,0),p(1,0,0),p(2,-1,0)],p(1,-1,0),p(2,0,0)).
piece(d,[p(1,0,-1),p(1,0,0),p(2,0,-1)],p(1,0,-1),p(2,0,0)).
piece(e,[p(0,1,0),p(0,2,0),p(0,1,1)],p(0,1,0),p(0,2,1)).
piece(e,[p(0,-1,1),p(0,0,1),p(0,0,2)],p(0,-1,1),p(0,0,2)).
piece(e,[p(0,-1,1),p(0,0,1),p(0,1,1)],p(0,-1,1),p(0,1,1)).
piece(e,[p(0,0,1),p(0,1,1),p(0,0,2)],p(0,0,1),p(0,1,2)).
piece(e,[p(0,1,0),p(0,2,0),p(1,1,0)],p(0,1,0),p(1,2,0)).
piece(e,[p(0,0,1),p(0,0,2),p(1,0,1)],p(0,0,1),p(1,0,2)).
piece(e,[p(1,-1,0),p(1,0,0),p(1,1,0)],p(1,-1,0),p(1,1,0)).
piece(e,[p(1,0,-1),p(1,0,0),p(1,0,1)],p(1,0,-1),p(1,0,1)).
piece(e,[p(1,0,0),p(1,1,0),p(2,0,0)],p(1,0,0),p(2,1,0)).
piece(e,[p(1,0,0),p(1,0,1),p(2,0,0)],p(1,0,0),p(2,0,1)).
piece(e,[p(1,-1,0),p(1,0,0),p(2,0,0)],p(1,-1,0),p(2,0,0)).
piece(e,[p(1,0,-1),p(1,0,0),p(2,0,0)],p(1,0,-1),p(2,0,0)).
piece(f,[p(0,1,0),p(0,2,0),p(0,0,1)],p(0,0,0),p(0,2,1)).
piece(f,[p(0,1,0),p(0,1,1),p(0,1,2)],p(0,1,0),p(0,1,2)).
piece(f,p(0,-2,1),p(0,-1,1),p(0,0,1)],p(0,-2,1),p(0,0,1)).
piece(f,[p(0,0,1),p(0,0,2),p(0,1,2)],p(0,0,1),p(0,1,2)).
piece(f,[p(0,1,0),p(0,2,0),p(0,2,1)],p(0,1,0),p(0,2,1)).
piece(f,[p(0,0,1),p(0,-1,2),p(0,0,2)],p(0,-1,1),p(0,0,2)).
piece(f,[p(0,0,1),p(0,1,1),p(0,2,1)],p(0,0,1),p(0,2,1)).
piece(f,[p(0,1,0),p(0,0,1),p(0,0,2)],p(0,0,0),p(0,1,2)).
piece(f,[p(0,1,0),p(0,2,0),p(1,0,0)],p(0,0,0),p(1,2,0)).
piece(f,[p(0,0,1),p(0,0,2),p(1,0,0)],p(0,0,0),p(1,0,2)).
piece(f,[p(0,0,1),p(0,0,2),p(1,0,0)],p(0,0,0),p(1,0,2)).
piece(f,[p(0,1,0),p(1,1,0),p(1,2,0)],p(1,0,0),p(1,2,0)).
piece(f,[p(1,0,0),p(1,0,1),p(1,0,2)],p(1,0,0),p(1,0,2)).
piece(f,[p(1,-2,0),p(1,-1,0),p(1,0,0)],p(1,-2,0),p(1,0,0)).
piece(f,[p(1,0,-2),p(1,0,-1),p(1,0,0)],p(1,0,-2),p(1,0,0)).
piece(f,[p(0,1,0),p(1,0,0),p(2,0,0)],p(0,0,0),p(2,1,0)).
piece(f,[p(0,0,1),p(1,0,0),p(2,0,0)],p(0,0,0),p(2,0,1)).
piece(f,[p(0,1,0),p(1,1,0),p(2,1,0)],p(0,1,0),p(2,1,0)).
piece(f,[p(0,0,1),p(1,0,1),p(2,0,1)],p(0,0,1),p(2,0,1)).

```

```

piece(f,[p(1,0,0),p(2,0,0),p(2,1,0)],p(1,0,0),p(2,1,0)).
piece(f,[p(1,0,0),p(2,0,0),p(2,0,1)],p(1,0,0),p(2,0,1)).
piece(f,[p(1,0,0),p(2,-1,0),p(2,0,0)],p(1,-1,0),p(2,0,0)).
piece(f,[p(1,0,0),p(2,0,-1),p(2,0,0)],p(1,0,-1),p(2,0,0)).
piece(g,[p(0,1,0),p(0,0,1)],p(0,0,0),p(0,1,1)).
piece(g,[p(0,1,0),p(0,1,1)],p(0,1,0),p(0,1,1)).
piece(g,[p(0,-1,1),p(0,0,1)],p(0,-1,1),p(0,0,1)).
piece(g,[p(0,0,1),p(0,1,1)],p(0,0,1),p(0,1,1)).
piece(g,[p(0,1,0),p(1,0,0)],p(0,0,0),p(1,1,0)).
piece(g,[p(0,0,1),p(1,0,0)],p(0,0,0),p(1,0,1)).
piece(g,[p(0,1,0),p(1,1,0)],p(0,1,0),p(1,1,0)).
piece(g,[p(0,0,1),p(1,0,1)],p(0,0,1),p(1,0,1)).
piece(g,[p(1,0,0),p(1,1,0)],p(1,0,0),p(1,1,0)).
piece(g,[p(1,0,0),p(1,0,1)],p(1,0,0),p(1,0,1)).
piece(g,[p(1,-1,0),p(1,0,0)],p(1,-1,0),p(1,0,0)).
piece(g,[p(1,0,-1),p(1,0,0)],p(1,0,-1),p(1,0,0)).

```

Comments:

The logic program 1 operates in two stages.

The first stage is designed to restrict the combinatorial search options by setting up in advance only those locations for each piece that are possible within the structure to be built. The second stage seeks a combination of locations, one for each of the pieces to be used in the structure, such that no two pieces clash.

Problem 110 [Abreu 1982, personal communication]

Verbal statement:

Given an array $m \times n$ ($n \leq 20$) when $x_{ij} = 1$ or 0 and $x_{ii} = 0$, draw the corresponding digraph, knowing that n_i belongs to one of three sets $\{a, b, c\}$. From the graphic point of view, the sets are represented as circles with the same center. Interaction among links from the digraph nodes is not admissible.

Logic program:

```

% To use this, just say at interpreter top level:
%
% case(N,Fname).
%
% Where N is the list of cases you want to solve, and Fname is
% the output
% file for the solution. e.g. case([1,2,3], 'solution.pl')
% will
% do it.

:-op(120,xfx,'in').
:-op(100,xfy,':').
:-op(99,xfx,'/').

```

```

case(N, Fname):-
    case(N),
    tell(Fname),
    write_solutions(N).

case([]):-
    write('I''ve just finished solving all you asked for.'), nl, nl.

case([N|T]):-
    write('case number '), write(N), write(' '), ttyflush,
    ( solve(N, Solution_N), !,
      assertz(solution(N, Solution_N)),
      write(' *** has been solved:'), nl,
      show(Solution_N) ;
      write(' *** can''t be solved!'), nl, nl), !,
    case(T), !.

write_solutions([]):-
    told, tell(user),
    write('And wrote out the solutions...'), nl, nl.

write_solutions([N|T]):-
    solution(N, X),
    output(N, X),
    write_solutions(T).

% Now the program itself...

solve(N,B):-
    bagof(Loc/Lev,
          Locs(level_description(Lev,Locs), Loc in Locs),
          Places),
    setof(X:Y/L, Ylevel(N,X,L), XX),
    list_of_links(N, Links), !,
    fill(N, XX, B, Places, XX, [], Links).

%-----%
% Predicate "fill" takes care of:
%   a) finding a combination.
%   b) making sure it's got no intersections.
% It's a non-determinate predicate, i.e. it may backtrack.
%-----%

fill(_, [], [], _, _, _, []).
fill(N,[X:Lc/Lv|Xl],[X:Lc/Lv:Lnks|R],Free,Lcs,Segs,[X:Lnks|L]):-
    remove(Lc/Lv, Free, NFree),
    show_move(X),
    ok(X:Lc/Lv, Lnks, Segs, NSegs, Lcs),
    fill(N, Xl, R, NFree, Lcs, NSegs, L).

%-----%
% Predicate "ok" makes sure a given (partial) combination is ok,

```

```
% i.e. it's got no intersections: It's given the name/location
% of the last point that was inserted and matches all the seg-
% ments involved to all the other ("previous") segments, i.e.
% those that don't contain the "last" point... these are
% assumed to be consistent that is, they have no intersections
% within their group...
%
% All these predicates (ok, et al.) are determinate.
%-----
```

```
ok(X,Links,Segments,NSegments,Locs):-  

    review(Segments,X,FSegs,OSegs),!, % "fixed-up" segments...  

    compatible(FSegs,Segments),!, % make sure they're ok.  

    segments(X,Links,Locs,NSegs),!, % figure out the new ones.  

    compatible(NSegs,Segments),!, % and make sure they're  

    % ok.  

    concatenate(NSegs,Segments,NSegments),!.  

review([], _, [], []):- !.  

review([(X,Y)|L], X, [(X,Y)|N], O):- review(L,X,N,O), !.  

review([(Y,X)|L], X, [(Y,X)|N], O):- review(L,X,N,O), !.  

review([(Y,Z)|L], X, N, [(Y,Z)|O]) :- review(L,X,N,O), !.-  

compatible([], _):- !.  

compatible(_, []):- !.  

compatible([S|L], Segs):-  

    compatible1(S, Segs), !,  

    compatible(L, Segs), !.  

compatible1(_,[ ]):- !.  

compatible1((_:P1/_, _:P2/_), _):-  

    (var(P1); var(P2)), !.  

compatible1(S, [(_:P3/_, _:P4/_),.. L]) :-  

    (var(P3); var(P4)), !,  

    compatible1(S, L), !.  

compatible1((A:P1/L1, B:P2/L2), [(C:P3/L3, D:P4/L4)|L]) :-  

    noint(P1,L1,P2,L2, P3,L3,P4,L4), !,  

    compatible1((A:P1/L1, B:P2/L2), L), !.  

segments(_, [], _, []):- !.  

segments(X, [Y|T], Locs, [(X,Y:P/L)|Segs]) :-  

    Y:P/L in Locs, !,  

    segments(X, T, Locs, Segs), !.  

noint(A,X,B,Y,C,Z,D,T) :- int(A,X,B,Y,C,Z,D,T), !, fail.  

noint(A,X,B,Y,C,Z,D,T) :- int(A,X,B,Y,D,T,C,Z), !, fail.  

noint(A,X,B,Y,C,Z,D,T) :- int(B,Y,A,X,C,Z,D,T), !, fail.  

noint(A,X,B,Y,C,Z,D,T) :- int(B,Y,A,X,D,T,C,Z), !, fail.  

noint(A,X,B,Y,C,Z,D,T) :- int(C,Z,D,T,A,X,B,Y), !, fail.  

noint(A,X,B,Y,C,Z,D,T) :- int(D,T,C,Z,A,X,B,Y), !, fail.
```

```

noint(A,X,B,Y,C,Z,D,T):- int(C,Z,D,T,B,Y,A,X), !, fail.
noint(A,X,B,Y,C,Z,D,T):- int(D,T,C,Z,B,Y,A,X), !, fail.
noint(_,_,_,_,_,_,_,_):- !.

list_of_links(N, L):-
    setof(X, Ylevel(N,X,Y), Xs), !,
    hack_links(N, Xs, L).

hack_links(_, [], []):- !.
hack_links(N, [X|T], [X:L|Ls]):-
    ( setof(Lk, link(N, X, Lk), L); L=[] ), !,
    hack_links(N, T, Ls), !.

% Output predicates. These are also all determinate.

output(N,X):-
    write('N '), write(N), nl,
    setof(R, LRadius(L,R), Radiuses), !,
    circles(Radiuses), !,
    defpoints(X), !,
    setof(P, Llevel(N,P,L), Points), !,
    deflinks(N,Points).

circles([]):- !.
circles([Radius|Radiuses]):-
    write('C '), write(Radius), nl,
    circles(Radiuses), !.

defpoints([]):- !.
defpoints([P:L:_|Points]):-
    point(L,X,Y), !,
    write('P '), write(P), write(X), write(Y), nl,
    defpoints(Points).

deflinks(_,[]):- !.
deflinks(N,[P|Points]):-
    ( setof(Q, link(N,P,Q), To) ;
      To = [] ), !,
    deflink(P, To), !,
    deflinks(N,Points), !.
deflink(_,[]):- !.
deflink(P,[Q|Rest]):-
    write('L '), write(P), write(' '), write(Q), nl, !,
    deflink(P,Rest), !.

/* Utility predicates, this should be compiled... */
remove(X,[X|T],T).
remove(X,[Y|T],[Y|Tr]):- remove(X,T,Tr).

X in [X|_].
X in [_|L]:- X in L.

```

```

concatenate([],L,L):- !.
concatenate(L,[],L):- !.
concatenate([X|Y], L, [X|Z]):- concatenate(Y, L, Z).

show1([]):- !.
show1([X:P/L:_,... Rest]):-
    write(X), write(':''), write(P), write('/'), write(L),
    write(' '), !,
    show1(Rest), !.
show(B):- write('           pattern is '), show1(B), nl, nl.
show_move(X):-
    write(' '), write(X), ttyflush.          % write current move.
show_move(_):-
    put(8), put(8), put(7),                 % erase last move
                                                when backtracking.
    write(' '), put(8), put(8), ttyflush, !, fail.

```

Execution:

```

| ?- case([1,2,3,4,5,6], 'bc.sol').
case number 1 c d i j m n p *** has been solved:
           pattern is c:1/3 d:2/3 i:5/3 j:1/1 m:1/2 n:2/1 p:4/3
case number 2 c d i j m n p *** has been solved:
           pattern is c:1/3 d:2/3 i:3/3 j:2/1 m:2/2 n:3/1 p:4/3
case number 3 c d i j m n p *** has been solved:
           pattern is c:1/3 d:1/2 i:3/3 j:1/1 m:4/2 n:2/1 p:2/3
case number 4 c d i j m n p *** has been solved:
           pattern is c:1/3 d:1/2 i:3/3 j:1/1 m:2/3 n:3/1 p:5/3
case number 5 c d i j m n p *** has been solved:
           pattern is c:1/3 d:3/3 i:2/3 j:2/1 m:2/2 n:3/1 p:5/3
case number 6 *** can't be solved,
I've just finished solving all you asked for.

no
| ?- [-apes2].
apes2 reconsulted   1968 words      3.38 sec.
yes
| ?- listing(case).
yes
| ?- told.
yes
| ?- listing(case).
case([]) :-
    write('I''ve just finished solving all you asked for.'), nl,nl.

```

```

case([_1|_2]) :-
    write('case number '),
    write(_1),
    write(' '),
    ttyflush,
    ( solve(_1,_3),! ,
        assertz(solution(_1,_3)),
        write(' *** has been solved:'),
        nl,
        show(_3);
    write(' *** can't be solved!'),
    nl,
    nl ),!,
    case(_2),!.
```

Comments:

This a good example of how a different style in logic programming can be adopted. The influence of C and Assembly languages is heavy!

A B C D	
A 0 1 0 1	
B 1 0 0 0	
C 1 1 0 0	where A-->a
D 0 0 1 0	B,C-->b
	D-->c

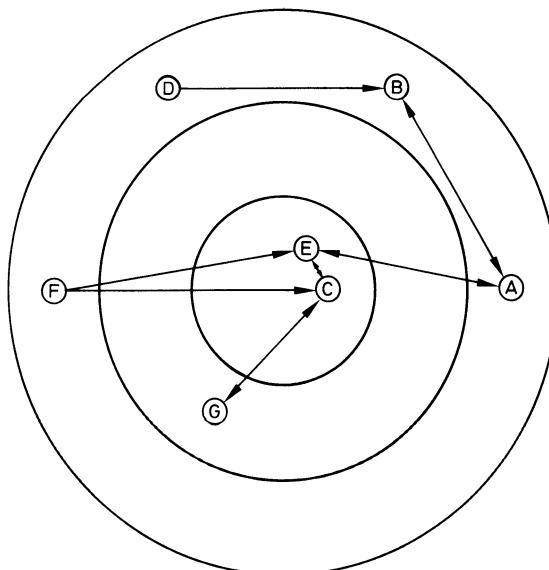


Fig. 14

Chapter 13 Planning with Prolog

A major theme in Artificial Intelligence is the quest for methods for reasoning about hypothetical sequences of activities (actions and their effects). Such methods are called automatic problem solvers, and their tasks are to find sequences of operators (plans) transforming the initial state of the world into a final state in which the goal statement is true.

Problem 111

Verbal statement (Monkey-and-bananas problem):

A monkey is in a room with a bunch of bananas which he very much desires. Bananas are suspended from the ceiling, and there is a chair across the room from the bananas. The monkey is able to move the chair. If he climbs the chair, which he can do, he can reach the bananas. Write a program to discover the way (plan) the monkey can solve his problem.

Logic program:

```
plan(P,P,[[]]).  
plan(Po,Pf,[Action|Actions]):- schedule(Po,Pi,Action),  
                                plan(Pi,Pf,Actions).  
  
/* Transitions between states */  
schedule(start,position(monkey,bananas,chair),start).  
schedule(position(Y,Y,Y),final,climb).  
schedule(position(X,Y,X),position(Y,Y,Y),carry(X,Y)).  
schedule(position(X,Y,Z),position(Z,Y,Z),walk(X,Z)).
```

Execution:

```
?- plan(start,final,Plan).  
Plan=[start,walk(monkey,chair),carry(chair,bananas),climb]
```

The monkey walks to reach the chair, after he carries it under the bananas, he climbs the chair and he grasps the bananas.

Comments:

This is a typical example of common sense reasoning. A simple form of planning model is adopted and described as a state transition diagram, with transitions corresponding to activities. The output of the model is a sequence of states (or activities) from a given start state to a certain goal state. The above program uses only three operators, ‘walk’, ‘carry’ and ‘climb’. A ‘plan’ is defined by the initial (‘Pi’) and final (‘Pf’) positions and by a sequence of actions able to change states. When the two positions are identical there are no actions, and the third argument of the relation ‘plan’ is empty (nil).

In this problem, states were not given individual names, but rather were identified in terms of more elementary components.

Problem 112

Verbal statement (Missionaries and cannibals problem):

Three missionaries and three cannibals seek to cross a river. A boat is available which holds two people, and which can be navigated by any combination of missionaries and cannibals involving one or two people. If the missionaries on either bank of the river, or ‘en route’ in the river, are outnumbered at any time by the cannibals, the cannibals will indulge their anthropophagic tendencies and do away with the missionaries. Find the simplest schedule of crossing that will permit all the missionaries and cannibals to cross the river safely.

Suggestion: Consider the representation of the left side of the river (for example) as s(M,C,B) where M is the number of missionaries, C is the number of cannibals, and B takes the value 0 or 1 accordingly with the position of the boat (left side or right side respectively).

Logic program:

```

:-op(300, xfy, =>).

plan:- movements(s(3,3,1),P),print(P).

movements(s(0,0,0),done).

movements(X,X=>Y=>Z):- move(X,Y),move_back(Y,Y1),
                           test(X,Y1),
                           movements(Y1,Z).

move(X,Y):- move(X,Y,2);move(X,Y,1);move11(X,Y).

move_back(s(0,0,0),s(0,0,0)).
move_back(X,Y):- move_back(X,Y,2); move_back(X,Y,1);
                           move_back11(X,Y).

move(s(M,C,X),s(M1,C,Y),N):-
                           (M=N ;
                            M>N, M-N>C),
                           M1 is M-N, M1=<3,
                           Y is 1-X.

```

```

move(s(M,C,X),s(M,C1,Y),N) :- C>=N, C1 is C-N,
                                         C1=<3, Y is 1-X.

move11(s(M,C,X),s(M1,C1,Y)) :- M>=1, C>=1,
                                         M1 is M-1, C1 is C-1,
                                         Y is 1-X.

move_back(s(M,C,X),s(M1,C,Y),N) :- N=<3-M, M+N=C,
                                         M1 is M+N, Y is 1-X.

move_back(s(M,C,X),s(M,C1,Y),N) :- N=<3-C,
                                         (M=0 ; M>=C+N),
                                         C1 is C+N, Y is 1-X.

move_back11(s(M,C,X),s(M1,C1,Y)) :- M>=1, C>=1,
                                         M1 is M+1, C1 is C+1,
                                         Y is 1-X.

test(s(M,C,_),s(M,C,_)) :- !, fail.
test(_,_).

print(X=>Y) :- write(X), nl, print(Y).
print(X) :- nl, write(X), nl.

```

Execution:

```
?- plan.
```

```

s(3,3,1)
s(3,1,0)
s(3,2,1)
s(3,0,0)
s(3,1,1)
s(1,1,0)
s(2,2,1)
s(0,2,0)
s(0,3,1)
s(0,1,0)
s(0,2,1)
s(0,0,0)

```

```
done
```

Comments:

The essence of this problem is that it involves a search through a state space, with choices available as to which transition to take.

The plan is defined as a sequence of movements between states (3,3,1) and (0,0,0). The sequence is represented by 'X = > Y = > Z', where 'X', 'Y' and 'Z' are the variables standing for states. The predicate 'test' inside 'movements' checks whether the 'move_back' 'Y1' is not equal to 'move' 'X'. This is the device to avoid the repetition of movements.

A ‘move’ may be achieved through three different forms: a transport with two persons of the same type, a transport with two persons of different type, and a transport of one single person.

Every ‘move’ is submitted to certain conditions. For example, the conditions affecting the transport of N missionaries are described by the first clause of ‘move’. The conditions affecting the transport of N cannibals are described by the second clause of ‘move’. And, the conditions affecting the transport of one cannibal and one missionary are described by ‘move11’. The specifications of the conditions affecting the backwards moves are described in general terms by two clauses ‘move _ back’. The first one states what happens when state (0,0,0) is reached. In fact a ‘move _ back’ is not possible when this state is reached. The second clause of ‘move _ back’ states that the transport may be composed either of two persons of the same type, different type, or with one person.

The ‘movements’ of N missionaries, N cannibals and two persons with different types are described by the three following clauses of ‘move _ back’.

The printing mechanism of a ‘plan’ consists in translating the sequence of moves.

Problem 113 [Warren 1974a]

Verbal statement:

Many problem domains can naturally be formalized as a world with a set of actions which transform that world from one state to another.

A particular problem is then specified by describing an initial state and a desired goal state. The problem solver is required to generate a plan, a simple sequence of actions which transforms the world from the initial state to the goal state.

Write a problem solver (WARPLAN), independent of any particular problem and able to generate a plan of actions (in order to transform an initial state into a goal state) for each database (world description) you give it.

Logic program:

```

:-op(700,xfy,&).
:-op(650,yfx,=>).

/* Problem solver entry : Generation and output of a plan */

plans(C,T):- not(consistent(C,true)),!,nl,
           write('impossible'),nl,nl.
plans(C,T):- plan(C,true,T,T1),nl,output(T1),nl,nl.

output(Xs=>X):- !,output1(Xs),write(X),write('.'),nl.
output1(Xs=>X):- !,output1(Xs),write(X),write(' ;'),nl.
output1(X):- write(X),write(' ;'),nl.

/* Entry point to the main recursive loop */

```

```

plan(X&C,P,T,T2):- !, solve(X,P,T,P1,T1)plan(C,P1,T1,T2).
plan(X,P,T,T1):- solve(X,P,T,P1,T1).

/* Ways of solving a goal */

solve(X,P,T,P,T):- always(X) ; X.
solve(X,P,T,P1,T):- holds(X,T), and(X,P,P1).
solve(X,P,T,X&P,T1):- add(X,U), achieve(X,U,P,T,T1).

/* Methods of achieving an action */

/* By extension */

achieve(X,U,P,T,T1=>U):- preserves(U,P),
                           can(U,C),
                           consistent(C,P),
                           plan(C,P,T,T1),
                           preserves(U,P).

/* By insertion */

achieve(X,U,P,T=>V,T1=>V):- preserved(X,V),
                           retrace(P,V,P1),
                           achieve(X,U,P1,T,T1),
                           preserved(X,V).

/* Check if a fact holds in a certain state */

holds(X,T=>V):- add(X,V).
holds(X,T=>V):- !, preserved(X,V),
                           holds(X,T),
                           preserved(X,V).
holds(X,T):- given(T,X).

/* Prove that an action preserves a fact */

preserves(U,X&C):- preserved(X,U), preserves(U,C).
preserves(_,true).

preserved(X,V):- numbervars(X&V,0,N),
                           del(X,V),!,fail.
preserved(_,_).

/* Retracing a goal already achieved */

retrace(P,V,P2):- can(V,C),
                           retrace1(P,V,C,P1),
                           append(C,P1,P2).

retrace1(X&P,V,C,P1):- add(Y,V), equiv(X,Y),!,
                           retrace1(P,V,C,P1).
retrace1(X&P,V,C,P1):- elem(Y,C), equiv(X,Y),!,
                           retrace1(P,V,C,P1).

```

```

retrace1(X&P,V,C,X&P1):-retrace1(P,V,C,P1).
retrace1(true,V,C,true).

/* Consistency with a goal already achieved */

consistent(C,P):- numbervars(C&P,0,N),
    imposs(S),
    not(not(intersect(C,S))),
    implied(S,C&P),!,fail.

consistent(_,_).

/* Utility routines */

and(X,P,P):- elem(Y,P),equiv(X,Y),!.
and(X,P,X&P).

append(X&C,P,X&P1):- !,append(C,P,P1).
append(X,P,X&P).

elem(X,Y&C):- elem(X,Y).
elem(X,Y&C):-!,elem(X,C).
elem(X,X).

implied(S1&S2,C):- !,implied(S1,C),implied(S2,C).
implied(X,C):- elem(X,C).
implied(X,C):- X.

intersect(S1,S2):- elem(X,S1),elem(X,S2).

not_equal(X,Y):- not(X=Y),
    not(X='$VAR'(_)),
    not(Y='$VAR'(_)).

equiv(X,Y):- not(nonequiv(X,Y)).

nonequiv(X,Y):- numbervars(X&Y,0,N),X=Y,! ,fail.
nonequiv(_,_).

```

Comments:

- 1) The operators' interpretation is the following:
 'X&Y' is 'X and Y'
 'X => Y' is 'the state after doing Y in X'
- 2) The central predicate is 'plan(C,P,T,T1)'; it has 4 arguments:
 C is a conjunction of goals to be solved;
 T is a (already generated) partial plan;
 P is a conjunction of goals already solved by T which must be protected;
 T1 is a new plan, which contains T as a subplan and preserves the already solved goals P, and which also solves the new goals C.
 Essentially this predicate states that a plan can be produced by 'solve'-ing each goal in the given order (C,P,T behave as input variables and T1 as output variable).

3) The predicate ‘solve(X,P,T,P1,T1)’ has the following arguments:

- X is an atomic goal;
- T is a partial plan;
- P is a conjunction of goals achieved by T;
- T1 is a plan, containing T as a subplan, which solves P1;
- P1 is a conjunction comprising P and X, where X is not repeated.

There are three ways (methods) in which a goal may be ‘solve’-d. It may be ‘always’ true in the world. It may be that it already ‘holds’ in the state produced by the current partial plan. Finally we may look in the database for action U which ‘add’-s the goal X and then ‘solve’-s X by ‘achieve’-ing U (X,P,T behave as input variables and P,T1 as output variables).

4) The predicate ‘achieve(X,U,P,T,T1)’ has two clauses corresponding to two methods of ‘achieve’-ing an action: extension and insertion.

The clause for the extension method (the first one) checks that the action U ‘preserves’ (i.e., does not delete) the protected facts P. Then we look up the preconditions C in the database and check that C is consistent with the protected facts. All being well, we call ‘plan’ recursively to modify the current plan T to a new T1 which produces a state in which C and P are attained. U can then be applied in T1, corresponding to the plan resulting from this call of ‘achieve’.

Finally, the check that U preserves P is repeated, since U and P may not have been instantiated to ground terms at the time of the original check.

The clause for the insertion method (the second one) follows the axiom: if the last action V in the current partial plan does not delete the current goal X, we can try to insert the action U somewhere before V, provided we retrace the set of protected facts to the point before V.

5) ‘holds(X,Y)’ follows the method: everything that ‘holds’ in a state of the world can be determined from the plan which produces that state of the world. The system chains backwards through the sequence of actions, so long as none of these actions deletes the sought-for facts, until the fact is found in the ‘add’-set of an action or was given in the initial state.

6) The predicate ‘numbervars(X,N1,N2)’ was a built-in predicate that existed in the DEC-10 implementation, having as effect instantiation of the variables occurring in X with a function of N1 (first occurrence) to N2. The actual terms used were \$VAR(N1) to \$VAR(N2); this is the reason why the tests in predicate ‘not _ equal’ are made against those terms.

7) The meaning of ‘output(Xs = > X)’ is: the state after doing ‘X’ in ‘Xs’.

Problem 114 [Warren 1974a]

Verbal statement:

Using the problem solver WARPLAN, introduced in the previous problem, write the world description and the initial state for the 3 blocks shown in Fig. 1.

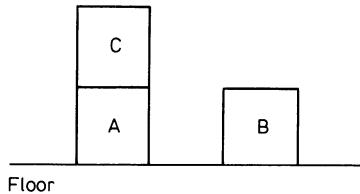


Fig. 1

Achieve the goal state $on(a,b) \& on(b,c)$.

Suggestion:

Use the predicates

$add(X, U)$	with the meaning	fact X is added by action U; i.e. X is true in any state resulting from U (and U is a possible action in some state in which X is not true)
$del(X, U)$	with the meaning	fact X is deleted by action U; i.e. it is not the case that X is preserved by U. (X is preserved by U if and only if X is not added by U and X is true in a state resulting from U whenever X is true in the preceding state)
$can(U, C)$	with the meaning	the conjunction of facts C is the precondition of action U; i.e. U is possible in any state in which C is true.
$always(X)$	with the meaning	fact X is true in any state
$imposs(X)$	with the meaning	the conjunction of facts X is impossible in any state
$given(T, X)$	with the meaning	fact X is true in the initial state T (but it is not the case that X is true in all states).

Logic program:

In order to use WARPLAN we have just to write the database according to the block situation above.

```
/* Data Base for the three blocks problem */

add(on(U,W),move(U,V,W)).
add(clear(V),move(U,V,W)).

del(on(U,Z),move(U,V,W)).
del(clear(W),move(U,V,W)).

can(move(U,V,floor),on(U,V)&not_equal(V,floor)&clear(U)).
can(move(U,V,W),clear(W)&on(U,V)&not_equal(U,W)&clear(U)).

imposs(on(X,Y)&clear(Y)).
imposs(on(X,Y)&on(X,Z)&not_equal(Y,Z)).
imposs(on(X,X)).
```

```
given(start, on(a, floor)).
given(start, on(b, floor)).
given(start, on(c, a)).
given(start, clear(b)).
given(start, clear(c)).
```

Execution:

After adding this database to WARPLAN you just have to give the command:

```
?-plans(on(a,b)&on(b,c), start).
```

the solution is:

```
start ;
move(c,a,floor) ;
move(b,floor,c) ;
move(a,floor,b) .
```

Problem 115

Verbal statement:

Consider the initial state of 5 blocks as shown in Fig.2 below. Apply the program WARPLAN, presented in Problem 113, to achieve the goal state:

```
on(a,b)&on(b,c)&on(c,d)&on(d,e),
```

as was done in the preceding problem.

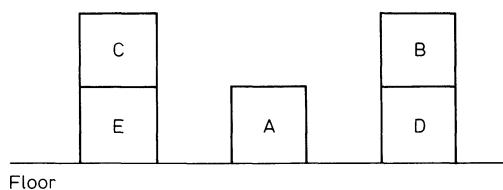


Fig. 2

Logic program:

Same facts ‘add’ , ‘del’ , ‘can’ , and ‘imposs’ as those in the preceding problem.

The description of the initial state is:

```
/* Data Base for the five blocks problem */

given(start, on(e, floor)).
given(start, on(c, e)).
given(start, clear(c)).
```

```
given(start, on(a,floor)).  
given(start, clear(a)).  
given(start, on(d,floor)).  
given(start, on(b,d)).  
given(start, clear(b)).
```

Execution:

The command:

```
?- plans(on(a,b)&on(b,c)&on(c,d)&on(d,e), start).
```

runs the WARPLAN program and produces:

```
start ;  
move(b,d,floor) ;  
move(c,e,floor) ;  
move(d,floor,e) ;  
move(c,floor,d) ;  
move(b,floor,c) ;  
move(a,floor,b) .
```

Problem 116 [Warren 1974a]

Verbal statement:

Consider a world comprising two areas ‘inside’ and ‘outside’. In the area ‘inside’ there are four distinct locations, namely ‘table’, ‘box1’, ‘box2’, ‘door’. There is a robot which is able to move about and transport objects. If the robot attempts to pickup an object at a location, all that can be ascertained is that it will be holding one of the objects, if any, at that location. The robot is only allowed to pickup an object if it is the only object at a location. Consider 3 objects ‘key1’, ‘key2’ and ‘red1’ and two actions ‘go’ and ‘take’. The robot is only allowed to ‘go’ or ‘take’ something ‘outside’ if both the objects ‘key1’ and ‘key2’ are at the ‘door’.

Consider the initial state, given in Fig.3:

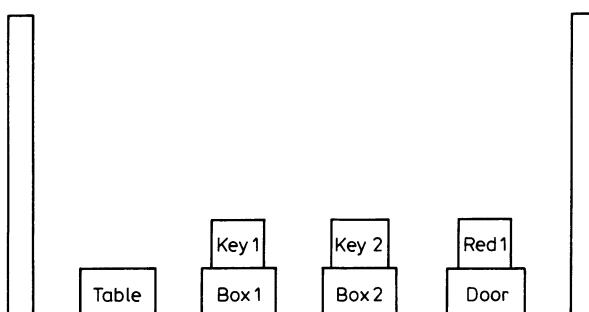


Fig. 3

Use the program WARPLAN, presented in Problem 112, to achieve the goal: to have 'redi' outside.

Logic program:

```

:-op(300,yfx,'IS').
:-op(200,fx,'set').
:-op(200,fx,'placed').
:-op(200,fx,'only').
:-op(100,yfx,'at').

add(position 'IS' P,go(P)).
add(X 'IS' placed Q,take(X,P,Q)).
add(set placed P 'IS' nothing,take(X,P,Q)).
add(set placed Q 'IS' only X,take1(X,P,Q)).

add(Z,take1(X,P,Q)):- add(Z,take(X,P,Q)).

del(position 'IS' Z,go(P)).
del(X 'IS' placed Z,take(X,P,Q)).
del(position 'IS' Z,take(X,P,Q)).
del(set placed Q 'IS' Z,take(X,P,Q)).
del(set placed P 'IS' Z,take(X,P,Q)).

del(Z,take1(X,P,Q)):- del(Z,take(X,P,Q)).

can(go(inside at L),true).
can(take(X,inside at L1,inside at L2),
    set placed inside at L1 'IS' only X &
    position 'IS' inside at L1).
can(take1(X,inside at L1, inside at L2),
    set placed inside at L2 'IS' nothing &
    set placed inside at L1 'IS' only X &
    position 'IS' inside at L1).
can(take(X,inside at L,outside),
    set placed inside at L 'IS' only X &
    position 'IS' inside at L &
    key1 'IS' placed inside at door &
    key2 'IS' placed inside at door).

given(start,set placed inside at table 'IS' nothing).
given(start,set placed inside at box1 'IS' only key1).
given(start, set placed inside at box2 'IS' only key2).
given(start, set placed inside at door 'IS' only red1).

imposs(position 'IS' P & position 'IS' Q & not_equal(P,Q)).

```

Execution:

The command:

```

?- plans(red1 'IS' placed outside,start).

```

runs WARPLAN after having added this database.

The answer is:

```
start ;
go(inside at door) ;
take1(red1, inside at door, inside at table) ;
go(inside at box1) ;
take(key1, inside at box1, inside at door) ;
go(inside at box2) ;
take(key2, inside at box2, inside at door) ;
go(inside at table) ;
take(red1, inside at table, outside) .
```

Comments:

The reason for the two versions of ‘take’ is that an action must have a unique set of preconditions, and the effects of ‘take’ depend on the set of objects at the destination. (See clauses ‘can’ in the logic program).

Problem 117 [Warren 1974a]

Verbal statement:

Consider a very simple computer, comprising an accumulator and an unspecified number of general purpose registers. There are just four instructions ‘load’, ‘store’, ‘add’ and ‘subtract’. To axiomatize the domain in order to use the program WARPLAN, presented in Problem 113, it is necessary to follow each instruction in an assembly language program by a comment. The comment is introduced by ‘#’ and states the value which will be in the accumulator after the instruction has been executed.

Describe such a machine and make it solve:

```
(1) acc is (c1-c2) + (c3-c4)
(2) acc is (c1-c2) + (c1-c2)
(3) acc is c1 + (c2-c3) &
    reg2 is c2-c3 &
    reg3 is c4 + c4
(4) reg1 is c1 + (c2-c3) &
    reg2 is c2-c3 &
    acc is c1
```

from the initial state: reg1 is c1
 reg2 is c2
 reg3 is c3
 reg4 is c4

Logic program:

```

:-op(250,yfx,'#').
:-op(250,yfx,'IS').
:-op(150,xfy,'+').
:-op(150,xfy,'-').
:-op(150,fx,'load').
:-op(250,fx,'add').
:-op(150,fx,'subtract').
:-op(150,fx,'store').
:-op(150,fx,'reg').

/* Machine code generation */

add(acc 'IS' V1+V2, add R # V1+V2).
add(acc 'IS' V1-V2, subtract R # V1-V2).
add(acc 'IS' V, load R # V).
add(reg R 'IS' V, store R # V).

del(acc 'IS' Z,U):- add(acc 'IS' V,U).
del(reg R 'IS' Z,U):- add(reg R 'IS' V,U).

can(load R # V, reg R 'IS' V).
can(store R # V, acc 'IS' V).
can(add R # V1+V2, reg R 'IS' V2 & acc 'IS' V1).
can(subtract R # V1-V2, reg R 'IS' V2 & acc 'IS' V1).

given(start,reg1 'IS' c1).
given(start,reg2 'IS' c2).
given(start,reg3 'IS' c3).
given(start,reg4 'IS' c4).

```

Execution:

```
(1) :- plans(acc 'IS' (c1-c2)+(c3-c4),start).
```

Answer:

```

start ;
load 3 # c3 ;
subtract 4 # c3-c4 ;
store X1 # c3-c4 ;
load 1 # c1;
subtract 2 # c1-c2 ;
add X1 # (c1-c2)+(c3-c4) .

```

```
(2) :- plans(acc 'IS' (c1-c2)+(c1-c2),start).
```

Answer:

```

start ;
load1 # c1
subtract2 # c1-c2

```

```

store X1 # c1-c2
add X1 # (c1-c2)+(c1-c2) .

(3) :- plans(reg1 'IS' c1+(c2-c3) &
            reg2 'IS' c2-c3 &
            reg3 'IS' c4+c4,start).

```

Answer:

```

start ;
load2 # c2;
subtract 3 # c2-c3 ;
store 2 # c2-c3 ;
load 1 # c1 ;
add 2 # c1+(c2-c3) ;
store 1 # c1+(c2-c3) ;
load 4 # c4;
add 4 # c4+c4 ;
store 3 # c4+c4 .

(4) :- plans(reg1 'IS' c1+(c2-c3) &
            reg2 'IS' c2-c3 &
            acc 'IS' c1,start).

```

Answer:

```

start ;
load 2 # c2;
subtract 3 # c2-c3 ;
store 2 # c2-c3 ;
load 1 # c1 ;
store X1 # c1 ;
add 2 # c1+(c2-c3) ;
store 1 # c1+(c2-c3) ;
load X1 # c1 .

```

Comments:

The first branch in WARPLAN's search space for question (4) is infinite. Interactive intervention to block this branch resulted in the above solution being found without any further assistance.

Problem 118 [Warren 1974a]

Verbal statement:

Consider the world presented in Fig.4, belonging to STRIPS1 (Fikes & Nilson 1971).

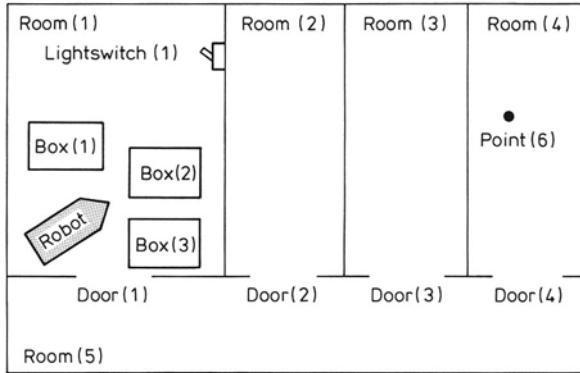


Fig.4

Describe it and use WARPLAN to achieve the goals:

- (1) status(lightswitch(1),on)
- (2) nextto(box(1),box(2))&nextto(box(2),box(3))
- (3) at(robot,point(6))
- (4) nextto(box(2),box(3))&nextto(box(3),door(1))&
status(lightswitch(1),on)&nextto(box(1),box(2))&
inroom(robot,room(2))

Logic program:

```
/* Data Base for the STRIPS1 problem */

add(at(robot,P),goto1(P,R)).
add(nextto(robot,X),goto2(X,R)).
add(nextto(X,Y),pushto(X,Y,R)).
add(nextto(Y,X),pushto(X,Y,R)).
add(status(S,on),turnon(S)).
add(on(robot,B),climbon(B)).
add(onfloor,climboff(B)).
add(inroom(robot,R2),gothru(D,R1,R2)).

del(at(X,Z),U):- moved(X,U).
del(nextto(Z,robot),U):- !,del(nextto(robot,Z),U).
del(nextto(robot,X),pushto(X,Y,R)):- !, fail.
del(nextto(robot,B),climbon(B)):- !,fail.
del(nextto(robot,B),climboff(B)):- !,fail.
del(nextto(X,Z),U):- moved(X,U).
del(nextto(Z,X),U):- moved(X,U).
del(on(X,Z),U):- moved(X,U).
del(onfloor,climbon(B)).
del(inroom(robot,Z),gothru(D,R1,R2)).
del(status(S,Z),turnon(S)).
```

```

moved(robot,goto1(P,R)).  

moved(robot,goto2(X,R)).  

moved(robot,pushto(X,Y,R)).  

moved(X,pushto(X,Y,R)).  

moved(robot,climbon(B)).  

moved(robot,climboff(B)).  

moved(robot,gothru(D,R1,R2)).  
  

can(goto1(P,R), locinroom(P,R)&inroom(robot,R)&onfloor).  

can(goto2(X,R), inroom(X,R)&inroom(robot,R)&onfloor).  

can(pushto(X,Y,R), pushable(X)&inroom(Y,R)&inroom(X,R)&  

    nextto(robot,X)&onfloor).  

can(turnon(lightswitch(S)), on(robot,box(1))&  

    nextto(box(1),lightswitch(S))).  

can(climbon(box(B)), nexto(robot,box(B))&onfloor).  

can(climboff(box(B)), on(robot,box(B))).  

can(gothru(D,R1,R2), connects(D,R1,R2)&inroom(robot,R1)&  

    nextto(robot,D)&onfloor).  
  

always(connects(D,R1,R2)):- connects1(D,R1,R2).  

always(connects(D,R2,R1)):- connects1(D,R1,R2).  

always(inroom(D,R1)):- always(connects(D,R0,R1)).  

always(pushable(box(N))).  

always(locinroom(point(6),room(4))).  

always(inroom(lightswitch(1),room(1))).  

always(at(lightswitch(1),point(4))).  
  

connects1(door(N), room(N), room(5)):- range(N,1,4).  
  

range(M,M,N).  

range(M,L,N):- not_equal(L,N), L1 is L+1, range(M,L1,N).  
  

given(strips1,at(box(N),point(N))):- range(N,1,3).  

given(strips1,at(robot,point(5))).  

given(strips1,inroom(box(N),room(1))):- range(N,1,3).  

given(strips1,inroom(robot,room(1))).  

given(strips1,onfloor).  

given(strips1,status(lightswitch(1),off)).
```

Execution:

(1) :- plans(status(lightswitch(1),on),start).

Answer:

```

start ;
goto2(box(1),room(1)) ;
pushto(box(1),lightswitch(1),room(1)) ;
climbon(box(1)) ;
turnon(lightswitch(1)) .
```

```
(2) :- plans(nextto(box(1),box(2) &
                     nexto(box(2),box(3)),start).
```

Answer:

```
start ;
goto2(box(2),room(1)) ;
pushto(box(2),box(3),room(1)) ;
goto2(box(1),room(1)) ;
pushto(box(1),box(2),room(1)) .
```

```
(3) :- plans(at(robot,point(6)),start).
```

Answer:

```
start ;
goto2(door(1),room(1)) ;
gothru(door(1),room(1),room(5)) ;
goto2(door(4),room(5)) ;
gothru(door(4),room(5),room(4)) ;
goto1(point(6),room(4)) .
```

```
(4) :- plans(nextto(box(2),box(3)) &
             nextto(box(3),door(1)) &
             status(lightswitch(1),on) &
             nextto(box(1),box(2)) &
             inroom(robot,room(2)),start).
```

Answer:

```
start ;
goto2(box(3),room(1)) ;
pushto(box(3),door(1),room(1)) ;
goto2(box(2),room(1)) ;
pushto(box(2),box(3),room(1)) ;
goto2(box(1),room(1)) ;
pushto(box(1),lightswitch(1),room(1)) ;
climbbox(1) ;
turnon(lightswitch(1)) ;
climbboxoff(1) ;
goto2(box(1),room(1)) ;
pushto(box(1),box(2),room(1)) ;
goto2(door(1),room(1),room(5)) ;
gothru(door(1),room(1),room(5)) ;
goto2(door(2),room(5)) ;
gothru(door(2),room(5),room(2)) .
```

Problem 119 [Warren 1975b]

Verbal statement:

Consider a world consisting of the following objects, facts and actions:

Objects:

```

numbers,      N = <1,2,...etc>
directions,   D = <left,right>
wheels,       W = <wheel1, wheel2,...,wheelN>
axles,        A = <axle1,...,axle N>
endofaxles,   E = <D1 endof A1> where D1  D and A1  A
holes,        H = <hole1,...,hole N>
vice,         V = <vice>
```

Facts:

Static, D1 is opposite of D2,	where D1,D2 D
Dynamic, W1 is attached to E1,	where W1 W and E1 E
Dynamic, A1 is thru H1,	where A1 A and H1 H
Dynamic, E1 points D1,	where E1 E and D1 D
Dynamic, carbody is blocked to D1,	where D1 D
Dynamic, carbody is unblocked to D1,	where D1 D
Dynamic, W1 + A1 is clamped,	where W1 W and A1 A
Dynamic, W1+A1+H1+V1 is free,	where W1 W,A1 A,H1 H and V1 V

Actions:

insert E1 into W1,	where E1 E and W1 W
push W1 from D1 to D2 onto E1 in H1,	where W1 W,H1 H,E1 E
slide E1 into H1 from D1,	where E1 E,H1 H and D1 D
block carbody to D1,	where D1 D
unblock carbody to D1,	where D1 D
clamp W1,	where W1 W
unclamp W1+A1,	where W1 W and A1 A

the actions may be pictured as follows (Figs. 5 and 6):

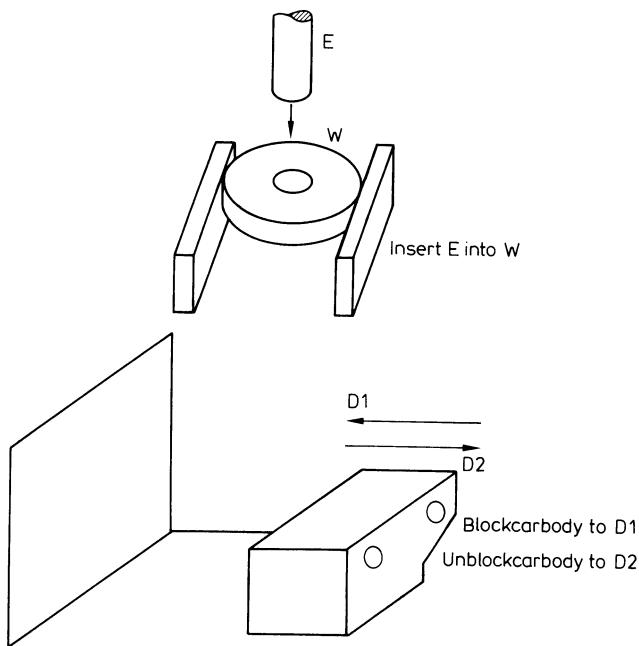


Fig. 5

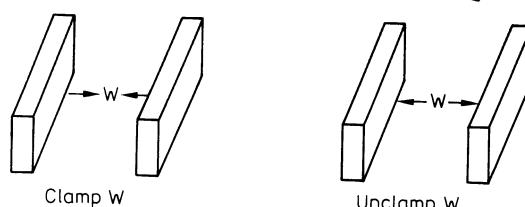
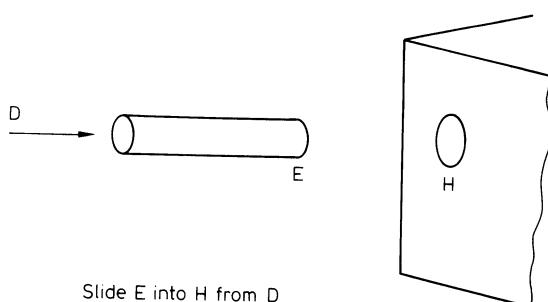
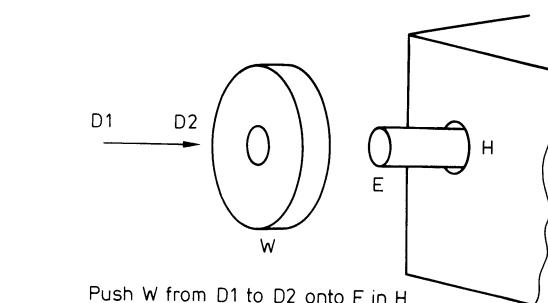


Fig. 6

Describe this world as a database in order to use WARPLAN to solve car assembly problems.

Logic program:

```
/* Operators definition */

:-op(600,fx,'insert').
:-op(600,fx,'push').
:-op(600,fx,'slide').
:-op(600,fx,'clamp').
:-op(600,fx,'unclamp').
:-op(600,fx,'block_car_body_to').
:-op(600,fx,'unblock_car_body_to').
:-op(600,fx,'car_body_is_unblocked_to').
:-op(600,fx,'car_body_is_blocked_to').

:-op(600,xf,'is_free').
:-op(600,xf,'is_clamped').

:-op(500,xfy,'is_thru').
:-op(500,xfy,'points').
:-op(500,xfy,'is_opposite_of').
:-op(500,xfy,'is_attached_to').

:-op(400,yfx,'into').
:-op(400,yfx,'from').
:-op(400,yfx,'to').
:-op(400,yfx,'onto').
:-op(400,yfx,'in').

:-op(300,xfy,'end_of').

:-op(200,fx,'wheel').
:-op(200,fx,'axle').
:-op(200,fx,'hole').

/* Definition of the goal state */

goal(T):- plans(axle A1 is_thru hole1 &
               axle A2 is_thru hole2 &
               wheel W1 is_attached_to left end_of axle A1 &
               wheel W2 is_attached_to left end_of axle A2 &
               wheel W3 is_attached_to right end_of axle A1 &
               wheel W4 is_attached_to right end_of axle A2,T).

/* Data base */

add(W is_attached_to E, insert E into W).
add(W is_attached_to E, push W from D1 to D2 onto E in H).
add(A is_thru H, slide D1 end_of A into H from D2).
add(wheel W is_clamped, clamp wheel W).
```

```

add(axle A is_clamped, insert D end_of axle A into W).
add(wheel W is_free, unclamp wheel W).
add(axle A is_free, unclamp axle A).
add(vice is_free, unclamp X).
add(car_body_is_blocked_to D, block_car_body_to D).
add(car_body_is_unblocked_to D, unblock_car_body_to D).
add(D1 end_of A points D, slide D1 end_of A into H from D2):-
    always(D is_opposite_of D2).
add(d end_of A points D2, slide D1 end_of A into H from D2):-
    always(D is_opposite_of D1).

can(insert D end_of axle A into wheel W, axle A is_free &
    D end_of axle A is_free &
    wheel W is_clamped).

can(push wheel W from D1 to D2 onto D end_of axle A in hole H,
    wheel W is_free &
    D end_of axle A is_free &
    axle A is_thru hole H &
    D end_of axle A points D1 &
    car_body_is_unblocked_to D1 &
    D2 is_opposite_of D1 &
    car_body_is_blocked_to D2).

can(slide D1 end_of axle A into hole H from D2,
    axle A is_free &
    D1 end_of axle A is_free &
    hole H is_free &
    car_body_is_unblocked_to D2).

can(clamp wheel W, wheel W is_free & vice is_free).
can(unclamp X, X is_clamped).
can(block_car_body_to D, true).
can(unblock_car_body_to D, true).

del(X is_free, U):- add(X is_clamped, U).
del(X is_free, U):- add(X is_attached_to Z, U).
del(X is_free, U):- add(Z is_attached_to X, U).
del(A is_free, slide D1 end_of A into H from D2).
del(H is_free, slide E into H from D2).
del(vice is_free, clamp W).
del(car_body_is_unblocked_to D, block_car_body_to D).
del(car_body_is_blocked_to D, unblock_car_body_to D).
del(Z is_clamped, unclamp Z).
del(W is_clamped, insert E into W).

imposs(axle A is_free & axle A is_thru hole H).
imposs(D end_of A is_free & W is_attached_to D end_of A).
imposs(wheel W is_free & wheel W is_attached_to E).
imposs(wheel W is_free & wheel W is_clamped).
imposs(hole H is_free & A is_thru hole H).

```

```

imposs(axle A is_free & axle A is clamped).
imposs(vice is_free & X is_clamped).
imposs(car_body_is_blocked_to D & car_body_is_unblocked_to D).

always(true).
always(left is_opposite_of right).
always(right is_opposite_of left).

```

Problem 120 [Warren 1975b]

Verbal statement:

Consider the initial state of the car assembly process as it is pictured in Fig. 7.

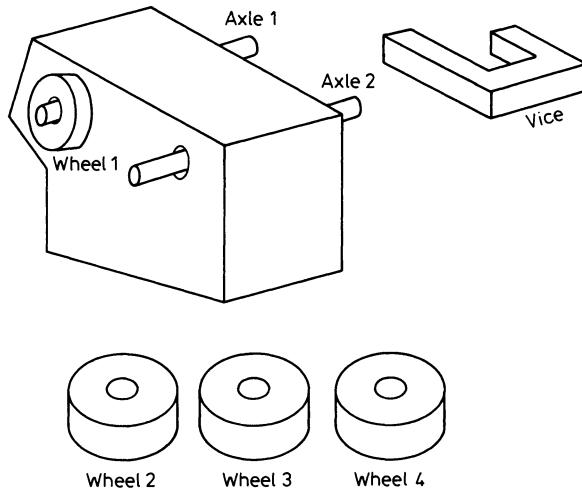


Fig. 7

Describe this initial state in order to use the preceding problem and the program WARPLAN to solve the rest of the assembly process.

Logic program:

This state is called the “middle state” just because a small part of the car is already assembled when the process starts.

```

/* Description of the middle state */

given(middle,wheel W is_free):- member(W,[2,3,4]).
given(middle,wheel 1 is_attached_to left end_of axle 1).
given(middle,axle N is_thru hole N):- member(N,[1,2]).
given(middle, D end_of axle A points D):-
    member(D,[left,right]),
    member(A,[1,2]).

```

```

given(middle,right end_of axle A is_free):- member(A,[1,2]).  

given(middle, left end_of axle 2 is_free).  

given(middle, vice is_free).  

given(middle,car_body_is_unblocked_to D):-  

    member(D,[left,right]).  

member(A,[A|_]).  

member(A,[_|B]):- member(A,B).

```

Execution:

To run the program WARPLAN with such an initial state and such a database you just need to give the command: “:- goal(middle).”.

The answer is:

```

block_car_body_to right ;  

push wheel2 from left to right onto left end_of axle 2  

in hole 2 ;  

unblock_car_body to right ;  

block_car_body to left ;  

push wheel 3 from right to left onto right end_of axle 1  

in hole 1 ;  

push wheel 4 from right to left onto right end_of axle 2  

in hole 2 .

```

Problem 121 [Warren 1975b]

Verbal statement:

Now consider the initial state of the car assembly process as in Fig.8.

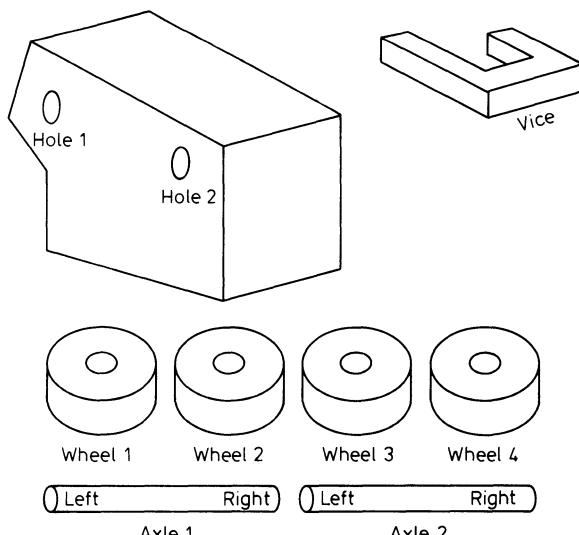


Fig.8

Solve the preceding problem considering this initial state and taking into account the program WARPLAN.

Logic program:

```
/* Description of the start state */

given(start,wheel W is_free):- member(W,[1,2,3,4]).
given(start,axle A is_free):- member(A,[1,2]).
given(start,D end_of axle A is_free):- member(D,[left,right]),
                                         member(A,[1,2]).
given(start,hole H is_free):- member(H,[1,2]).
given(start,vice is_free).
given(start,car_body_is_unblocked_to D):-
                                         member(D,[left,right]).

member(A[A|_]).  

member(A,[_|B]):- member(A,B).
```

Execution:

```
?-goal(start).

slide left end_of axle 1 into hole 1 from left ;
slide left end_of axle 2 into hole 2 from left ;
block_car_body_to left ;
push wheel 1 from right to left onto left end_of axle 1
in hole 1 ;
push wheel 2 from right to left onto left end_of axle 2
in hole 2 ;
unblock_car_body_to left ;
block_car_body_to right ;
push wheel 3 from left to right onto right end_of axle 1
in hole 1 ;
push wheel 4 from left to right onto right end_of axle 2
in hole 2 .
```

Chapter 14 Seeing with Prolog

Seeing and understanding what is seen is quite suitable for deep problem solving and program organization. This chapter introduces an exercise illustrating facts about vision processing and knowledge structures that are relevant for planning.

Problem 122 [Warren 1975b]

Verbal statement:

Write a program that receives a scene as input and produces an interpretation of the scene as output.

Consider a scene composed of a certain disposition of solids. The input for the program is a description of the solids in terms of their edges, and the output will be an interpretation in terms of what solids constitute the scene and how their faces are oriented.

Consider only vertical edges, positive and negative slope edges, and that a face may be left, right or horizontal.

Apply your program to the scene:

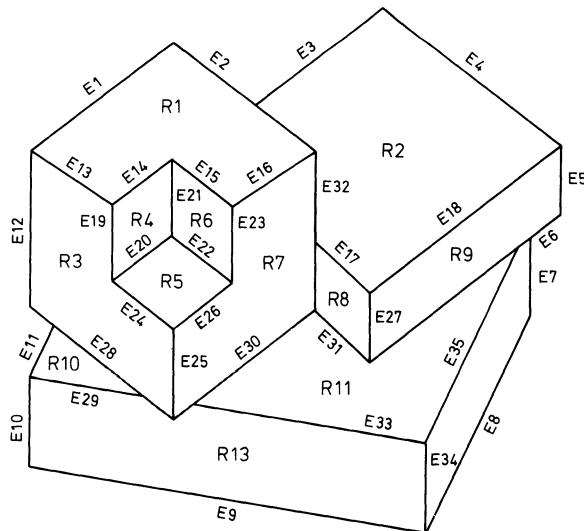


Fig. 1

where the edges and the faces are numbered (E1,...,E35 and R1,...R13, respectively) in an arbitrary order.

Suggestion: 1) When defining the scene begin with the edges that limit it, that is, begin with the boundary edges.

- 2) Define a set of rules characterizing the different kinds of edges in terms of convexity or concavity of the regions they establish.
- 3) Label the contour edges clockwise.

Logic program:

```

:- op(500,xfy,';').
:- op(300,xfy,':').

goal:- solution(X),nl,nl,write(X),nl,nl.

/* Set of interpretation rules */

vertical_edge(left:B,right:B,positive).
vertical_edge(right:B1,left:B2,negative).
vertical_edge(right:B,X,down).
vertical_edge(X,left:B,up).

positive_edge(horizontal:B,right:B,positive).
positive_edge(right:B1,horizontal:B2,negative).
positive_edge(X,horizontal:B,up).
positive_edge(right:B,X,down).

negative_edge(left:B,vertical:B,positive).
negative_edge(vertical:B1,left:B2,negative).
negative_edge(vertical:B,X,down).
negative_edge(X,vertical:B,up).

/* Defining the scene */

solution(R1;R2;R3;R4;R5;R6;R7;R8;R9;R10;R11;R12;R13):-
    positive_edge(background,R1,E1),
    negative_edge(R1,background,E2),
    positive_edge(background,R2,E3),
    negative_edge(R2,background,E4),
    vertical_edge(R9,background,E5),
    positive_edge(R9,background,E6),
    vertical_edge(R12,background,E7),
    positive_edge(R12,background,E8),
    negative_edge(background,R13,E9),
    vertical_edge(background,R13,E10),
    positive_edge(background,R10,E11),
    vertical_edge(background,R3,E12),
    negative_edge(R3,R1,E13),
    positive_edge(R1,R4,E14),
    negative_edge(R6,R1,E15),
    positive_edge(R1,R7,E16),

```

```

negative_edge(R8,R2,E17),
positive_edge(R2,R9,E18),
vertical_edge(R3,R4,E19),
positive_edge(R4,R5,E20),
vertical_edge(R4,R6,E21),
negative_edge(R5,R6,E22),
vertical_edge(R6,R7,E23),
negative_edge(R3,R5,E24),
vertical_edge(R3,R7,E25),
positive_edge(R5,R7,E26),
vertical_edge(R8,R9,E27),
negative_edge(R10,R3,E28),
negative_edge(R13,R10,E29),
positive_edge(R7,R11,E30),
negative_edge(R11,R8,E31),
vertical_edge(R7,R8,E32),
negative_edge(R13,R11,E33),
vertical_edge(R13,R12,E34),
positive_edge(R11,R12,E35).

```

Execution:

?- goal.

```

horizontal:X1 ; horizontal:X2 ; left:X1 ; right:X1 ;
horizontal:X1 ; left:X1 ; right:X1 ; left:X2 ;
right:X2 ; horizontal:X3 ; horizontal:X3 ; right:X3 ;
left:X3

```

This interpretation corresponds to the scene:

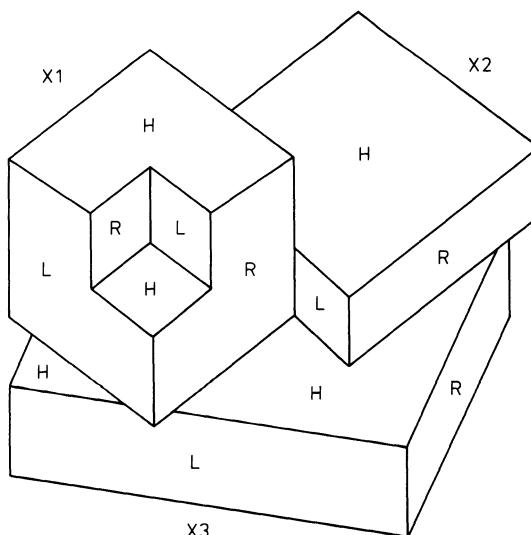


Fig. 2

Chapter 15 Engineering Grammars with Prolog

Within this chapter we explain how to write natural language grammars expressed in clausal logic. The notion of definite clause grammars (DCG's) is behind this as a paradigm for organizing the knowledge required in understanding, which allows efficient parsing. In some exercises syntactic and semantic knowledge are applied concurrently to fill a logical structure that contains all the information for semantic interpretation. The syntax plays a guiding role in the processing, and the logical structure describes the meaning of the corresponding sentence by assembling well-formed formulae belonging to a logical system for representing a subset of the natural language considered.

Problem 123

Verbal statement:

Transform a natural language sentence, taken as a string of characters with some spaces between them, into a list whose elements are its words.

- Suggestion:
- 1) Consider the following cut marks : ‘space’, ‘?’, ‘:’, ‘!’, ‘;’, ‘line _ feed’ and ‘CR’.
 - 2) Suppose the sentences may end with ‘?’, ‘:’, or ‘!’.
 - 3) Knowing that PROLOG cannot read word by word, you have to imagine a character by character transformation.

Logic program:

```
sentence(F):- get(C),words(C,F).  
words(C,[P|Ps]):- letter(C),word(C,C1,L),  
name(P,L),words(C1,Ps).  
words(44,[' '|Ps]):- get(C1),words(C1,Ps).  
words(63,[?]).  
words(46,[.]).  
words(33,[!]).  
words(_,P):- get(C),words(C,P).  
word(C,C1,[C|Cs]):- get0(C2),(letter(C2),word(C2,C1,Cs);  
C1=C2,Cs=[]).
```

```

letter(32):- !, fail.          /* space */
letter(63):- !, fail.          /* ? */
letter(46):- !, fail.          /* . */
letter(33):- !, fail.          /* ! */
letter(44):- !, fail.          /* , */
letter(10):- !, fail.          /* line_feed */
letter(13):- !, fail.          /* CR */
letter(_).

```

Comments:

Notice the difference between the use of the built-in “get” and “get0” directly inherited from the fact that “get” skips blank characters (space, tab) while get0 really gets everything.

Problem 124

Verbal statement:

In the preceding problem, make the suitable modification in order that capital letters might be transformed into small letters, knowing that the internal codes of capital letters go from 64 to 90 and that the codes of the corresponding small letter can be obtained by adding 32.

Logic programs:

```

/* Program 1

Define two new predicates "get/2" and "get0/2" as follows: */

get(C,A):- get(C), (C=<90,C>=64, A is C + 32 ; A=C).
get0(C,A):- get0(C), (C=<90,C>=64, A is C+32; A=C).

```

/ Replace the usual calls to "get" and "get0" by new calls of the above predicates, the resulting program will be (we will write only the new clauses): */*

```

sentence(F):- get(C,A), words(A,F).

words(44,[' '|Ps]):- get(C1,A), words(A,Ps).
words(_,P):- get(C,A), words(A,P).
word(C,C1,[C|Cs]):- get0(C2,A,(letter(A),word(A,C1,Cs);
C1=A, Cs=[]).


```

/* Program 2

*Redefine the predicate "letter" as follows: */*

```

letter(32,_):- !,fail.          /* space */
letter(63,_):- !,fail.          /* ? */
letter(46,_):- !,fail.          /* . */

```

```

letter(33,_):- !,fail.          /*      !      */
letter(44,_):- !,fail.          /*      ,      */
letter(10,_):- !,fail.          /* line_feed */
letter(13,_):- !,fail.          /*      CR      */
letter(L,NL):- (L=<90,L>=64,NL is L+32; L=NL).

/* Then we replace the calls of the old "letter" by calls to the
new "letter": */

words(C,[P|Ps]):- letter(C,NC),word(NC,C1,L),
                 name(P,L),words(C1,Ps).

word(C,C1,[C|Cs]):- get0(C2),(letter(C2,NC2),word(NC2,C1,Cs);
                      C1 = C2, Cs = []).

```

Problem 125

Verbal statement:

Suppose you have a list L of atoms, for instance proper nouns. Output that list word by word, each word with the first letter as a capital, and separated from the previous by a comma, the last one separated by “and”.

Logic program:

```

output([N]):-capitals(N,N1),write(N1),write('.').
output([N,N1]):- capitals(N,N2),write(N2),write(' and '),
               output([N1]). 
output([N|Ns]):- capitals(N,N1),write(N1),write(', '),
               output(Ns).

capitals(X,X1):- name(X,[A|L]), A>=97,
                 B is A-32, name(X1,[B|L]). 

capitals(X,X).

```

Comments: 1) the second clause of “capitals” is only needed when the word already has a capital letter.

2) output of the list [john,mary,james] will be:
“John, Mary and James.”.

Problem 126 [Warren 1977b]

Verbal statement:

Consider the following BNF grammar and write it as a PROLOG program.

```

<statement>      ::= <name>:=<expr> |
                      IF <test> THEN <statement> ELSE <statement> |
                      WHILE <test> DO <statement> |
                      READ <name> |
                      WRITE <expr> |
                      (<statements>)

<test>          ::= <expr><comparison op><expr>
<expr>           ::= <expr><op 2><expr 1> |
                      <expr 1>
<expr 1>         ::= <expr 1><op 1><expr 0> |
                      <expr 0>
<expr 0>         ::= <name> |
                      <integer> |
                      (<expr>)

<comparison op> ::= = | < | > | =< | >= | != \=
<op 2>          ::= * | /
<op 1>          ::= + | -

```

Logic program:

```

program(Z0,Z,X) :- statements(Z0,Z,X).

statements(Z0,Z,X) :- statement(Z0,Z1,X0),
                     restatements(Z1,Z,X0,X).

restatements((';'.Z0),Z,X0,(X0;X)) :- statements(Z0,Z,X).
restatements(Z,Z,X,X).

statement((V.'=:'.Z0),Z,assign(name(V),Expr)) :-
    atom(V), expr(Z0,Z,Expr).
statement((if.Z0),Z,if(Test,Then,Else)) :-
    test(Z0,(then.Z1),Test),
    statement(Z1,(else.Z2),Then),
    statement(Z2,Z,Else).
statement((while.Z0),Z,while(Test,Do)) :-
    test(Z0,(do.Z1),Test),
    statement(Z1,Z,Do).
statement((read.V.Z),Z,read(name(V))) :- atom(V).
statement((write.Z0),Z,write(expr)) :- expr(Z0,Z,Expr).
statement((('.'.Z0),Z,S) :- statements(Z0,(').Z),S).

test(Z0,Z,test(Op,X1,X2)) :-
    expr(Z0,(Op.Z1),X1), comparisonop(Op),
    expr(Z1,Z,X2).

expr(Z0,Z,X) :- subexpr(2,Z0,Z,X).

subexpr(N,Z0,Z,X) :- N>0, N1 is N-1,
                     subexpr(N1,Z0,Z1,X0),
                     restexpr(N,Z1,Z,X0,X).

```

```

subexpr(0,(X.Z),Z,name(X)) :- atom(X).
subexpr(0,(X.Z),Z,const(X)) :- integer(X).
subexpr(0,((''.Z0),Z,X) :- subexp(2,Z0,(')'.Z),X).

restexpr(N,(Op.Z0),Z,X1,X) :- op(N,Op), N1 is N-1,
    subexpr(N1,Z0,Z1,X2),
    restexp(N,Z1,Z,expr(Op,X1,X2),X).
restexpr(N,Z,Z,X,X).

comparisonop(=).
comparisonop(<).
comparisonop(>).
comparisonop(=<).
comparisonop(>=).
comparisonop(\=).

op(2,*).           op(1,+).
op(2,/).          op(1,-).

```

Problem 127 [L.Pereira et al. 1978]

Verbal statement:

Construct a simple grammar which parses an arithmetic expression (made up of digits and operators) and computes its value. Consider “ $-2+3*5+1$ ” as an example of an arithmetic expression.

Logic program:

```

expr(Z) --> term(X), "+", expr(Y), {Z is X+Y}.
expr(Z) --> term(X), "-", expr(Y), {Z is X-Y}.
expr(Z) --> term(Z).

term(Z) --> number(X), "*", term(Y), {Z is X*Y}.
term(Z) --> number(X), "/", term(Y), {Z is X/Y}.
term(Z) --> number(Z).

number(C) --> "+", number(C).
number(C) --> "-", number(X), {C is -X}.
number(X) --> [C], {"0"=<C, C<"9", X is C-"0"}.

```

Execution:

The question: `?-expr(Z, "-2+3*5+1", []).`
 will compute `Z=14`

Problem 128 [Warren 1974b]

Verbal statement:

Write a program able to translate the functional LISPish notation into the usual PROLOG notation, for example translate

```
:rev([X|L]) = :app(:rev(L),[X]) &
:rev([]) = [] &
```

```
:app([X|L1],L2) = [X]:app(L1,L2) &
:app([],L) = L
```

into

```
rev([X|L1],L) :- rev(L1,L2), app(L2,[X],L).
rev([],[]).
```

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

Logic program:

```

:-op(300,yfx,&).
:-op(200,yfx,=).
:-op(100,fx,:).

/* Translation from functional notation into
   PROLOG notation */

let(P):- let1(P).
let(_).

let1(P&Q):- let1(P).
let1(P&Q):- !,let1(Q).
let1(:T0=V0):- T0=..[F|A0],trans(V0,V,[],C),
               app(A0,[V],A1), T=..[F|A1],
               write(T),output(C),nl,fail.

trans(T0,T0,C0,C0):- var(T0,!).
trans(:T0,V,C0,C):- !, T0=..[F|A0],translist(A0,A,[T|C0],C),
                   app(A,[V],A1), T=..[F|A1].
trans(T0,T,C0,C):- T0=..[F|A0],translist(A0,A,C0,C),
                   T=..[F|A]. 

translist([T0|A0],[T|A],C0,C):- translist(A0,A,C0,C1),
                                trans(T0,T,C1,C).

translist([],[],C0,C0).

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

output([]):- write('.'),nl,!.
output(X):- write(':- '),output1(X).
```

```
output1([A]):- write(A), write('.'), nl, !.
output1([A|B]):- write(A), write(', '), output1(B).
```

Execution:

To perform the program give the command “:-let(P).” where P must be what you want to translate into PROLOG, for example:

```
:-let(  : rev([X|L]) = :app( :rev(L),[X]) &
      : rev([]) = [] &
      : app([X|L1],L2) = [X]:app(L1,L2) &
      : app([],L) = L    ).
```

would have the following result:

```
rev([X|L],L1):- rev(L,L2), app(L2,[X],L1).
rev([],[]).

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

Problem 129

Verbal statement:

Write a program able to translate the PROLOG notation into pseudo-English.

Consider the following definitions as a start point:

P :- Q	means	P if Q
P , Q	means	P and Q
P ; Q	means	P or Q

The program must also be able to accept new definitions from the user and to translate English into PROLOG notation.

Logic program:

```
:-op(300,xf,?).

process(T-D):- !, asserta(definition(T,D)).
process([T1!T2]?):- retrovert(T,[T1!T2]),!, nl, write(T).
process(T):- translate(T,D), nl, output(D), !.
process(_):- nl, write(untranslatable), nl.

begin:- asserta(definition(X,X)),
        asserta(definition((P;Q),[P,or,Q])),
        asserta(definition((P:-Q),[P,if,Q])),
        asserta(definition((P,Q),[P, and,Q])),
        asserta(definition((P, and,Q),[P,',',Q])),
        repeat, nl, nl, read(T), (T=stop; process(T), fail).
```

```

/* Retroverser process */

retrovert(T,D):- definition(T1,D),!, T1=..[N|L],
    retrovert_args(Args,L), T=..[N|Args]. 

retrovert_args([A|Args],[L|Ls]):- retrovert(A,L),
    retrovert_args(Args,Ls).

retrovert_args([],[]).

/* Translation process */

translate(X,X):- var(X),!.
translate(T,D):- T=..[N|Args], args(Args,L),
    T1=..[N|L], definition(T1,D).

args([A|Args],[L|Ls]):- translate(A,L), args(Args,Ls).
args([],[]).

/* Output process */

output([X|S]) :- var(X), write(X), write(' '), output(S).
output([[X]|S]) :- output([X|S]).
output([[X|L]|S]) :- output([X|L]),
    output(L).
output([X|S]) :- write(X), write(' '), output(S).
output([]).

```

Execution:

```

| ?- begin.

| : p;q.

p or q

| : p:- q.

p if q

| : p,q.

p and q

| : p,q,r,s.

p and q and r and s

| : p,q;r;t:-u.

p and q or r or t if u

| : [p,or,q]? .

p;q

| : [p,if,q]? .

p:-q

```

```

l: [p, and, q]? .
p, q
l: [p, and, q, or, t]? .
[p, andl(q;t)]
l: (arvore)-[tree].
l: arvore.
tree
l: [tree]? .
arvore
l: (a+b)-[a, plus, b].
l: a+b.
a plus b
l: [a, plus, b]? .
a+b
l: kdktjfu.
untranslatable
l: a+b+c.
untranslatable
l: (a+b+c)-[a, plusb, b, plusb, c].
l: a+b+c.
a plusb b plusb c
l: stop.

```

Problem 130

Verbal statement:

A database with personal information is described as follows:

<name, address, age, profession, driving_license>

Each elementary relation built upon the above data is associated with an atomic formula.

Natural language queries to that database are translated into logical notation, defined by atomic formulae, connectors ‘and’ and ‘or’, existential and universal quantifiers on typed variables.

Write a program to evaluate the logical formulae attached to sample queries.

Logic program:

```
/* Data base */

person(helder,lisbon,39,scientist,yes).
person(joao,sintra,33,scientist,yes).
person(francisco,alges,17,student,no).
person(monica,alges,15,student,no).
person(jose,alges,39,designer,no).
person(ronald,philadelphia,34,scientist,yes).
person(luis,carapica,36,professor,yes).

/* Types */

type(X,name):- name(X).
type(X,address):- address(X).
type(X,age):- age(X).
type(X,profession):- profession(X).
type(X,boolean):- boolean(X).

name(helder).      address(lisbon).      age(39).
name(joao).         address(sintra).      age(33).
name(francisco).   address(alges).        age(15).
name(monica).       address(phiadelphia). age(34).
name(jose).         address(carapica).    age(36).
name(ronald).
name(luis).

profession(scientist).   boolean(yes).
profession(student).     boolean(no).
profession(designer).
profession(professor).

/* Relations */

relation(address(X,Y)).
relation(age(X,Y)).
relation(license(X,Y)).
relation(profession(X,Y)).
relation(older(X,Y)).
relation(younger(X,Y)).
relation(different(X,Y)).

/* Evaluation of relations */

address(X,Y):- person(X,Y,_,_,_).
age(X,Y):- person(X,_,Y,_,_).
profession(X,Y):- person(X,_,_,Y,_).
license(X,Y):- person(X,_,_,_,Y).
```

```

older(X,Y):- person(X,_,A,_,_),  

            person(Y,_,B,_,_),  

            A>B.  

younger(X,Y):- person(X,_,A,_,_),  

              person(Y,_,B,_,_),  

              B>A.  

different(X,Y):- X=\=Y.  

true(P):- relation(P),P.  

true(not(P)):- not(true(P)).  

true(and(P,Q)):- true(P),true(Q).  

true(or(P,Q)):- true(P);true(Q).  

true(exists(X,T,P)):- type(X,T),true(P).  

true(all(X,T,P)):- not(true(exists(X,T,not(P)))).  

/* Negation */  

not(P):- P,! ,fail.  

not(_).  

/* Answer to queries */  

answer(Q):- eval(X,Q),  

           write('Answer: '),write(X),nl.  

eval(X,query(X,T,P)):- type(X,T),true(P).

```

Execution:

“At which address does Francisco live?”

```
?- answer(query(X,address,address(francisco,X))).
```

“Does Luis has a driving license?”

```
?- answer(query(X,boolean,license(luis,X))).
```

“Which are the addresses having at least one person aged less than 30 years old and having a driving license?”

```
?- answer(query(X,address,exists(Y,name, and(address(Y,X),  

           and(age(Y,A), and(younger(A,30),license(Y,yes))))))).
```

Problem 131

Verbal statement:

The parsing problem is formulated in the following way: “given a grammar and an initial string of words demonstrate that the string is a sentence” (Knuth 1971).

Construct a parse tree for verifying whether the following parsed string

The new author writes a paper
is grammatical.

Logical program:

```

sentence(S0,S) :- noun_phrase(S0,S1),
                 verb(S1,S2),
                 complements(S2,S).

noun_phrase(S0,S) :- article(S0,S1),
                   adjective(S1,S2),
                   common_noun(S2,S).

noun_phrase(S0,S) :- article(S0,S1),
                   common_noun(S1,S).

complements(S0,S) :- noun_phrase(S0,S).

verb(S0,S) :- connects(S0,writes,S).

article(S0,S) :- connects(S0,the,S).
article(S0,S) :- connects(S0,a,S).
adjective(S0,S) :- connects(S0,new,S).

common_noun(S0,S) :- connects(S0,paper,S).
common_noun(S0,S) :- connects(S0,author,S).

connects(1,the,2).
connects(2,new,3).
connects(3,author,4).
connects(4,writes,5).
connects(5,a,6).
connects(6,paper,7).

```

Execution:

```

?- sentence(1,7).
yes

```

Comments:

The parse is the path in the tree of goal statements from the root (initial sentence) to the last statement.

The specification of the parsing problem contains the transcription of a CFG and a representation of the string of words:

```

sentence --> noun_phrase,verb,complements.
noun_phrase --> article,adjective,common_noun.
noun_phrase --> article,common_noun.
complements --> noun_phrase.

verb --> [writes].
article --> [the].
article --> [a].
adjective --> [new].
common_noun --> [author].
common_noun --> [paper].

```

The arguments of the predicate represent the beginning and the end points in the string of a phrase for that non-terminal.

The first clause means “a sentence extends from S0 to S if there is a noun phrase from S0 to S1, a verb from S1 to S2 and complements from S2 to S”.

We use a 3-place predicate, ‘connects’, to represent terminal symbols in rules, where ‘connects(S1,T,S2)’ means “terminal symbol T lies between points S1 and S2 in the string”.

Problem 132

Verbal statement:

Write a program to build up contractions between prepositions and determiners in the Portuguese language. For example, the contraction of preposition “de” (of) with determiner “o” (the) is “do” (of).

Logic program:

```

contraction(X,X2):- preposition(X,X1), determiner(X,X2).

preposition(X,X1):- de(X,X1).
preposition(X,X1):- sobre(X,X1).
preposition(X,X1):- em(X,X1).

determiner([o|X],X).
determiner([a|X],X).

de(X,[o|X1]):- do(X,X1).
de(X,[a|X1]):- da(X,X1).
de([d,e|X],X).

em(X,[o|X1]):- no(X,X1).
em(X,[a|X1]):- na(X,X1).
em([e,m|X],X).

sobre([s,o,b,r,e|X],X).

do([d,o|X],X).
da([d,a|X],X).

no([n,o|X],X).
na([n,a|X],X).

```

Problem 133

Verbal statement:

Write a simple grammar G1 for analyzing the sentences:

"the giraffe dreams"
"the giraffe eats apples"

(This problem was suggested by Robert Kowalski).

Logic program:

```
sentence(X, Y) :- noun_phrase(X, U), verb_phrase(U, Y).

noun_phrase(X, Y) :- determiner(X, U), noun(U, Y).
noun_phrase(X, Y) :- noun(X, Y).

verb_phrase(X, Y) :- iverb(X, Y).
verb_phrase(X, Y) :- tverb(X, U), noun_phrase(U, Y).

determiner([the|Y], Y).

noun([giraffe|Y], Y).
noun([apples|Y], Y).

iverb([dreams|Y], Y).

tverb([dreams|Y], Y).
tverb([eats|Y], Y).
```

Problem 134

Verbal statement:

Reconsider the previous grammar G1 in order to deal with the following context-sensitive aspects:

- 1) the number of a noun phrase agrees with that of the corresponding verb phrase;
- 2) the number of a noun phrase need not be determined by the noun only (this fish–these fish) and not by the ‘th-word’ only (the giraffe–the giraffes), but number is a feature of the entire noun phrase;
- 3) a noun phrase need not have a determiner (giraffes dream) presumably provided that the noun phrase is plural.

(This problem was suggested by Robert Kowalski).

Logic program:

```
sentence(X, Y) :- np(N, X, U), vp(N, U, Y).

np(N, X, Y) :- th(N, X, U), noun(N, U, Y).
np(plu, X, Y) :- noun(plu, X, Y).
```

```

vp(N,X,Y):- verb(i,N,X,Y).
vp(N,X,Y):- verb(t,N,X,U),np(M,U,Y).

noun(sin,[U|Y],Y):- npr(U,V).
noun(plu,[V|Y],Y):- npr(U,V).

verb(T,sin,[U|Y],Y):- conj(T,U,V).
verb(T,plu,[V|Y],Y):- conj(T,U,V).

npr(giraffe,giraffes).
npr(fish,fish).
npr(dream,dreams).

conj(T,dreams,dream).
conj(T,eats,eat).

th(N,[the|Y],Y).
th(sin,[this|Y],Y).
th(plu,[these|Y],Y).

```

Problem 135

Verbal statement:

Consider a simplified grammar which parses English sentences of the library problem domain and produces their corresponding meaning (logical structures).

Take as an example the following question: "Does Hodges write for Penguin?"

Logic program:

```

go:- read(S), sentence(LS,S,[]), answer(LS), nl.

answer(LS):- look_up(LS,true), write('Yes.'), nl.
answer(_):- write('No.'), nl.

/* Syntax plus semantics */
sentence(S) --> noun_phrase(NP,S2,O),
                  verb([subject-X|L],O1),
                  complements(L,O1,O2).

complements([],O,O) --> [].
complements([K-N|L],O1,O3) --> complements(L,O1,O2),
                                 case(K),
                                 noun_phrase(N,O2,O3).

noun_phrase(N,O2,O4) --> article(N,O1,O2,O3),
                           common_noun([subject-N|L],O1),
                           complements(L,O3,O4).

noun_phrase(PN,O,O) --> [PN], { proper_noun(PN) }.

case(for) --> [for].
case(direct) --> [].

```

```

/* Morphology */
verb([subject-A,for-P],is_published_by(A,P)) --> [writes].
common_noun([subject-P],publisher(P)) --> [publisher].
article(A,O1,O2, and(O1,O2)) --> [a].
proper_noun(hodges).
proper_noun(penguin).

/* Database verification */
look_up(LS,true):- LS,!.
look_up(LS,false).

/* Intensional database */
is_published_by(A,P):- book(A,_,_,P).

/* Extensional database */
book(hodges,1,logic,penguin,1977).
book(bartlett,2,remembering,cambridge,1972).
book(culicover,3,syntax,academic,1977).

```

Execution:

```

?- go.
[hodges,writes,for,penguin]
Yes.

```

Comments:

The notation of DCG's is adopted. The grammar is defined by two modules: the syntax plus semantics and the morphology.

Non-terminals are allowed to be compound terms in addition to the single atoms allowed in the context-free case. In the right-hand side of a rule, in addition to non-terminals and lists of terminals, there may also be sequences of procedure calls, written within the brackets '{' and '}'. These are used to express extra conditions which must be satisfied for the rule to be valid.

A non-terminal symbol is translated into an $N+2$ place predicate (having the same name), whose first N arguments are those explicit in the non-terminal and whose last two arguments are as in the translation of a context-free non-terminal; procedure calls in the right-hand side of a rule are simply translated as themselves. For example, the rule

```
noun_phrase(PN,O,O) --> [PN], { proper_noun(PN) }.
```

represents the clause

```
noun_phrase(PN,O,O,S0,S):- connects(S0,PN,S),
proper_noun(PN).
```

The first grammar rule of G allows only for sentences comprising a noun phrase followed by a verb with possibly some complements. The first grammar rule for complements admits their absence (the terminal [] stands for the empty list), and the second rule defines the sequence of complements as a string composed by complement, a case and a noun phrase.

Let us explain the role of the logical variable in DCG's, a feature of logic programs. Different arguments of different non-terminals are linked by the same variable. This allows the building up of structures in the course of the unification process.

Consider for example the noun phrase "a publisher" which may be parsed and translated by the grammar rule,

```
noun_phrase(N,Oa,Ob) --> article(N,Oc,Od,Oe),
                           common_noun(N,Of),
                           {constraints(Oa,Ob,Oc,Od,Oe,Of)}.
```

The non-terminal for a noun phrase has three arguments. The interpretation of the last argument Ob will depend on a property Oa of an individual N, because in general a noun phrase contains an article such as 'a'. The word 'a' has the interpretation Oe,

and(Oc, Od)

in the context of two properties Oc and Od of an individual N. The property Oc will correspond to the rest of the noun phrase containing the word 'a', and the property Od will come from the rest of the sentence. Therefore, Oe will contain an overall interpretation, and it is linked to Ob by the same variable. As Of is the property of the common noun, it is linked to Oc by the same variable. Oa has the description of the properties of N, and it will depend on the properties coming from the rest of the sentence. Therefore, Oa is linked to Od by the same variable.

Note that each grammar rule takes an input string, analyzes some initial part, and produces the remaining part as output for further analysis.

Problem 136

Verbal statement:

Write a simple grammar able to parse sentences, such as

"John ate the cake"

and to construct their corresponding deep tree structure.

(Suggestion: use the definite clause grammar (DCG) formalism).

Logic program:

```
sentence(s(T1,T2)) --> noun_phrase(T1),
                           verb_phrase(T2).
```

```
noun_phrase(np(T1,T2)) --> determiner(T1),
                           noun(T2).
```

```

verb_phrase(vp(T1,T2)) --> verb(T1), noun_phrase(T2).

determiner(det(the)) --> [the].
determiner(det([])) --> [].

noun(n(john)) --> [john].
noun(n(cake)) --> [cake].

verb(v(ate)) --> [ate].

```

Problem 137 [F. Pereira; Warren 1978]

Verbal statement:

Write a context-free grammar that covers the following sentences,

"John loves Mary",
 "every man that lives loves a woman"

and that formalizes the mapping between English and formulae of classical logic.

Suggestion: use the definite clause grammar formalism

Logic program:

```

:-op(900, xfx, =>).
:-op(800, xfy, &).
:-op(300, xfx, :).

sentence(P) --> noun_phrase(X,P1,P), verb_phrase(X,P1).

noun_phrase(X,P1,P) -->
  determiner(X,P2,P1,P), noun(X,P3), rel_clause(X,P3,P2).
noun_phrase(X,P,P) --> name(X).

verb_phrase(X,P) --> trans_verb(X,Y,P1), noun_phrase(Y,P1,P).
verb_phrase(X,P) --> intrans_verb(X,P).

rel_clause(X,P1,P1&P2) --> [that], verb_phrase(X,P2).
rel_clause(_,P,P) --> [].

determiner(X,P1,P2, all(X):(P1=>P2) )--> [every].
determiner(X,P1,P2, exists(X):(P1&P2) ) --> [a].

noun(X,man(X)) --> [man].
noun(X,woman(X)) --> [woman].

name(john) --> [john].

trans_verb(X,Y,loves(X,Y)) --> [loves].
intrans_verb(X,lives(X)) --> [lives].

```

Problem 138 [L.Pereira et al. 1978 c]

Verbal statement:

Write a grammar that covers the following sentences,

"fred shot john"

"mary was liked by John"

"fred told mary to shoot John"

"John was believed to have been shot by fred"

"was John believed to have told mary to tell fred"

Suggestion: use the definite clause grammar formalism.

Logic program:

```

sentence(S) -->
    [W], {aux_verb(W,Verb,Tense)}.
    noun_phrase(G_Subj),
    rest_sentence(q,G_Subj,Verb,Tense,S).

sentence(S) -->
    noun_phrase(G_Subj),
    [W], {verb(W,Verb,Tense)},
    rest_sentence(del,G_Subj,Verb,Tense,S).

rest_sentence(Type,G_Subj,Verb,Tense,
             s(Type,L_Subj,tns(Tense1),VP) ) -->
    rest_verb(Verb,Tense,Verb1,Tense1),
    {verbtype(Verb1,VType)},
    complement(VType,Verb1,G_Subj,L_Subj,VP).

rest_verb(have,Tense,Verb,(Tense,perfect)) -->
    [W], {past_participle(W,Verb)}.

rest_verb(Verb,Tense,Verb,Tense) --> [].

complement(copula,be,Obj,Subj, vp(v(Verb),Obj1) ) -->
    [W], {past_participle(W,Verb)}, transitive(Verb),
    rest_object(Obj,Verb,Obj1),
    agent(Subj).

complement(transitive,Verb,Subj,Subj, vp(v(Verb),Obj1) ) -->
    noun_phrase(Obj),
    rest_object(Obj,Verb,Obj1).

complement(intransitive,Verb,Subj,Subj, vp(v(Verb)) ) --> [].

rest_object(Obj,Verb,S) -->
    {s_transitive(Verb)},
    [to,Verb1], {infinitive(Verb1)},
    rest_sentence(del,Obj,Verb1,present,S).

rest_object(Obj,_,Obj) --> [].

```

```

agent(Subj) --> [by], noun_phrase(Subj).
agent(np(pro(someone))) --> [].

noun_phrase(np(Det,adj(Adjs),n(Noun))) -->
    [Det], {determiner(Det)},
    adjectives(Adjs),
    [Noun], {noun(Noun)}.
noun_phrase(np(npr(PN))) --> [PN], {proper_noun(PN)}.

adjectives([Adj|Adjs]) -->
    [Adj], {adjective(Adj)},
    adjectives(Adjs).
adjectives([]) --> [].

aux_verb(W,V,T):- verb(W,V,T), auxiliary(V).

auxiliary(be).
auxiliary(have).

verb(is,be,present).
verb(was,be,perfect).
verb(tell,tell,present).
verb(told,tell,perfect).
verb(shoot,shoot,present).
verb(shot,shoot,perfect).
verb(like,like,present).
verb(liked,like,perfect).
verb(believe,believe,present).
verb(believed,believe,perfect).

proper_noun(john).
proper_noun(fred).
proper_noun(mary).

determiner(the).

adjective(nice).

noun(book).

verbtype(be,copula).
verbtype(V,transitive) :- transitive(V).
verbtype(V,intransitive) :- intransitive(V).

transitive(V) :- s_transitive(V).
transitive(shoot).

s_transitive(tell).
s_transitive(believe).
s_transitive(like).

infinitive(be).
infinitive(shoot).
infinitive(tell).
infinitive(have).

```

```

past_participle(been,be).
past_participle(shot,shoot).
past_participle(told,tell).
past_participle(liked,like).
past_participle(believed,believe).

```

Problem 139

Verbal statement:

Write a simple program able to derive Portuguese relative constructions with “que” (that/who), given their sentential components.

Hint:

Consider the following two sentences,

"eu vi o rapaz" (I saw the boy)
 "o rapaz comprou o livro" (The boy bought the book)

Build up their deep structure and analyze how they may be transformed to represent the corresponding relative sentence.

(This problem was suggested by L. Pereira).

Logic program:

```

go:- sentence(F), write(F), nl
      phrase(Tree, F, []), nl, write(Tree)
      transrel(Tree, RelTree), nl, write(RelTree), nl,
      phrase(RelFr, RelTree, []), nl, write(RelFr), nl, fail.
go.
/* Grammar */
phrase([f,X,Y,Z]) --> np(X), aux(Y), vp(Z).
phrase([f(rel), que, X]) --> [que], phrase(X).

np([sn,X]) --> n(X).
np([np,X,Y]) --> det(X), n(Y).
np([np,X]) --> pro(X).
np([np,X,Y]) --> np(X), phrase(Y).
np([]) --> [].

vp([vp,X]) --> v(X).
vp([vp,X,Y]) --> cop(X), adj(Y).
vp([vp,X,Y]) --> v(X), np(Y).
vp([vp,X]) --> cop(X), np(Y).

aux([aux,X]) --> tpo(X).
/* Lexicon */

```

```

tpo([tpo,past]) --> [past].
cop([cop,estar]) --> [estar].
cop([cop,ser]) --> [ser].
adj([adj,contente]) --> [contente].
adj([adj,bonito]) --> [bonito].
adj([adj,vermelho]) --> [vermelho].
pro([pro,que]) --> [que].
n([n,eu]) --> [eu].
n([n,rapaz]) --> [rapaz].
n([n,livro]) --> [livro].
det([det,o]) --> [o].
v([v,ver]) --> [ver]
v([v,comprar]) --> [comprar].
/* Transformations for the grammar */
/* Applies the relativization transformation to T to get V */
transrel(T,V):- change([np,[np|ST1],R],
                      [np,[np|ST1],[f|ST2]],
                      T,
                      V),
               change([np,[pro,que]],
                      [np|ST1]
                      [f|ST2],
                      R).

/* Useful predicates */
/* If ST is a subtree of T, then substitute the initial tree
   CH in T, obtaining the final tree R */
change(CH,T,T,CH):- !.
change(CH,ST,[OP,T|TS],[OP,R|TS]):- change(CH,ST,T,R).
change(CH,ST,[OP,T|TS],[OP,T|RS]):- change1(CH,ST,TS,RS).

change1(CH,ST,[T|TS],[R|TS]):- change(CH,ST,T,R).
change1(CH,ST,[T|TS],[T|RS]):- change1(CH,ST,TS,RS).

/* Sentences */
sentence([eu,past,ver,o,rapaz,o,rapaz,past,comprar,o,livro]).
```

Execution:

```

?- go.
[eu,past,ver,o,rapaz,o,rapaz,past,comprar,o,livro]
[f,[np,[n,eu]], [aux,[tpo,past]], [vp,[v,ver],[np,[np,[det,o],
[n,rapaz]], [f,[np,[det,o],[n,rapaz]]], [aux,[tpo,past]], [vp,
```

```
[v,comprar],[np,[det,o],[n,livro]]]]]]]
[f,[np,[n,eu]],[aux,[tpo,past]],[vp,[vp,ver],[np,[np,[det,o],
[n,rapaz]], [f,[np,[pro,que]]],[aux,[tpo,past]],[vp,[v,comprar],
[np,[det,o],[n,livro]]]]]]]
```

Logic program 2:

In the logic program 1 we substitute the transformation 'transrel' by:

```
transrel(T,V):- change([np,[np|ST1],[f(rel),que,[f|ST2]]], 
    [np,[np|ST1],[f|ST2]], 
    T, 
    Q),
    subtreeinphrase([np|ST1],ST2),!, 
    transrel(Q,S),
    change([f(rel),que,W], 
        [f(rel),que,ST3], 
        S,
        V),
    change([], [np|ST1],ST3,W),!.
transrel(T,T).

subtreeinphrase1(T,T).
subtreeinphrase1(ST,[OP,T|TS]):- OP\==f, subtreeinphrase1(ST,T).
subtreeinphrase1(ST,[OP,T|TS]):- OP\==f, subtreeinphrase(ST,TS).

subtreeinphrase(ST,[T|TS]):- subtreeinphrase1(ST,T).
subtreeinphrase(ST,[T|TS]):- subtreeinphrase(ST,TS).

sentence([o,livro,o.rapaz,past,comprar,o,livro,o,livro,past,ser,
vermelho,past,ser,bonito]).
```

Execution:

```
?- go.
[o,livro,o.rapaz,past,comprar,o,livro,o,livro,past,ser,vermelho,
past,ser,bonito]

[f,[np,[np,[det,o],[n,livro]], [f,[np,[det,o],[n,rapaz]],
[aux,[tpo,past,]], [vp,[v,comprar],[np,[np,[det,o],[n,livro]],
[f,[np,[det,o],[n,livro]], [aux,[tpo,past]], [vp,[cop,ser],
[adj,vermelho]]]]]], [aux,[tpo,past]], [vp,[cop,ser],[adj,bonito]]]
[f,[np,[np,[det,o],[n,livro]], [f(rel),que,[f,[np,[det,o],
[n,rapaz]], [aux,[tpo,past]], [vp,[v,comprar],[np,[[],[f(rel),
que,[f,[],[aux,[tpo,past]], [vp,[cop,ser],[adj,vermelho]]]]]]], [aux,[tpo,past]], [vp,[cop,ser],[adj,bonito]]]]]
```

Logic program 3:

```

hi:- sentence(F), write(F), nl,
      phrase(Tree, F, []), nl,
      write(Tree), nl, nl,
      print_tree(Tree), nl,
      transrel(Tree, RelTree), nl,
      write(RelTree), nl, nl,
      print_tree(RelTree), nl, nl,
      phrase(Relfr, Reltree, []), nl,
      write(Relfr), nl, fail.

hi.

/* Grammar */

phrase(f(X,Y,Z)) --> np(X), aux(Y), vp(Z).
phrase(f(rel,que,X)) --> [que], phrase(X).

np(np(X)) --> n(X).
np(np(X,Y)) --> det(X), n(Y).
np(np(X)) --> pro(X).
np(np(X,Y)) --> np(X), phrase(Y).
np(void) --> [].

vp(vp(X)) --> v(X).
vp(vp(X,Y)) --> cop(X), adj(Y).
vp(vp(X,Y)) --> v(X), np(Y).
vp(vp(X,Y)) --> cop(X), np(Y).

aux(aux(X)) --> tpo(X).

/* Lexicon */

tpo(tpo(past)) --> [past].
cop(cop(estar)) --> [estar].
cop(cop(er)) --> [er].
adj(adj(bonito)) --> [bonito].
adj(adj(vermelho)) --> [vermelho].
pro(pro(que)) --> [que].

n(n(eu)) --> [eu].
n(n(bolo)) --> [bolo].
n(n(rapaz)) --> [rapaz].
n(n(livro)) --> [livro].
det(det(o)) --> [o].
v(v(ver)) --> [ver].
v(v(comprar)) --> [comprar].

/* Transformations for the grammar */

/* Applies the relativization transformation to T to get V */

```

```

transrel(T,V):- change(t(t(void,np,ST1),np,
                         t(t(void,rel,
                               t(void,que,t(void,f,ST2))
                               ),f,void)),
                         t(t(void,np,ST1),np,t(void,f,ST2)),T,Q),
                         subtreeinphrase(t(void,np,ST1),ST2),!,
                         transrel(Q,S),
                         change(t(t(void,rel,t(void,que,W)),f,void),
                               t(t(void,rel,t(void,que,ST3)),f,void),
                               S,
                               V),
                         change(void,t(void,np,ST1),ST3,W),!.
transrel(T,T).

/* Useful predicates */

/* If -ST is a subtree of -T-, then substitute the initial
   tree -CH- in -T-, obtaining the final tree -R- */

change(CH,T,T,CH).
change(CH,ST,t(T,X,TS),t(R,X,TS)):- change(CH,ST,T,R).
change(CH,ST,t(T,X,TS),t(T,X,RS)):- change1(CH,ST,TS,RS).

change1(CH,ST,[T|TS],[R|TS]):- change(CH,ST,T,R).
change1(CH,ST,[T|TS],[T|RS]):- change1(CH,ST,TS,RS).

subtreeinphrase1(T,T).
subtreeinphrase1(ST,t(T,X,TS)):- X\==f,subtreeinphrase1(ST,T).
subtreeinphrase1(ST,t(T,X,TS)):- X\==f,subtreeinphrase(ST,TS).

subtreeinphrase(ST,[T|TS]):- subtreeinphrase1(ST,T).
subtreeinphrase(ST,[T|TS]):- subtreeinphrase(ST,TS).

/* Print term as a Tree */

print_tree(T):- numbervars(T,1,_),
               pt(T,0),nl,fail.
print_tree(_).

pt(A,I):- as_is(A),!,
          tab(I),write(A),nl.
pt([T|TS],I):- !,pt(T,I),pt(TS,I).
pt(T,I):- !,T=..[F|As],
          tab(I),write(F),nl,
          I0 is I+3,p1(As,I0).

p1([],_):- !.
p1([A|As],I):- !,
               pt(A,I),
               p1(As,I).

as_is(A):- atomic(A),!.
as_is('$VAR'(_)):- !.
as_is(X):- quote(X).

```

```
/* Sentences */
sentence([o,livro,o,rapaz,past,comprar,o,livro,o,livro,
past,ser,vermelho,past,bonito]).
setence([o,bolo,eu,past,ver,o,rapaz,o,bonito,past,ver,o,livro]).
```

Problem 140

Verbal statement:

Write a program able to interpret document titles and to be included in a question-answering system, due to implement an automatic library service.

The program goal consists of proposing categories to the user by reading the document titles, and interpreting the role played by prepositions.

Consider the following title :

"Logic for problem solving".

Logic program:

```
/* Title interpretation      */
:-op(400,xfy,'->').
/* Using the title definition of a text for classification
purposes */
/* Grasping meaning for the title entities of a document */
parser(T,C):-
    title(Out,T,[]),semant(Out,C1,C2),
    output(C1,C2),append(C1,C2,C).

output([],[]):-!.
output([],C2):-display('I recommend as categories:  '), 
    print_ph(C2),nl,!.
output(C1,C2):-display('I recommend as categories:  '),
    print_ph(C1),display('and '),print_ph(C2),nl.

/* Basic grammar rules governing the title construction */
title([np(X),Y,Z]) --> bp(X,Y),!,title(Z).
title([np(X)]) --> np(X).

bp([],P) --> prep(P).
bp([Y|L],P) --> det(Y),!,bp(L,P).
bp([n(X)|Y],P) --> [X],bp(Y,P).

np([Y|L]) --> det(Y),!,np1(L).
np(X) --> np1(X).
```

```

np1([n(X)|Y]) --> [X],np1(Y).
np1([n(X)]) --> [X].
/* Dictionary: lexical units */
prep(prep(for)) --> [for].
det([a]) --> [a].
det([an]) --> [an].
det([the]) --> [the].
/* Prepositional cases: ordering plans for conversation */
prep_case(for,X,Y):-
    assert(case(tool->X,application->Y)),nl,nl,
    write('your document deals with ''),
    print_ph(X),write(''),
    write(' as a tool,'),nl,
    write('with the objective to contribute to '),
    print_ph(Y),write('.'),nl,nl,hypothesis1(X,Y).

hypothesis1(X,Y):-
    write('My first hypothesis for classifying it '),
    write('is to take its nouns as hints,'),
    nl,write('So " '),print_ph(X),write(''),
    write(' will drive us, in order '),
    write('to attain the main category, and'),
    nl,write(''),print_ph(Y),
    write(''),write(' towards other categories.'),nl,nl.

/* Gathering of clues,given by user,for classifying a
document */

semant([np(X)],X,Y).
semant([np(X),prep(Y),[np(Z)]],W,T):-
    list_nouns(X,Z,U,V),
    pre_case(Y,U,V),
    drive_in(W,T).

/* Driving the conversation about title content */
drive_in(W,V):-case(X,Y),
    drive(X,U),drive(Y,V),test(X,U,W).
drive(X->Y,C):-move(X,Y),plan(X,L),state(X,L,C),!.
drive(X->Y,[]).

move(X,Y):-write('Do you want to follow the idea that '),
    print_ph(Y),write('is an AI '),
    write(X),write(?),nl,agreement(yes),
    write('OK.Let us talk about this hypothesis'),
    write(' and fix the kind of '),
    write(X),write(!),nl,!.

```

```

agreement(Input):-read(Input).

state(_,[],[]):- !.
state(A,X,Y):-
    write('During our dialogue I grasp the following:'),nl,
    print_ph(X),write('as core topics of your text.'),nl,
    under(A,X,Y),state1(Y),nl,!.

state1([]):- !.
state1(Y):-write('So I conclude it may be under'),
           write(' the categories: '),
           print_ph(Y).

under(A,[X|L],[N|NL]):-choose(A,X,N),under(_,L,NL).
under(_,[],[]).

choose(tool,X,Y):-g_tool(X,Y).
choose(application,X,Y):-g_application(X,Y).
choose(____,_).

/* Plan 1: through the tree of tools */

plan(tool,L):-
    try(t([techniques,programming_languages],
        [t([reasoning,representational,searching],
            [t([puzzle_solving,question_answering,common_sense],
                [[],[],t([deduction,planning],[[],[]])]),
            t([predicate_calculus,semantic_nets],[[],[]])],
            t([state_space,problem_reduction],[[],[]])]),
        t([numerical_languages,ai_languages],[[],t([list_type,predicate_logic_type],[[],[]])])),L,tool),!.

/* Plan 2: through the tree of applications */

plan(application,L):-
try(t([cognition,other_fields],[t([activities,abilities],
    [t([intellectual_skills,motor_skills],
        [t([playing,proving],[[],t([theorems,programs],[[],[]])],[],[])
    ]),t([language,vision],[[],[]])]),[],L,application).

try(t([X|_],[T|_],[X|L],C):-
    request(X,C),!,try(T,L,C).

try(t([_|XN],[_|TN]),L,C):-try(t(XN,TN),L,C).

try(____,_).

request(X,C):-write('Is '),write(X),
             write(' an adequate '),write(C),
             write(' worked out in your text?'),nl,
             agreement(yes).

/* testing return to a previous conversation */

```

```
test(X->Y,[],Z):-  
    write('As no conclusion was taken about core topics of '),  
    write(X),nl,  
    write('I suggest we restart our conversation!'),  
    nl,write('Do you agree?'),nl,  
    agreement(yes).  
test(_,Y,Y).
```

Execution:

```
?- parser.  
Document title?  
[logic,for,problem,solving].  
  
Your document deals with "logic" as a tool,  
with the objective to contribute to "problem solving".  
  
My first hypothesis for classifying it is to take its nouns as  
hints.  
So "logic" will drive us, in order to attain the main  
category, and "problem solving" towards other categories.  
  
Do you want to follow the idea that logic is an AI tool?  
yes.  
Ok.Let us talk about this hypothesis and fix the kind of tool!  
Is techniques an adequate tool worked out in your text?  
yes.  
Is reasoning an adequate tool worked out in your text?  
yes.  
Is puzzle_solving an adequate tool worked out in your text?  
no.  
Is question_answering an adequate tool worked out in your text?  
no.  
Is common_sense an adequate tool worked out in your text?  
no.  
During our dialogue I grasp the following:  
techniques reasoning as core topics of your text.  
So I conclude it may be under the categories: [] 364  
Do you want to follow the idea that problem solving is an AI  
application?  
Ok.Let us talk about this hypothesis and fix the kind of  
application!  
Is cognition an adequate application worked out in your text?  
yes.  
Is activities an adequate application worked out in your text?  
no.  
Is abilities an adequate application worked out in your text?  
yes.
```

Is language an adequate application worked out in your text?
yes.

During our dialogue I grasp the following:
cognition abilities language as core topics of your text.
So I conclude it may be under the categories: [] 336 365
I recommend as categories: [] 364 and [] 336 365

Problem 141 [Abreu 1980, personal communication]

Verbal statement:

Write a lexical analyzer generator that transforms regular expressions into non-deterministic finite automata, and afterwards into deterministic finite automata, using the subset construction.

Logic program:

```
/* utilities */
:- op(300,xfy,in).

X in [X|_].
X in [_|L] :- X in L.

index(0,[]):- !.
index(N,L) :- N > 0, !, index1(N,1,L).

index1(N,N,[N]) :- !.
index1(N,O,[O|L]) :- P is O+1, index1(N,P,L).

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

union([],X,X).
union([X|R],Y,Z) :- X in Y, !, union(R,Y,Z).
union([X|R],Y,[X|Z]) :- union(R,Y,Z).

remove(X,[X|L],L) :- !.
remove(X,[Y|L1],[Y|L2]) :- remove(X,L1,L2).

/* operators */
:- op(200,xfy,to).
:- op(250,xfy,on).
:- op(250,xf,*).
:- op(250,xf,+).
:- op(250,xf,?).
:- op(300,xfy,&).
:- op(350,xfy,!).

/* tree from a regular expression */
```

```

tree(X!Y,node(!,Tx,Ty)):- tree(X,Tx),tree(Y,Ty).
tree(X&Y,node(&,Tx,Ty)):- tree(X,Tx),tree(Y,Ty).
tree(X*,node(*,Tx)):- tree(X,Tx).
tree(X+,node(+,Tx)):- tree(X,Tx).
tree(X?,node(?,Tx)):- tree(X,Tx).
tree(X,node(X)).

/* NFA for an 'optional' node (?) */

node_to_nfa(node(?,_X),[First,Last],
           [First to S1 on epsilon|Transitions]):-
    next_state(First),
    node_to_nfa(_X,[S1,S2],Others),
    next_state(Last),
    conc(Others,[First to Last on epsilon,
                 S2 to Last on epsilon],
         Transitions).

/* NFA for a leaf node */

node_to_nfa(node(_X),[First,Last],[First to Last on _X]):-
    next_state(First),next_state(Last).

/* NFA for an 'or' node (!) */

node_to_nfa(node(!_X,Y),[First,Last],[First to S1 on epsilon,
                                         First to S3 on epsilon|
                                         Others]):-
    next_state(First),
    node_to_nfa(_X,[S1,S2],Rx),
    node_to_nfa(Y,[S3,S4],Ry),
    conc(Rx,Ry,Temp),next_state(Last),
    conc(Temp,[S2 to Last on epsilon,
               S4 to Last on epsilon],Others).

/* NFA for an 'and' node (&) */

node_to_nfa(node(&,_X,_Y),[First,Last],Others):-
    node_to_nfa(_X,[First,Middle],Rx),
    node_to_nfa(_Y,[Middle,Last],Ry),
    conc(Rx,Ry,Others).

/* NFA for a 'repetition' node (*) */

node_to_nfa(node(*,_X),[First,Last],[First to S1 on epsilon,
                                         First to Last on epsilon|
                                         Others]):-
    next_state(First),
    node_to_nfa(_X,[S1,S2],Temp),
    next_state(Last),
    conc(Temp,[S2 to S1 on epsilon,
               S2 to Last on epsilon],Others).

```

```

/* NFA for a 1 or more repetitions node (+) */
node_to_nfa(node(+,X),[First,Last],[First to S1 on epsilon|
                                         Others]):-
    next_state(First),
    node_to_nfa(X,[S1,S2],Temp),
    next_state(Last),
    conc(Temp,[S2 to S1 on epsilon,
              S2 to Last on epsilon],Others).

next_state(X):- current_state(X), Y is X+1,
              retract(current_state(X)),
              assert(current_state(Y)).

next_state(_).

ep_closure([],[]).

/* compute the epsilon-closure of a set of states */
ep_closure([X],Sorted):-
    nfa(_,S),i_closure(X to _ on epsilon,S,L),
    sort([X|L],Sorted).

ep_closure([X|R],E):-
    ep_closure([X],E1),ep_closure(R,E2),
    union(E1,E2,E3),sort(E3,E).

/* compute the transition on some item of a set of states */
transition([X],C,E):-
    nfa(_,S),
    i_closure(X to _ on C,S,E1),sort(E1,E).

transition([X|R],C,E):-
    nfa(_,S),i_closure(X to _ on C,S,E1),
    transition(R,C,E2),
    union(E1,E2,E3),sort(E3,E).

/* used by ep_closure and transition */
i_closure(X to Y on C,Set,[Y|L]):-
    remove(X to Y on C,Set,NSet),
    i_closure(X to _ on C,NSet,L1),
    (C\==epsilon,L2=[];
     i_closure(Y to _ on C,NSet,L2)),
    conc(L2,L1,L).

i_closure(_,_,[]).

/* conversion from an NFA to a DFA - 'explore' (a set of DFA
   states) */

explore(_,[],[],_).

explore(State,L,[C|Rc],Full_c):-
    not_there(State,C,DS),
    explore(State,L1,Rc,Full_c),
    %remaining transitions
    transition(State,
              C,TState,ep_closure(TState,ETState),

```

```

(ETState\==[],explore(ETState,L2,Full_c),
%explore transition
check_dfa_state(ETState,EDS),
L3=[DS to EDS on C];
This=[],L2=[],L3=[]),
conc(L3,L1,L4),conc(L4,L2,L).

explore(_,[],_,_).

not_there(S,C,N):- (dfa_state(N,S,Set_of_C),
!,\+ C in Set_of_C,
retract(dfa_state(N,S,_)),
assert(dfa_state(N,S,[C|Set_of_C]))),
next_state(N),assert(dfa_state(N,S,[C])).

check_dfa_state(Nfa_state,Dfa_state):-
(dfa_state(Dfa_state,Nfa_states,_);
next_state(Dfa_state),
assert(dfa_state(Dfa_state,Nfa_states,[])), 
nfa([_,F],_), 
((F in Nfa_states,
retract(accept(L)),union([Dfa_state],L,L1),
assert(accept(L1)));true).

/* interface with user */

ask:- repeat,write('*** give regular expression ***'),nl,read(X),
(X=end_of_file;check(X),fail).

check(E):- abolish(current_state,1),assert(current_state(0)),
nl,write('>>> regular expression : '),
writeq(E), write(' <<<'), nl,nl,
tree(E,T),
write('>>> tree for regular expression <<<'),nl,nl,
write(' '),write(T),nl,nl,
node_to_nfa(T,[Ini,Fin],Nfa),
write('>>>non-deterministic finite automaton<<<'),
nl,nl,
write('    initial state    : '),write(Ini),nl,
write('    accepting state : '),write(Fin),nl,nl,
write('>>> transitions <<<'),nl,nl,
show(Nfa),
abolish(nfa,2),assert(nfa([Ini,Fin],Nfa)),
ep_closure([Ini],L),
abolish(dfa_state,3),assert(dfa_state(0,L,[])),
abolish(current_state,1),assert(current_state(1)),
(list_of_itms(List),
abolish(accept,1),assert(accept([])),
explore(L,Dfa,List,List),
abolish(dfa,1),assert(dfa(Dfa))),
```

```

write('>>> deterministic finite automaton <<<'),
nl,nl,
write('    initial state   : 0'),nl,
write('    accepting state : '),accept(Z),
write(Z),nl,
nl,write('>>> transitions <<<'),nl,nl,
show(Dfa);
(write('>>> this is already a deterministic finite
automaton<<<'),nl,nl,
assert(accept(Fin)),abolish(dfa,1),
assert(dfa(Nfa))),!.
```

```

list_of_items(L):- nfa(_,S),list_of_items(S,L1),
               remove(epsilon,L1,L2),sort(L2,L).
list_of_items([],[]).
list_of_items(S,N1):- remove(A to B on C,S,S1),
                   list_of_items(s1,L),
                   ((\+ C in L,N1=[C|L]);N1=L).
```

```

show([X]):- show1(X),write(' .'),nl,nl.
show([X|L]):- show1(X),write(' ,'),nl,
             show(L).
```

```

show1(X to Y on epsilon):- write('    from state '),write(X),
                         write(' to state '),write(Y),
                         tab(30),write('-- on <epsilon>').
show1(X to Y on C):-      write('    from state '),write(X),
                     write(' to state '),write(Y),
                     write(' on input itm '''),write(C),
                     write("')).
```

Execution:

```

?- ask.
give regular expression
(a&b)*&b&b&a.
```

Chapter 16 Constructing Personal Databases with Prolog

Answering questions is one of the tasks that may show the ability to make logical deductions. In a question-answering system, facts are represented by unit clauses, say logical formulas. Then, to answer a question from the facts, it must be proved that a formula corresponding to the answer is derivable from the formulas representing the facts.

Problem 142

Build up a list of the objects having some property.

Logic program:

```
build_up(Object, Property, _):- Property,  
                     assert(agenda(Object)), !, fail.  
build_up(_, _, List):- query(List).  
query([Object|List]):- retract(agenda(Object)), !, query(List).
```

Comment:

This program is useful for database applications. The program builds an internal database of facts, called ‘agenda’, by inspecting when some property is true for the object declared by the user. After the inspection is over, the program re-collects all the stored information and at the same time it builds a list structure with this information. Observe that the program uses extra facilities provided by Prolog implementations as ‘built-in’ predicates, ‘assert’ for adding clauses and ‘retract’ for deleting clauses.

Problem 143 [Warren et al. 1977]

Verbal statement:

Calculate the population density per square mile and find countries of similar population density (differing by less than 5%).

Logic program:

```

density(C,D):- population(C,P), area(C,A), D is (P*1000)/A.

answer(C1,D1,C2,D2):- density(C1,D1), density(C2,D2),
    D1>D2, 20*D2 < 19*D1.

population(china,825).           area(china,3380).
population(india,586).          area(india,1139).
population(ussr,252).           area(ussr,8709).
population(usa,212).            area(usa,3609).

```

Problem 144 [Emden 1977]

Verbal statement:

Design a deductive information retrieval system concerning some aspects of the operation of a university department (Kowalski's departmental database), such as those presented under set notation:

Takes

M.	:Adiri	Math!	129
C.Y.K	:Chenk	Math!	225
T.L.	:Cook	Math!	129
T.L.	:Cook	Math!	225

Teaches

J.F.	:Glotz	Math!	129
J.F.	:Glotz	Math!	225
C.	:Twill	Math!	170

Student

M.	:Adiri	Biology		1974
N.A.	:Buczek	Recreation		1976
C.Y.K	:Chenk	Physics		1976
T.L.	:Cook	Engineering		1975
G.C.	:Giusti	Engineering		1976
A	:Hammer	Child Care		1973
K.L.	:Mensink	Kinesiology		1973

Scheduled

Math!	129	Phy@3009	2.30
Math!	301	Phy@3009	2.30

Logic program:

```

:-op(500,xfy,:).
:-op(300,xfy,!).
:-op(300,xfy,@).
:-op(100,xfy,..).

```

```

takes(X,Math!129):- year(X,1),program(X,engineering).

year(X,Z):- student(X,Y,Z1),Z is 1977-Z1.

program(X,Y):- student(X,Y,Z).

course_prefix(X!Y,X).

course_number(X!Y,Y).

initials(X:Y,X).

lastname(X:Y,Y).

graduate_course(X):- course_number(X,Y),Y>\499.

conflict(X1,X2):- scheduled(X1,Y,Z),
                  scheduled(X2,Y,Z),
                  X1=\=X2.

teaches(X,Y):- takes(Z,Y).

takes(m.:adiri,math!129).
takes(c.y.k:chenk,math!225).
takes(t.l:cook,math!129).
takes(t.l:cook,math!225).

teaches(j.f:glotz,math!129).
teaches(j.f:glotz,math!225).
teaches(c:twill,math!170).

student(m:adiri,biology,1974).
student(n.a:buczek,recreation,1976).
student(c.j.k:chenk,physics,1976).
student(t.l:cook,engineering,1975).
student(g.c:giusti,engineering,1976).
student(a:hammer,child_care,1973).
student(k.l:mensink,kinesiology,1973).

scheduled(math!129,phy@3009,2.30).
scheduled(math!301,phy@3009,2.30).

```

Execution:

Example of simple queries:

```

?- teaches(X,math!225).

?- student(t.l:cook,X,_).

?- student(c.y:chenk,_,X).

?- takes(t.l:cook,math!129).

?- scheduled(math!301,_,X).

```

```
?- student(X,engineering,_),fail.  
?- takes(X,math!_),fail.  
?- student(X,child_care,_),fail.  
?- student(X,child_care,_).  
?- teaches(_:glotz,X),scheduled(X,Y,_).
```

Example of complex queries:

```
?- takes(t.l:cook,X),graduate_course(X).  
?- teaches(_:glotz,X),conflict(X,_),fail.  
?- teaches(_:glotz,X),course_prefix(X,Y).  
?- teaches(_:twill,X),graduate_course(X).  
?- teaches(glotz,X),takes(Y,X),fail.  
?- program(X,engineering),last_name(X,Y),fail.
```

Problem 145 [Warren 1975 a]

Verbal statement:

John likes Mary. John likes Jane. Mary likes Pete. Pete likes Kate. Jane likes John. Kate likes every one. Who does like someone and is liked back?

Logic program:

```
:op(900,xfy,'&').  
:op(700,xfy,'likes').  
  
john likes mary.  
john likes jane.  
mary likes pete.  
pete likes kate.  
jane likes john.  
kate likes _.
```

Execution:

```
?-X likes Y, Y likes X, write((X&Y)).
```

Execution:

```
john & jane;  
pete & kate;  
jane & john;
```

kate & pete;
 kate & kate;
 no

Problem 146 [Warren 1975 a]

Verbal statement:

Suppose we are given the following family tree, with each person's age in brackets:

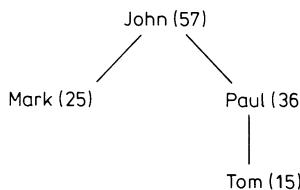


Fig. 1

and we wish to find all solutions to the question “Does John have two descendants whose ages differ by ten years?”

Write a program to solve this problem.

Logic program:

```

has_descendant(X, Z) :- begat(X, Z).
has_descendant(X, Z) :- has_descendant(X, Y),
                     begat(Y, Z).

begat(john, mark).
begat(john, paul).
begat(paul, tom).

is_aged(john, 57).
is_aged(mark, 25).
is_aged(paul, 36).
is_aged(tom, 15).
  
```

Execution:

```

?-has_descendant(john, X), is_aged(X, N),
   has_descendant(john, Y), is_aged(Y, M), M is N+10.
  
```

Problem 147 [L. Pereira et al. 1978]

Verbal statement:

Construct a small database obeying:

"the goal 'descendant(X,Y)' is true if Y is a descendant of X"

Consider that Ishmael and Isaac are descendants of Abraham, and Esau and Jacob are descendants of Isaac.

Logic program:

```
descendant(X,Y):- offspring(X,Y).
descendant(X,Z):- offspring(X,Y),descendant(Y,Z).

offspring(abraham,ishmael).
offspring(abraham,isaac).
offspring(isaac,esau).
offspring(isaac,jacob).
```

Execution:

If for example the question:

```
?-descendant(abraham,X).
```

is executed, Prolog's backtracking results in different descendants of Abraham being returned as successive instances of the variable X, i.e.,

```
X = ishmael
X = isaac
X = esau
X = jacob
```

Problem 148

Verbal statement:

Write a program considering the following facts:

- number(X,N) means that person X can be reached by calling phone number N.
- visits(X,Y) means that person X is visiting person Y.
- at(X,Y) means that person X is at the residence of person Y.
- phone(X,N) mean that person X has phone number N.

and the following premises:

- 1) (VX) (VY) (VZ) [visits(X,Y) & at(Y,Z) ==> at(X,Z)]
- 2) (VU) (VV) (VN) [at(U,V) & phone(V,N) ==> number(U,N)]

able to find the way of reaching a person, having a particular phone number base and knowing who visits whom and where is the visited person.
 (This problem was suggested from “The Thinking Computer” by Bertrand Raphael).

Logic program

```
/* Finding Process */
find(X):- number(X,N),write('phone : '),write(N),nl.
find(_):- write('Don''t Know.'),nl.

number(X,N):- at(X,Y),phone(Y,N),!.
number(X,N):- phone(X,N).

at(X,Z):- visits(X,Y),at(Y,Z).

/* Phone numbers database */

phone(coleman,'100001').
phone(gordon,'100002').
phone(wagner,'100003').
phone(smith,'100004').
```

Execution:

Suppose you want to reach Coleman and you know he is visiting with Wagner and that Wagner is at Gordon's house.

You just have to write:

```
visits(coleman,wagner).
at(wagner,gordon).
?-find(coleman).
```

the answer would be:

```
phone : 100002
```

Problem 149

Verbal statement:

Cooking afloat needs advice. Write a program to help choosing recipes for yachts-men.

Logic program:

```
soup(chicken_in_mushroom_sauce).
soup(blanquette_of_veal).
```

```
soup(fish_in_tomato_sauce).
soup(gammon_casserole).
soup(braised_onions).

meat(canned_pork_pie).
meat(bully_beef_and_tomatoes).
meat(stuffed_marrows_or_potatoes).
meat(beef_pudding).
meat(meat_curry).
meat(savoury_sausages_in_tomato_sauce).
meat(ox_tongue).

fish(boiled_haddock).
fish(grilled_salmon).
fish(fish_pie_with_canned_salmon).
fish(sole_meuniere).
fish(stuffed_fillets_of_plaice).
fish(turbot_cooked_in_wine).

sauce(madeira_sauce).
sauce(wine_sauce).
sauce(mustard_sauce).
sauce(gooseberry_sauce).

dessert(fruit_crumble).
dessert(whole_apples).
dessert(cheese_straws).
dessert(sand_cake).
dessert(easy_fruit_cake).
dessert(spiced_scones).

drinks(dry_special).
drinks(yellow_peril).
drinks(bloodhound_cocktail).
drinks(corpse_reviver).
drinks(yankee_doodle).
drinks(orange_ginger).

dish(D):- fish(D);meat(D).

full_meal(Drink,Soup,Dish,Dessert):- drink(Drink),
                                         soup(Soup),
                                         dish(Dish),
                                         dessert(Dessert).

single_meal(Soup,Dish):- soup(Soup),
                         dish(Dish).
```

Execution:

Let us see the kind of question a yachtsman may put to the program:

```
"is fish_in_tomato_sauce a meal?"  
?- meat(fish_in_tomato_sauce).  
no  
"Is braised_onions a soup?"  
?- soup(braised_onions).  
yes  
"What kind of drinks can I prepare?"  
?- drinks(D).  
D=dry_special;  
D=yellow_peril;  
D=bloodhound_cocktail;  
D=corpse_reviver;  
D=yankee_doodle;  
D=orange_ginger;  
no
```

Problem 150

Verbal statement:

A doctor wants to prepare diets for his patients taking into account the number of units of specific heat per 100 g of the food. Write a small database with the relevant information.

Logic program:

```
value(100,bread,250).  
value(100,fruit,50).  
value(100,chicken,122).  
value(100,fish,68).  
value(100,milk,65).  
value(100,cheese,186).  
value(100,egg,162).  
value(100,pork,330).  
value(100,cow,159).  
value(100,pork_liver,130).  
value(100,sheep_kidney,100).  
value(100,green_beans,35).  
value(100,apples,47).  
value(100,cauliflower,32).  
value(100,sausages,88).
```

```

value(100,peas,53).
value(100,tomato,21).

soup(chicken_in_mushroon_sauce).

meat(bully_beef_and_tomatoes).

value(bully_beef_and_tomatoes,500).
value(chicken_in_mushroon_sauce,150).

single_meal(S,D):- soup(Soup),
                  dish(Dish).

dish(D):- fish(D);meat(D).

balanced_meal(S,D,V):- single_meal(S,D),value(S,V1),value(D,V2),
                      V is V1+V2,    V<700.

```

Problem 151

Verbal statement:

Write a database with information about the Lisbon region, including all the counties and the town councils.

Logic problem:

```

county(lisboa,alenquer).
county(lisboa,arruda_dos_vinhos).
county(lisboa,cadaval).
county(lisboa,cascais).
county(lisboa,lisboa).
county(lisboa,loures).
county(lisboa,lourinha).
county(lisboa,mafra).
county(lisboa,oeiras).
county(lisboa,sintra).
county(lisboa,sobral_de_monte_agraco).
county(lisboa,torres_vedras).
county(lisboa,vila_franca_de_xira).

town_council(lisboa,ajuda).
town_council(lisboa,alcantara).
town_council(lisboa,alto_do_pina).
town_council(lisboa,alvalade).
town_council(lisboa,ameixoeira).
town_council(lisboa,anjos).
town_council(lisboa,beato).
town_council(lisboa,benfica).
town_council(lisboa,campo_grande).

```

```
town_council(lisboa,campolide).
town_council(lisboa,carnide).
town_council(lisboa,castelo).
town_council(lisboa,charneca).
town_council(lisboa,coracao_de_jesus).
town_council(lisboa,encarnacao).
town_council(lisboa,graca).
town_council(lisboa,lapa).
town_council(lisboa,lumiar).
town_council(lisboa,madalena).
town_council(lisboa,martires).
town_council(lisboa,marvila).
town_council(lisboa,merces).
town_council(lisboa,nossa_senhora_de_fatima).
town_council(lisboa,pena).
town_council(lisboa,penha_de_franca).
town_council(lisboa,prazeres).
town_council(lisboa,sacramento).
town_council(lisboa,santa_catarina).
town_council(lisboa,santa_engracia).
town_council(lisboa,santa_isabel).
town_council(lisboa,santa_justa).
town_council(lisboa,santa_maria_de_belem).
town_council(lisboa,santa_maria_dos_olivais).
town_council(lisboa,santiago).
town_council(lisboa,santo_condestavel).
town_council(lisboa,santo_estevao).
town_council(lisboa,santos_o_velho).
town_council(lisboa,sao_cristovao).
town_council(lisboa,sao_domingos_de_benfica).
town_council(lisboa,sao_francisco_xavier).
town_council(lisboa,sao_joao).
town_council(lisboa,sao_joao_de_brito).
town_council(lisboa,sao_joao_de_deus).
town_council(lisboa,sao_jorge_de_arroios).
town_council(lisboa,sao_jose).
town_council(lisboa,sao_mamede).
town_council(lisboa,sao_miguel).
town_council(lisboa,sao_nicolau).
town_council(lisboa,sao_paulo).
town_council(lisboa,sao_sebastiao_da_pedreira).
town_council(lisboa,sao_vicente_de_fora).
town_council(lisboa,se).
town_council(lisboa,socorro).

ajuda(21536,1300).
alcantara(21276,1300).
```

```

alto_do_pina(9886,1100).
alvalade(13957,1700).
ameixoeira(6047,1700).
anjos(17254,1000).
beato(16729,1900).
benfica(37769,1500).

post_office_code(T,C):- town_council(_,T),
                      P=..[T,_,C],P.

voters(T,V):- town_council(_,T),
                  P=..[T,V,_],P.

between(X,A,B):- X>A,X<B.

```

Execution:

Examples of queries:

“Which is the post – office code of Benfica?”

```
?- post_office_code(benfica,C).
C=1500
```

“Which parish has 1300 as post – office code?”

```
?- post_office_code(P,1300).
P=ajuda;
P=alcantara;
no
```

“How many voters has Anjos?”

```
?- voters(anjos,V).
V=17254
```

“Which parishes have a population of voters between 6000 and 10000?”

```
?- voter(T,V),between(N,6000,10000).
T=alto_do_pina
V=9886;
T=ameixoeira
V=6047;
no
```

Problem 152 [L. Pereira; Porto 1980]

Verbal statement:

Write the following query:

“Is there a student such that a professor teaches him two different courses in the same room?”

for a database of students who take courses, professors who teach courses, and courses held on certain weekdays and rooms.

Logic program:

```

query(S,P):- student(S,C1),
            course(C1,D1,R),
            professor(P,C1),
            student(S,C2),
            course(C2,D2,R),
            professor(P,C2),C1 =\= C2.

student(robert,prolog).
student(john,music).
student(john,prolog).
student(john,surf).
student(mary,science).
student(mary,art).
student(mary,physics).

professor(luis,prolog).
professor(luis,surf).
professor(antonio,prolog).
professor(eureka,music).
professor(eureka,art).
professor(eureka,science).
professor(eureka,physics).

course(prolog,monday,room1).
course(prolog,friday,room1).
course(surf,sunday,beach).
course(maths,tuesday,room1).
course(maths,friday,room2).
course(science,thursday,room1).
course(science,friday,room2).
course(art,tuesday,room1).
course(physics,thursday,room3).
course(physics,saturday,room2).

```

Problem 153

Verbal statement:

Write a simple question-answering system for the exploration of a database with transistor information (transistor name, material, polarity, function, power, collector base voltage, collector emitter voltage, amplification factor and capsule type), and according to the table:

characteristics

Transistor Name	mat	pol	fun	pow	vcb	vce	hfe	cap
2n2219	si	n	hsa	800	60	30	120	TO5
2n2904	si	p	hss	300	60	40	120	TO5
2n3055	si	n	lpa	115000	100	70	70	TO3
2n3904	si	n	hss	310	60	40	300	TO92
2n3906	si	p	hss	310	40	40	300	TO92

The system is requested to deal with users' changes of mind, translation of users' input data, and output of all characteristics for a given transistor, specified by the user.

Logic program:

```

begin:- write('Characteristics(value)'),nl,repeat,nl,
        continue(_).

continue(L):- write(' - '),ttyflush,read(C),process(C,L).

process('Goodbye',_).
process(stop,L):- ( trans(X,L),nl,write(trans(X,L)),nl,
                   write('Do you want it(yes or no) ?'),
                   nl,read(S),S=yes,!,
                   write('New characteristics'),nl,fail ;
                   write('No transistors available'),nl,continue(L) ).

process(C,L):- ( subs(C,L),L=L1 ; replace(C,L,L1) ),
               continue(L1).

subst(C,L):- reduct(C,C1),match(C1,L).

reduct(C,C1):-C=..[X,N,mili],C1=..[X,N].
reduct(C,C1):-C=..[X,N,w],C1=..[X,M],M is N*1000.
reduct(C,C1):-C=..[X,N,_],C1=..[X,N].
reduct(C,C).

match(mat(A),[A|_]). 
match(pol(B),[_,_B|_]). 
match(fun(C),[_,_,_,C|_]). 
match(pow(D),[_,_,_,_,D|_]). 
match(vcb(E),[_,_,_,_,_,E|_]). 
match(vce(F),[_,_,_,_,_,_,F|_]). 
match(hfe(G),[_,_,_,_,_,_,_,G|_]). 
match(cap(H),[_,_,_,_,_,_,_,_,H|_]). 

replace(C,L,L1):- subst(C,L1),replace1(L,L1).

replace1(X,X):- var(X),!.
replace1([],[]):- !.

```

```

replace1([H|T],[H1|T1]) :- !, replace1(H,H1), replace1(T,T1).
replace1(_,_).

/* Transistor information */

trans(t2n2219,[si,n,hsa,800,60,30,120,'T05']).
trans(t2n2904,[si,p,hss,300,60,40,120,'T05']).
trans(t2n3055,[si,n,lpa,'115000',100,70,70,'T03']).
trans(t2n3904,[si,n,hss,310,60,40,300,'T092']).
trans(t2n3906,[si,p,hss,310,40,40,300,'T092']).

```

Execution:

To run the system, just perform the command: ‘:-begin.’

```

:-begin.

Characteristics(value)

- mat(si).
- pol(p).
- fun(hss).
- pot(300,mili).
- vcb(60).
- stop.

trans(t2n2904,[si,p,hss,300,60,40,120,T05])

```

Do you want it (yes or no) ?
no.

No transistors available
- fun(hsa).
- pol(n).
- pot(800).
- stop.

trans(t2n2219,[si,n,hsa,800,60,30,120,T05])

Do you want it(yes or no) ?
yes.
New characteristics
- Goodbye.

Problem 154 [Mellish 1977]**Verbal statement:**

Write a program that behaves as a travel agent and holds an interface with a user in order to determine his precise travel requirements and to present him with information enabling him to choose air flights.

Imagine a question-asking strategy determined by the requirement to fill in slots in a system of frames in a depth-first manner. To prevent unnecessary questions imagine a system of default values and a mechanism that can avoid asking questions whose answers have been provided either implicitly or explicitly at some earlier time.

The program must also be able to cope with the client changing his mind by means of dealing with the contradictions arising from his inputs.

Suppose you have the following flight information, already written as a PROLOG database:

```
flight(edinburgh, paris, jan, morn, f1).
flight(edinburgh, paris, jan, aft, f2).
flight(edinburgh, paris, jan, aft, f3).
flight(paris, rome, feb, morn, f4).
flight(paris, rome, feb, morn, f5).
flight(paris, rome, feb, aft, f6).
flight(rome, edinburgh, mar, morn, f7).
```

where `flight(F,T,D,Ti,Fl)` means a flight from F to T in month D at time Ti with number Fl.

Logic program:

```
:op(410, xfy, '.').

/* Overall control of the dialogue */

talk:- default(trip(1),exists),default(trip(2),exists),
       default(homeport(1),edinburgh),
       repeat,dialogue(X), nl,nl,
       display('trips booked :'),nl,nl,
       output(X),nl.

dialogue(X):- nextafter(0,M),tripspecification(M,X).

/* Gathering of trip information */

tripspecification(N,tr(d(A),t(B),f(C),to(D),f1(E),tr(F)).Ts):-
    discover(date(N),A), discover(time(N),B),
    discover(homeport(N),C), discover(foreignport(N),D),
    getflight(C,D,A,B,E,N), discover(traveller(N),F),!,
    continue(D,F,N,Ts).

continue(Dest,Trav,N,Ts):- nextafter(N,M),!,
    default(homeport(M),Dest),
    recall(homeport(1),H), default(foreignport(M),H),
    default(traveller(M),Trav),!,
    tripspecification(M,Ts).

continue(_,_,_,[]).

getflight(F,T,D,Ti,Fl,N):- fact(ok(Fl,N),yes),
    feasible(F,T,D,Ti,Fl,N) , !.
```

```

getflight(F,T,D,Ti,F1,N):- flight(F,T,D,Ti,F1),
    flightok(F1,N,Con), !, Con = 0.
getflight(_,_,_,_,_,_):- display('no flights available'),
    nl,discover(change,Z).

flightok(F1,N,0):- discover(ok(F1,N),R), !, R = yes.
flightok(_,_,1).

feasible(F,T,D,Ti,F1,N):- flight(F,T,D,Ti,F1),!.
feasible(_,_,_,_,N):- retract(fact(ok(F,N),X)),fail.

/* Manipulation of facts */

discover(Lit,Val):- fact(Lit,Val), !.
discover(Lit,Val):- repeat,askclient(Lit,Val,Con), !,
    Con = 0.

default(Lit,Val):- fact(Lit,Newval), !.
default(Lit,Val):- assert(fact(Lit,Val)), !.

recall(Lit,Val):- fact(Lit,Val), !.

/* Communication with user */

askclient(Lit,Val,Con):-
    display(Lit), display(' ?'),nl,
    read(Input), interpret(Input,Con), !,
    notaskagain(Lit,Val,Con), !.

notaskagain(Lit,Val,0):- !, fact(Lit,Val).
notaskagain(_,_,_).

interpret([],0).
interpret(A,B,R):- dealwith(A,S), interpret(B,T),
    R is S + T.

dealwith(A,B,0):- fact(A,B), !.
dealwith(A,B,1):- fact(A,C), retract(fact(A,C)),
    assert(fact(A,B)), !.
dealwith(A,B,[]):- assert(fact(A,B)).

/* Miscellaneous */

nextafter(N,M):- fact(trip(M),exists), N < M,
    P is N + 1, nonebetween(P,M), !.

nonebetween(X,X):- !.
nonebetween(X,Y):- fact(trip(X),exists), !, fail.
nonebetween(X,Y):- Z is X + 1, nonebetween(Z,Y).

output(A,B):- display(A),nl,output(B).
output(_).

/* Flight information */

```

```

flight(edinburgh,paris,jan,morn,f1).
flight(edinburgh,paris,jan,aft,f2).
flight(edinburgh,paris,jan,aft,f3).
flight(paris,rome,feb,morn,f4).
flight(paris,rome,feb,morn,f5).
flight(paris,rome,feb,aft,f6).
flight(rome,edinburgh,mar,morn,f7).

```

Execution:

```

:-talk.
date(1)?
| [trip(3).exists,foreignport(1).paris,foreignport(2).rome].
date(1)?
| [date(1).jan,traveller(1).me].
time(1)?
| [time(1).morn].
ok(f1,1)?
| [ok(f1,1).no].
no flights available
change?
| [time(1).aft,date(2).feb].
ok(f2,1)?
| [ok(f2,1).no].
ok(f3,1)?
| [ok(f3,1).yes].
time(2)?
| [time(2).morn,traveller(2).fred].
ok(f4,2)?
| [ok(f4,2).yes].
date(3)?
| [date(3).mar,trip(1).notexists,ok(f4,2)no].
ok(f5,2)?
| [time(2).aft].
ok(f6,2)?
| [ok(f6,2).yes].
time(3)?
| [time(3).morn].
ok(f7,3)?
| [ok(f7,3).yes].
trips booked:
tr(d(feb),t(aft),f(paris),to(rome),fl(f6),tr(fred))
tr(d(mar),t(morn),f(rome),to(edinburgh),fl(f7),tr(fred))

```

Comments:

This is a good example of a clean program in Prolog. It was designed to compare KRL-0 to Prolog, concerning a precise application, the system GUS. The author proved that frame-like structures (e.g. the trip specification) and the control structures, appropriate or support dialogues, were easily written in Prolog.

Frame: trip specification

Variables	Meaning	Domains	Procedures
A	date	month	discover
B	time	{morning, afternoon}	discover
C	from	town	default
D	to	town	discover
E	flight	{f1, f2, ...}	getflight
F	traveler	name	discover

Problem 155 [Silva; Cotta 1978]

Verbal statement:

In the preceding problem, the program has two ways of achieving a situation in which there is “no flight available”.

The first way corresponds to the inexistence of a flight in the database for the user’s specifications, and the other corresponds to a disagreement about all the flights the program knows for the user’s specifications.

Write a new version of that program where these situations are distinguished. For the first situation output the literals whose values are preventing the attainment of a flight.

Logic program:

The procedure “getflight” has now the following form:

```

getflight(F,T,D,Ti,F1,N):- fact(ok(F1,N),yes),
                           feasible(F,T,D,Ti,F1,N),!.
getflight(F,T,D,Ti,F1,N):- flight(F,T,D,Ti,F1),
                           flightok(F1,N,Con),!, Con=0.
getflight(F,T,D,Ti,F1,N):- (flight(F,T,D,Ti,_),
                           display('no more flights available'),nl,!,
                           discover(change,Z)).
getflight(F,T,D,Ti,F1,N):- display('no flights available'),nl,
                           display('possible changes:'),nl,alterations(F,T,D,Ti).
getflight(_,_,-,-,N):- read(Input),interpret(Input,Con),fail.

```

The new procedure alteration has the form:

```

alterations(F,T,D,Ti):- first(flight(F,T,D,_,_)),
                           display('time'),nl,fail.

```

```

alterations(F,T,D,Ti):- first(flight(F,T,_,Ti,_)),
                     display('date'),nl,fail.
alterations(F,T,D,Ti):- first(flight(F,_,D,Ti,_)),
                     display('foreignport'),nl,fail.
alterations(F,T,D,Ti):- first((flight(F,T,X,Y,_),X\==D,
                               Y\==Ti)),
                     display('time and date'),nl,fail.
alterations(F,T,D,Ti):- first((flight(F,X,D,Y,_),X\==T,
                               Y\==Ti)),
                     display('time and foreignport'),nl,fail.
alterations(F,T,D,Ti):- first((flight(F,X,Y,Ti,_),X\==T,
                               display('date and foreignport'),nl,fail.
alterations(F,T,D,Ti):- first((flight(F,X,Y,Z,_),X\==T,
                               Y\==D, Z\==Ti)),
                     display('time,date and foreignport'),nl,
                     fail.

first(P):- P, !.
first((P,Q)):- P,first(Q),!.

```

Problem 156

Verbal statement:

Specify a virtual relational database for the kinship facts.

Take the following family tree for facts:

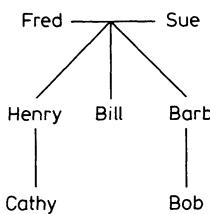


Fig.2

Logic program:

```

grandparent(X,Z):- grandfather(X,Z); grandmother(X,Z).
grandfather(X,Z):- father(X,Y),parent(Y,Z).
grandmother(X,Z):- mother(X,Y),parent(Y,Z).
parent(X,Y):- father(X,Y); mother(X,Y).
brother(X,Y):- male(X),parent(Z,X),parent(Z,Y),X\==Y.

```

```

sister(X,Y):- female(X),parent(Z,X),parent(Z,Y),X\==Y.

uncle(X,Y):- male(X),parent(Z,X),grandparent(Z,Y).

aunt(X,Y):- female(X),parent(Z,X),grandparent(Z,Y).

cousin(X,Y):- parent(Z,X),parent(T,Y),
             (brother(Z,T);sister(Z,T)).

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

wife(X,Y):- husband(Y,X).

father(fred,henry).      mother(sue,henry).
father(fred,bill).       mother(sue,bill).
father(fred,barb).        mother(sue,barb).
father(henry,cathy).     mother(barb,bob).

male(fred).               female(barb).
male(bob).                female(sue).
male(henry).              female(cathy).
male(bill).

```

Problem 157 [Townshend 1979]

Verbal statement:

Write a program to implement a pre-registration appointments matching scheme. It covers the allocation of the pre-registration six-month surgical and medical appointments that the law requires (here called jobs, or “posts” with a particular consultant’s firm) to newly qualified doctors (here referred to as applicants).

A general representation of a job might be: the ‘job number’, the special restrictions on the applicants to be allocated to this job, the numbers of posts available in the six-month periods starting in August and February, the ‘job preference list’, the list of the reference numbers of applicants for the posts in that job, ranked by the consultant in charge in the order of his preference and the result lists.

The corresponding record for an applicant might be: the ‘job number’, the sex, the season, the preference lists and the corresponding result lists.

The problem of assigning the members of two sets to one another is dealt with according to the following definition of a ‘stable marriage’. “Consider two distinct sets A and B. An assignment of the members of A to the members of B is said to be a stable marriage if and only if there exist no elements a and b (belonging to A and B respectively) who are not assigned to each other but who would both prefer each other to their present partners”.

Logic program:

```

/* PRAMS PART 1 in PROLOG */

program:-jobs(Jobslist),apps(Appslist),
         prepare(Jobslist),
         alloc1(Joblist,Applist,Jobslist).

```

```

alloc1([Job|JL], Appslist, Jobslist) :-  

    offer(Job, Appslist, Jobslist),  

    alloc1(JL, Appslist, Jobslist).  

alloc1([], Appslist, Jobslist) :- show(Jobslist).  

offer(Job(J, Lim, NA, NF, Jprefs, As, Af, All), Appslist, Jobslist) :-  

    write(J),  

    N is NA+NF,  

    offer1(J, Jprefs, N, 0, All, Appslist, Jobslist).  

offer1(_, [], _, _, _, _).  

offer1(_, _, N, N, _, _).  

offer1(J, [A|Jprefs], N, K, All, Appslist, Jobslist) :-  

    member(A, All),  

    accept(A, J, Appslist, Jobslist),  

    K1 is K+1,  

    offer1(J, Jprefs, N, K1, All, Appslist, Jobslist).  

offer1(J, [A|Jprefs], N, K, All, Appslist, Jobslist) :-  

    offer1(J, Jprefs, N, K, All, Appslist, Jobslist).  

accept(A, J, Appslist, Jobslist) :-  

    member(app(A, _, _, Maprefs, X, Saprefs, Y), Appslist),  

    accept1(A, Maprefs, Saprefs, X, Y, J, Appslist, Jobslist).  

accept1(A, Maprefs, Saprefs, X, Y, J, Appslist, Jobslist) :-  

    member(J, Maprefs), !,  

    accept2(A, J, Maprefs, X, Appslist, Jobslist).  

accept1(A, Maprefs, Saprefs, X, Y, J, Appslist, Jobslist) :-  

    member(J, Saprefs), !,  

    accept2(A, J, Saprefs, Y, Appslist, Jobslist).  

accept2(_, J, _, X, _, _) :- nonvar(X), X = J, !.  

accept2(_, J, _, X, _, _) :- nonvar(X), !, fail.  

accept2(A, J, Prefs, X, Appslist, Jobslist) :-  

    accept3(A, J, Prefs, X, Appslist, Jobslist).  

accept3(_, J, [J1|_], J, _, _).  

accept3(A, J, [J1|Prefs], X, Appslist, Jobslist) :-  

    not(ask(A, J1, X, Appslist, Jobslist)),  

    accept3(A, J, Prefs, X, Appslist, Jobslist).  

ask(A, J, J, Appslist, Jobslist) :-  

    member(Job(J, Lim, NA, NF, Jprefs, As, Af, All), Jobslist),  

    N is NA+NF,  

    ask1(A, J, Jprefs, N, All, Appslist, Jobslist).  

    ask1(A, J, Jprefs, N, All, Appslist, Jobslist) :-  

        ismember(A, All).  

ask1(A, J, Jprefs, N, All, Appslist, Jobslist) :-  

    offer1(J, Jprefs, N, 0, All, Appslist, Jobslist), !,  

    ismember(A, All).

```

```

prepare([Job(No,Lim,NA,NF,Prefs,As,Af,All)|JL]:-
    makelist(NA,[],As),
    makelist(NF,[],Af),
    N is NA+NF,
    makelist(N,[],All),
    prepare(JL).

prepare([]).

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

ismember(X,[Y|_]):-nonvar(Y), X=Y, !.
ismember(X,[_|L]):-ismember(X,L).

jobs([
    job(1,0,1,1,[1,9,4,6,2,7,3,5,8],-,-,-),
    job(2,0,1,1,[3,2,1,5,7,6,10],-,-,-),
    job(3,0,1,0,[9,6,7,5,3,4,1,2],-,-,-),
    job(4,0,1,2,[4,7,1,6,2,5,3,10],-,-,-),
    job(5,0,2,2,[1,4,8,3,2,5,10,7,6],-,-,-),
    job(6,0,3,3,[9,10,4,6,5,3,2,1],-,-,-),
    job(7,0,2,2,[1,2,9,10,6,4,3,5],-,-,-),
    job(8,0,3,0,[7,8,1,2,6,4,3,5],-,-,-),
    job(9,0,0,3,[4,3,8,10,9],-,-,-)
]).

apps([
    app(1,m,0,[1,2,3,4],-[5,6,7,8],-),
    app(2,f,0,[1,4,2,3],-[6,7,8,5],-),
    app(3,m,1,[1,3,2,4],-[5,8,7,9],-),
    app(4,m,0,[1,3,4,9],-[6,5,8,7],-),
    app(5,m,2,[3,1,4,2],-[5,6,8,7],-),
    app(6,f,3,[4,1,2,3],-[7,5,6,8],-),
    app(7,m,0,[2,4,1,3],-[8,5],-),
    app(8,m,0,[1],-[5,8,9],-),
    app(9,f,1,[3,1],-[6,7,9],-),
    app(10,m,0,[2,4,9],-[5,6,7],-)
]).
```

Execution:

```
?-program.
123456789
```

RESULTS

```

1   [ 1 4 ]
2   [ 3 5 ]
3   [ 9 ]
4   [ 7 6 2 ]
```

```

5  [ 1 8 3 5 ]
6  [ 9 10 4 2 _ _ ]
7  [ 6 _ _ _ ]
8  [ 7 _ _ ]
9  [ 10 _ _ ]

```

Problem 158 [Dahl 1980, personal communication]

Verbal statement:

Assume that procedures and queries are respectively noted:

```

axiom(A,[A1,A2|An]):-
?- prove([G1,G2|Gn])

```

instead of ‘A:- A1,..,An’ and ‘?- G1,..,Gn.’

Assume that a predicate of the form ‘delay(P,N)’ has been defined, which associates a delay value ‘n’ to a subgoal ‘p’.

Define the predicate ‘prove(Q)’, so that the subgoals in Q are executed in the order implied by ‘delay’.

Logic program:

```

prove(Q):- pr(0,Q).
pr(0,[]):- !.
pr(N,Q):- solve(N,Q,Q1,n),pr(n,Q1).
solve(N,[Q,Q1],Q,0):- delay(S,N1),equal(N1,N),!,
                     axiom(S,Q2),concatenate(Q2,Q1,Q).
solve(N,[Q,Q1],[S,Q2],n):- solve(N,Q,Q2,n).
solve(N,[],[],n+1):- not(maxdelay(N)),!.
solve(N,[],[],n):- write('errors'),fail.

```

Chapter 17 Text Formatting with Prolog

This chapter illustrates the use of Prolog features for word processing. The problems were selected from the book by (Welsh; Elder 1979), an introduction to PASCAL. The idea is to apply Prolog in a rather devious application domain, in what concerns the intended capabilities of logic programming. The general aim consists in testing whether Prolog can be compared to PASCAL in a matter where this programming language is quite able. Also, another intention is involved: to check how Prolog behaves in commercial applications, too involved with file manipulation, and often related to symbolic processing. We are convinced that such exercises are rather desirable because if Prolog goes commercial diverse situations arise in the design of real programs, and some knowledge about the ways Prolog copes and interacts with other special purpose programming languages is required.

Problem 159

Verbal statement:

A program is required to examine a piece of text and produce a list, in alphabetical order, of all the distinct words which appear in the text, e.g. the examination of the input

the black dog chased the black cat

would produce the corresponding output

```
black
cat
chased
dog
the
```

Logic program:

```
concordance(L2,L):- sort(L2,L),nl,format(L).
format([X|R]):- tab(2),write(X),nl,format(R)
format([]).
```

Execution:

```
?- concordance([the,black,dog,chased,the,black,cat]).
```

Problem 160

Verbal statement:

In printing text the textual information must be formatted to suit the available size of the printing device involved. A program is required to read a series of lines of text, each line containing not more than 80 characters, and output this text as a series of lines containing not more than 60 characters. The input text consists of words (of not more than 16 letters), commas, semicolons, full stops and layout devices such as blank lines and new paragraphs. Any non-blank line having three or more leading blanks is considered to be the start of a new paragraph.

Logic program:

```

format(F):- instructions, sentence(F), format1(F), retract_all.

instructions:- nl, write('INSTRUCTIONS'), nl,
              write('Write your text after character prompt ":"'),
              nl, write('Press "CR" for changing to new line'), nl,
              write('Write "*" after "CR" for paragraph'),
              nl, write('For ending the text press "#"'), nl, nl.

sentence(F):- get0(C), words(C,F).

words(C,[P|Ps]) :- letter(C), word(C,C1,L),
                  name(P,L), words(C1,Ps).

words(9,[' '|Ps]) :- get0(C1), words(C1,Ps).
words(31,['@'|Ps]) :- get0(C1), words(C1,Ps).
words(32,[' '|Ps]) :- get0(C1), words(C1,Ps).
words(34,['''|Ps]) :- get0(C1), words(C1,Ps).
words(35,[#]). 
words(36,['#'|Ps]) :- get0(C1), words(C1,Ps).
words(38,['&'|Ps]) :- get0(C1), words(C1,Ps).
words(40,['('|Ps]) :- get0(C1), words(C1,Ps).
words(42,['*'|Ps]) :- get0(C1), words(C1,Ps).
words(_,P) :- get0(C), words(C,P).

word(C,C1,[C|Cs]) :- get0(C2), (letter(C2), word(C2,C1,Cs);
                                C1=C2, Cs=[ ]).

word(33,'!', [C|Cs]) :- C1=C2, Cs=[ ], name(P,L), sentence(F).
word(37,'%', [C|Cs]) :- C1=C2, Cs=[ ], name(P,L), sentence(F).
word(41,',',[C|Cs]) :- C1=C2, Cs=[ ], name(P,L), sentence(F).
word(44,';',[C|Cs]) :- C1=C2, Cs=[ ], name(P,L), sentence(F).
word(45,'-',[C|Cs]) :- C1=C2, Cs=[ ], name(P,L), sentence(F).
word(46,'.',[C|Cs]) :- C1=C2, Cs=[ ], name(P,L), sentence(F).
word(58,:',[C|Cs]) :- C1=C2, Cs=[ ], name(P,L), sentence(F).
word(59,';',[C|Cs]) :- C1=C2, Cs=[ ], name(P,L), sentence(F).
word(63,'?', [C|Cs]) :- C1=C2, Cs=[ ], name(P,L), sentence(F).

```

```

letter(9):- !, fail.          /*      tab      */
letter(10):- !, fail.
letter(13):- !, fail.
letter(31):- !, fail.          /*      return    */
letter(32):- !, fail.          /*      space    */
letter(34):- !, fail.          /*      "        */
letter(35):- !, fail.          /*      -        */
letter(36):- !, fail.          /*      $        */
letter(38):- !, fail.          /*      &       */
letter(40):- !, fail.          /*      (        */
letter(42):- !, fail.          /*      *        */
letter(_).

format1(F):- title,word1(F).

title:- nl,nl,write(' ** TEXT ** '),nl.nl.

word1([X|R]):- name(X,L1),length(L1,N),(addition(N1);N1=0),
              N2 is N1+N,addition(X,N,N2),word1(R).
word1([]).

addition(X,N,N2):- (test(X));
                  ((N2>61,nl,write(X),N1=N,
                   asserta(addition(N1)));
                   (N2=<61,write(X),N1=N2,
                    asserta(addition(N1)))). 

test(@):- tab(1).
test(#).
test(*):- nl,N1=0,asserta(addition(N1)).
test(_):- !, fail.

retract_all:- retract(addition(N1)),fail.
retract_all.

```

Execution:

```
| ?- format(_).
```

INSTRUCTIONS

Write text after character prompt “|:”

Press “CR” for changing to new line

Write “*” after “CR” for paragraph

For ending the text press “#”

```
|: PROLOG**
```

```
|:     Prolog is a simple but powerful programming language.*  

|:     Clear, readable, concise programs can be written quickly  

|:     with few errors.* #  

**      TEXT      **
```

PROLOG

Prolog is a simple but powerful programming language.
 Clear, readable, concise programs can be written quickly with few errors.

Problem 161

Verbal statement:

An editor assists the user in the development of files of textual information. Corrections and amendments to the data in an existing file are specified by means of simple instructions. The editor obeys these instructions in editing the contents of an existing oldfile to produce a newfile.

- 1) Write a simple editor in Prolog that makes use of at least four types of editing commands: T (transcription), D (delete), I (insert) and E (end).
- 2) Write a program able to open windows on the terminal screen. The program must be able to unify the current coordinates of the cursor, scroll the text within an area of the screen, write a string in the previous scrolled area and return the cursor to the previous coordinates. Think of a solution within the context of the Arity-Prolog, where some built-in predicates make more concise the final clause.

Logic program 1:

```

edit:- message_0,copy_file,initialize,get_commands.
copy_file:- asserta(number_of_lines(0)),see(newfile),
           tell(oldfile),
           spyer.

spyer:- get0(C),(s_character(C);(put(C),spyer)).

s_character(26):- (retract(contents(0)),nl,seen);
                 (seen,told).
s_character(31):- retract(number_of_lines(Nf)),Nf1 is Nf+1,
                 confirm_1(Nf1),put(31),spyer.
s_character(_):- !, fail.

initialize:- C1=0,asserta(current_line(C1)).

get_commands:- read(T),((functor(T,C,1),command(T,C));
                         (command(T));(message_1)).

command(T,d):- arg(1,T,N),((integer(N),((N=<0,message_3);
                                         (current_line(C1),
                                         number_of_lines(Nf),C1 is C1+N,
                                         ((C1=<Nf,confirm_2(C1));message_4))),
                                         message_2);message_1).

command(T,k):- arg(1,T,N),((integer(N),asserta(number(N)),
                           (N=<0,message_3);(current_line(C1),
                           
```

```

number_of_lines(Nf),Cl1 is Nf-Cl,
((N>Cl1,message_4);(asserta(trans_line(1)),
see(oldfile),tell(newfile),
(Cl=0;transcription),Tl=1,
delete(Tl)))))),message_2);
(message_1)).
command(T,u):- arg(1,T,N),((integer(N),((N=<0,message_3);
(current_line(Cl),
number_of_lines(Nf),Cl1 is Cl-N,
((Cl1>=0,confirm_2(Cl1));message_4))),,
message_2);message_1).

command(_,_):- !, fail.

command(e):- number_of_lines(Nf),Cl is Nf-1,
asserta(current_line(Cl)),message_2.

command(h):- help.

command(i):- copy_lines.

command(l):- current_line(Cl),
see(newfile),(Cl=0;
(Tl=1,jump(Tl))),nl,print,message_2.

command(p):- nl,see(oldfile),asserta(contents(0)),
asserta(number_of_lines(0)),spyer,
message_2.

command(t):- initialize,message_2.

command(x):- nl,write('EDITTEXT WAS PLEASED TO SERVE YOU.'),nl,nl.

command(_):- !, fail.

transcription:- get0(C),((t_character(C),put(C),transcription);
put(C)). 

t_character(31):- trans_line(Tl),current_line(Cl),
((Tl<Cl,Tl1 is Tl+1,confirm_3(Tl1));
(!,fail)).
t_character(_).

delete(Tl):- skip(31),number(N),((N=Tl,spyer,copy_file);
(Tl1 is Tl+1,confirm_4(Tl1))).

help:- nl,write('SET OF EDIT COMMANDS :'),nl,
nl,write('*** d(n).           down_lines'),
nl,write('*** u(n).           up_lines'),
nl,write('*** k(n).           kill lines'),
nl,write('*** h.              print instructions'),
nl,write('*** i.              insert'),
nl,write('*** t.              top of file'),
nl,write('*** e.              end of file'),
nl,write('*** l.              print current line'),
nl,write('*** p.              print all file'),
nl,write('*** x.              exit'),
nl,nl,message_2.

```

```

copy_lines:- current_line(Cl), asserta(counter(0)),
            ((Cl=0, tell(newfile), insert,
              see(oldfile));(see(oldfile), tell(newfile),
              asserta(trans_line(1)), transcription, seen,
              insert, see(oldfile), Tl=1, jump(Tl))),
            counter(I), Cl1 is Cl+I, confirm_2(Cl1),
            spyer, copy_file, message_2.

insert:- get0(C), (i_character(C);(put(C), insert)).

i_character(27).

i_character(31):- counter(I), I1 is I+1, confirm_5(I1),
                put(31), insert.
i_character(_):- !, fail.

jump(T1):- skip(31), current_line(Cl), (Cl=T1;
                                         (Tl1 is Tl+1, confirm_6(Tl1))). 

print:- get0(C), ((p_character(C), put(C), print);
                  (nl, nl, seen)). 

p_character(31):- !, fail.
p_character(_).

confirm_1(Nf1):- Nf=Nf1, asserta(number_of_lines(Nf)).
confirm_2(Cl1):- Cl=Cl1, asserta(current_line(Cl)).
confirm_3(Tl1):- Tl=Tl1, asserta(trans_line(Tl)).
confirm_4(Tl1):- Tl=Tl1, delete(Tl).
confirm_5(I1):- I=I1, asserta(counter(I)).
confirm_6(Tl1):- Tl=Tl1, jump(Tl).

message_0:- nl, write('EDITTEXT AT YOUR SERVICE.'),
            nl, nl.
message_1:- nl, write('EDIT COMMAND NOT RECOGNIZED.'), nl,
            nl, message_2.
message_2:- get_commands.
message_3:- nl,
            write('ZERO AND NEGATIVE INTEGERS AREN'T ALLOWED.'),
            nl, nl.
message_4:- nl, write('OLD FILE EXHAUSTED PREMATURELY.'), 
            nl, nl.

```

Execution:

```

| ?- edit.

EDITTEXT AT YOUR SERVICE.

| : p.

```

MESSAGES:

-- command errors
-- dimension limits.

|: h.

SET OF EDIT COMMANDS:

*** d(n). down_lines
*** u(n). up_lines
*** k(n). kill lines
*** h. print instructions
*** i. insert
*** t. top of file
*** e. end of file
*** l. print current line
*** p. print all file
*** x. exit

|: a.

EDIT COMMAND NOT RECOGNIZED.

|: b(3).

EDIT COMMAND NOT RECOGNIZED.

|: c(h).

EDIT COMMAND NOT RECOGNIZED.

|: d(0).

ZERO AND NEGATIVE INTEGERS AREN'T ALLOWED.

|: d(-4).

ZERO AND NEGATIVE INTEGERS AREN'T ALLOWED.

|: d(4).

OLD FILE EXHAUSTED PREMATURELY.

|: u(0).

ZERO AND NEGATIVE INTEGERS AREN'T ALLOWED.

|: u(-8).

ZERO AND NEGATIVE INTEGERS AREN'T ALLOWED.

|: u(4).

```
OLD FILE EXHAUSTED PREMATURELY.  
|: k(0).  
  
ZERO AND NEGATIVE INTEGERS AREN'T ALLOWED.  
|: k(-12).  
  
ZERO AND NEGATIVE INTEGERS AREN'T ALLOWED.  
|: k(4).  
  
OLD FILE EXHAUSTED PREMATURELY.  
|: p.
```

MESSAGES:

```
-- command errors  
-- dimension limits.
```

```
|: x.  
  
EDITTEXT WAS PLEASED TO SERVE YOU.  
| ?- edit.
```

EDITTEXT AT YOUR SERVICE.

```
|: p.
```

Here, we have an example to show the commands used to place the pointer in any spot of the text before any desired correction is triggered.

```
|: l.  
  
Here, we have an example to show the commands used
```

```
|: d(1).  
|: l.
```

to place the pointer in any spot of the text

```
|: e.  
|: l.
```

before any desired correction is triggered

```
|: u(1).  
|: l.
```

to place the pointer in any spot of the text

```
|: t.  
|: l.
```

Here, we have an example to show the commands used

|: x.

EDITTEXT WAS PLEASED TO SERVE YOU.

| ?- edit.

EDITTEXT AT YOUR SERVICE.

|: p.

How the
user
must delete
and insert
pieces of
text.

|: d(1).
|: k(2).
|: i.
|: delete
|: p.

How to
delete and insert
pieces of
text.

|: d(1).
|: l.

pieces of

|: k(1).
|: p.

How to
delete and insert
text.

|: e.
|: i.
|: um
: u(2).
|: l.

How to

|: k(2).
|: i.
|: Commands for
|: delete and insert
|: p.

Commands for
delete and insert
a text.

```
|: x.
```

EDITTEXT WAS PLEASED TO SERVE YOU.

Comments:

The editing instructions make use of a conceptual line-number counter which references a particular line in the oldfile. The lines of a file are numbered from zero upwards. The conceptual line-number counter is assumed initially to be zero, i.e. it points to the first line.

The editor behind the above program allows the use of ten commands: D (increases the counter), U (decreases the counter), K (kills lines), I (inserts lines), T (top of the file), E (end of the file), H (helps showing the available commands), L (prints the current line), P (prints the file), and X (exit from the editor).

At the end of each instruction implying text changes, the two files (newfile and oldfile) have the same content. This is achieved through “copy-file”.

Logic program 2 [Viccarri 1986, personal communication]:

```
window:-  
    tget(Row,Column),  
    tscroll(0,(R,C),(R,C)),  
    tmove(R1,C1),  
    write('string'),  
    tmove(Row,Column).
```

Comments:

In order to generate a window on the VDU screen it is necessary: (1) to keep the actual position of the cursor ('tget'), (2) to clean the screen area ('tscroll'), (3) to set the cursor in this area ('tmove'), (4) to write the desired message ('write'), and (5) to return to the cursor initial position ('tmove').

Chapter 18 Management with Prolog

The business world (management information systems and financial communities) has become attracted to Artificial Intelligence on account of its new way of approaching old problems. Also, AI has started to be considered as a new tool for doing things that could not have been done before.

Within this chapter we present examples of traditional management application that could be programmed in Pascal, and also some problems about planning and simulation models.

Problem 162

Verbal statement:

A wholesale distribution firm maintains a file, known as the stock file, in which there is one record for each type of item stored. When the quantity in stock plus the quantity on order for any item falls below its reorder point, an order for further supplies of the item must be raised to restore the quantity available to the value target stock. The records in the stock file are ordered in ascending sequences of item number. Each day the stock file must be updated, i.e. a new stock file must be produced which reflects the transactions (deliveries and dispatches) which taken place since the last update. These transactions are accumulated on another file, each transaction representing one delivery or dispatch of items of a given type. The records on the transaction file are also sorted in ascending item-number order (the next problem covers how such a sorted file might be obtained from the unsorted data collected each day).

A program is required which updates the current stock file, according to the contents of the transaction file, to produce a new stock file. Items which are below their reorder point after that day's transactions are to be printed as an output order list.

Logic program:

```
stock_update:- nl,nl,begin,user(In,A),order_all(I),scribe.  
begin:- tab(22),write('** OLD STOCK **'),nl,  
       write('item(  
              Item number,In stock,On order,Target stock,' ),  
       write('Reorder point)'),
```

```

nl, see(stock), listing(item), seen, nl, nl,
write('
    The program will be waiting for input of a transaction.
'),
nl, nl, write('EXAMPLE:'), nl,
write('!:delivery(Item number,Amount).'), nl,
write('!:dispatch(Item number,Amount).'), nl, nl,
write('When you finish the transactions,
        just type any letter '),
write('followed by a full-stop.'), nl, nl,
tab(20), write('** TRANSACTIONS **'), nl, nl.

user(In,A):- ttyflush, read(T), ((functor(T,Tt,2), arg(1,T,In),
                                arg(2,T,A)), transaction(Tt,In,A)); option).

transaction(delivery, In, A):- ((item(In,Is,Oo,Ts,Rp),
                                  I is Is+A, O is Oo-A, Io is I+Oo,
                                  ((Io=<Rp, On is Ts-I,
                                  ((retract(order(On,In)),
                                  assert(order(On,In))))),
                                  (assert(order(On,In)))),
                                  retract(item(In,Is,Oo,Ts,Rp)),
                                  assert(item(In,I,On,Ts,Rp)));
                                  (Io>Rp,
                                  retract(item(In,Is,Oo,Ts,Rp)),
                                  assert(item(In,I,O,Ts,Rp)))));
                                  rejected(In)), user(_,_)).

transaction(dispatch, In, A):- ((item(In,Is,Oo,Ts,Rp),
                                  I is Is-A, Io is I+Oo,
                                  ((Io=<Rp, On is Ts-I,
                                  ((retract(order(On,In)),
                                  assert(order(On,In))))),
                                  (assert(order(On,In)))),
                                  retract(item(in,Is,Oo,Ts,Rp)),
                                  assert(item(In,I,On,Ts,Rp)));
                                  (Io>Rp,
                                  retract(item(In,Is,Oo,Ts,Rp)),
                                  assert(item(In,I,Oo,Ts,Rp)))));
                                  rejected(In)), user(_,_)).

rejected(In):- assertz(rejected(In)).

option:- (nl, rejection(In), order(On, In)); (nl, rejection(In));
          (nl, order(On, In)); nl.

rejection(In):- retract(rejected(In)),
              write('Transaction rejected for item '),
              write(In), nl, fail.

rejection.

```

```

order(On, In):- retract(order(On, In)), write('Order '),
              write(On), write(' of item '),
              write(In), nl, fail.

order.

order_all(item(_, _, _, _, _)):- retract(item(In, Is, Oo, Ts, Rp)),
((item(Ic),
  compair(In, Ic,
          item(In, Is, Oo, Ts, Rp)));
(Ic is In, assert(item(Ic)),
 tell(stock), write('['), write(In),
 assertz(item(In, Is, Oo, Ts, Rp)),
 order_all(I))).
```

```

compair(In, Ic, item(In, Is, Oo, Ts, Rp)):- (Ic=In, write('.').), told,
                                         assertz(
                                         item(In, Is, Oo, Ts, Rp)));
                                         (tell(stock), write(','), write(In),
                                         assertz(
                                         item(In, Is, Oo, Ts, Rp)),
                                         order_all(I)).
```

```

scribe:- see(stock), read(T), sort(T, L), seen, scribe_all(L, I),
         told, end.
```

```

scribe_all([In|R], item(_, _, _, _, _)):- tell(stock),
                                         retract(
                                         item(In, Is, Oo, Ts, Rp)),
                                         write(item(In, Is, Oo, Ts, Rp)),
                                         write('.'), nl,
                                         scribe_all(R, I).
```

```

scribe_all([ ], I).
```

```

end:- retract(item(Ic)), nl, consult(stock), nl, nl, tab(22),
       write('** NEW STOCK **'),
       see(stock), listing(item), seen, nl.
```

Execution:

```

| ?- stock_update.
               ** OLD STOCK **

item(Item number,In stock,On order,Target stock,Reorder point)
item(11081,-1080,8080,7000,5000).
item(11202,2930,5070,8000,6000).
item(23934,4230,4770,9000,7000).
item(28454,4770,0,5000,4000).
item(36666,9090,0,9500,7000).
```

```
item(37775,820,1000,2000,1500).
item(39399,4620,0,6000,4500).
item(42000,3550,1400,5000,4000).
item(42111,11460,-3500,8000,6000).
item(42281,3750,1500,6000,4500).
item(43327,8260,0,9000,7000).
item(53553,8000,0,9000,7000).
item(55376,2800,2000,5000,4000).
item(57862,1370,1150,3000,2000).
item(59097,3540,1460,5000,4000).
```

The program will be waiting for input of a transaction.

EXAMPLE:

```
|:delivery(Item number,Amount).
|:dispatch(Item number,Amount).
```

When you finish the transactions, just type any letter followed by a full stop.

** TRANSACTIONS **

```
|: delivery(10754,1000).
|: dispatch(11081,350).
|: dispatch(11081,1240).
|: dispatch(11202,1500).
|: delivery(29334,500).
|: delivery(42111,3500).
|: dispatch(11081,2000).
|: delivery(57862,1150).
|: e.
```

Transaction rejected for item 10754

Transaction rejected for item 29334

Order 11670 of item 11081

stock consulted 219words 0.32 sec.

** NEW STOCK **

```
item(11081,-4670,11670,7000,5000).
item(11202,1430,5070,8000,6000).
item(23934,4230,4770,9000,7000).
item(28454,4770,0,5000,4000).
item(36666,9090,0,9500,7000).
item(37775,820,1000,2000,1500).
item(39399,4620,0,6000,4500).
item(42000,3550,1400,5000,4000).
item(42111,14960,-7000,8000,6000).
item(42281,3750,1500,6000,4500).
item(43327,8260,0,9000,7000).
```

```
item(53553,8000,0,9000,7000).
item(55376,2800,2000,5000,4000).
item(57862,2520,0,3000,2000).
item(59097,3540,1460,5000,4000).
```

Comments:

The program produces a transaction file whose components were sorted in order of ascending item number, from an initially unsorted transaction file. Such a file sort is a common requirement in file manipulation.

Problem 163**Verbal statement:**

A program is required which will sort the records in the unsorted file datafile so that the records are stored in ascending order of the value of the field key of each record. Take into account the previous problem and the corresponding program.

Logic program:

```
merge:- nl,nl,begin,first_user,unsorted_keys,natural_merge.

begin:- write('The program will be waiting for input of
        a transaction.'),nl,nl,write('EXAMPLE'),
        nl,write('!:delivery(Item number,Amount).'),nl,
        write('!:dispatch(Item number,Amount).'),nl,nl,
        write('After each sequence of transactions type "[]." and
        after all the transactions type "end."'),nl,nl,
        tab(20),write('TRANSACTIONS'),nl,nl.

first_user:- tell(keys),write('[['),read(T),arg(1,T,In),
            write(In),user(_).

user(In):- ttyflush,read(T),(test(T);(arg(1,T,In),write(','),
            write(In),user(_))). 

test([]):- write('],['),read(T),
            arg(1,T,In),write(In),user(_).
test(end):- write(']]').),told.
test(_):- !, fail.

unsorted_keys:- nl,nl,write('UNSORTED RECORD KEYS'),nl,nl,
               see(keys),read(K),scribe(K),seen.

scribe([L|R]):- first_scribe(L),scribe(R).
scribe([ ]). 

first_scribe([X|X1]):- write(X),nl,first_scribe(X1).
first_scribe([ ]). 
```

```

natural_merge:- see(keys),read(K),
              (K=[[L]],sort_list(K,C),end(K));
              (merge1(k)).

merge1(K):- (verify_end(K),end(K));
           (N=1,asserta(number(N)),split(K,A,B),
            merge_set(A,B),retract(element(C)),D=C,aux(D)).

split([L|R],[L|A],B):- number(N),0 is N mod 2,retract(number(N)),
                     N1 is N+1,confirm(N1),split(R,A,B).
split([L|R],A,[L|B]):- number(N),1 is N mod 2,retract(number(N)),
                     N1 is N+1,confirm(N1),split(R,A,B).
split([ ],[ ],[ ]):- retract(number(N)).

confirm(N1):- N=N1,asserta(number(N)).

merge_set([A1|As],[B1|Bs]):- concatenate(A1,B1,K),order(K),
                           merge_set(As,Bs).

merge_set([ ],[ ]).
merge_set([ ],Bs):- concatenate1([ ],Bs,K),order(K).
merge_set(As,[ ]):- concatenate1(As,[ ],K),order(K).

concatenate([A2|R],B1,[A2|R1]):- concatenate(R,B1,R1).
concatenate([ ],L,L).

concatenate1([ ],[B2],B2).

order(K):- sort_list(K,C),assertz(element(C)).

sort_list([L|R],C):- sort_list(R,T),insert(L,T,C).
sort_list([ ],[ ]).

insert(X,[L|R],[L|T]):- L<X,!,insert(X,R,T).
insert(X,T,[X|T]). 

aux(D):- (retract(element(C)),E=C,union(D,D,E,K),aux1(K));
         (K=D,merge1(K)).

aux1(K):- D=K,aux(D).

union(D,[D1|Ds],E,[K1|Ks]):- (integer(D1),K=[K1,Ks],K1=D,Ks=[E]);
                                (K1=D1,
                                 (Ds=[ ],Ks=[E]);union(D,Ds,E,Ks))). 

verify_end([L|R]):- integer(L),seen.

end(K):- nl,write('SORTED RECORD KEYS'),nl,nl,scribe1(K).

scribe1([L|R]):- write(L),nl,scribe1(R).
scribe([ ]).

```

Execution:

| ?- merge.

The program will be waiting for input of a transaction.

EXAMPLE

```
|:delivery(Item number,Amount).  
|:dispatch(Item number,Amount).
```

After each sequence of transactions type “[].” and after all the transactions type “end.”

TRANSACTIONS

```
|: delivery(57862,500).  
|: delivery(29334,760).  
|: dispatch(42111,950).  
|: [].  
|: delivery(42000,890).  
|: delivery(36666,870).  
|: [].  
|: delivery(11202,350).  
|: dispatch(11081,500).  
|: delivery(55376,800).  
|: [].  
|: delivery(28434,750).  
|: [].  
|: dispatch(23934,500).  
|: delivery(42111,780).  
|: delivery(11081,545).  
|: delivery(55376,900).  
|: [].  
|: delivery(28454,890).  
|: dispatch(42281,850).  
|: [].  
|: delivery(37775,890).  
|: delivery(10754,950).  
|: [].  
|: dispatch(59907,760).  
|: end.
```

UNSORTED RECORD KEYS

57862
29334
42111
42000
36666
11202
11081
55376
28434
23934
42111

```
11081
55376
28454
42281
37775
10754
59907
```

SORTED RECORD KEYS

```
10754
11081
11081
11202
23934
28434
28454
29334
36666
37775
42000
42111
42111
42281
55376
55376
57862
59907
```

```
yes
| ?- merge.
```

The program will be waiting for input of a transaction.

EXAMPLE

```
| :delivery(Item number,Amount).
| :dispatch(Item number,Amount).
```

After each sequence of transactions type “[.]” and after all the transactions type “end.”

TRANSACTIONS

```
| : delivery(11081,780).
| : delivery(11202,560).
| : dispatch(54567,890).
| : delivery(54545,750).
| : end.
```

UNSORTED RECORD KEYS

11081
11202
54567
54545

SORTED RECORD KEYS

11081
11202
54545
54567

Problem 164 [Lee 1985b]

Consider the following plan for building a house:

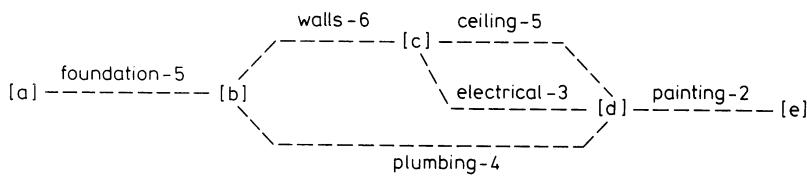


Fig. 1

- 1) Describe the above PERT diagram by Prolog assertions.
- 2) Compute the critical path.

Logic program:

```

/* PERT diagram */

activity(a,b,foundation,5).
activity(b,c,walls,6).
activity(b,d,plumbing,4).
activity(c,d,ceiling,5).
activity(c,d,electrical,3).
activity(d,e,painting,2).

/* Critical path */

critical(X,X,[],0).
critical(X,Z,Total_Time,Activity_List):-
    setof((T,Y,[A|L]),
          (T1,T2)^(activity(X,Y,A,T1),critical(Y,Z,L,T2),
          T is (T1+T2),SS),
          maxof(SS,(Total_Time,_,Activity_List))).
  
```

```
maxof([X], X):- !.
maxof([X|L], Y):- maxof(L, Y), Y@>X, !.
maxof([X|L], X).
```

Comments:

In a PERT diagram, multiple arcs emanating from a node indicate not a choice but rather concurrency of the indicated activities. Thus, work on the walls and the plumbing can proceed in parallel, once the foundation has been laid. Likewise, both the ceiling and electrical work can be done in parallel once the walls are done. All these must be completed before painting can begin.

In a PERT diagram nodes no longer represent states of the entire system. They represent ‘sub-states’, indicating a momentary status of only part of the system. The nodes are viewed as instantaneous transitions between two or more activities. The total state of the system is therefore given as the set of activities that are currently being performed. The critical path is found as the maximum length path of activities between two nodes.

Problem 165 [Lee 1985b]

Verbal statement:

Consider the case of two machines in sequence, controlled by a single operator. Thus each machine must wait while the other is processing. A sketch of the system is as follows:

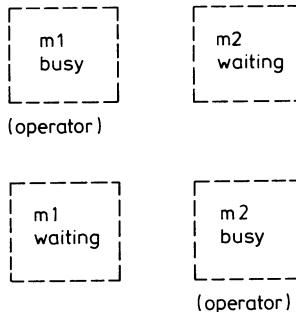


Fig. 2

Describe the above system through transition assertions (state transition diagram) and write the basic logic for its simulation.

Logic program:

```
/* model base */
trans(s([m1,busy],(m2,wait))),
s([(m1,wait),(m2,busy)]), X):- eran(X,5).
```

```

trans(s([(m1,wait),(m2,busy)]),
      s([(m1,busy),(m2,wait)]), X):- eran(X,3).

/* simulation logic */

go(L):- sim(s([(m1,busy),(m2,wait)]),100,0,[],L).

sim(X,Q,T,L,L):- T>=Q,!.
sim(X,Q,T0,L,LL):- trans(X,Y,DT),
    update(Y,DT,L,W),
    T1 is T0+DT,
    sim(Y,Q,T1,W,LL).

update(Y,DT,[ (Y,T0)|L],[ (Y,T1)|L]):- T1 is T0+DT,!.
update(Y,DT,[],[(Y,DT)]):- !.
update(Y,DT,[Z|L],[Z|W]):- update(Y,DT,L,W).

```

Execution:

```

?- go(L).
L=[(s([(m1,wait),(m2,busy)]),66),(s([(m1,busy),(m2,wait)]),36)]

?- go(L).
L=[(s([(m1,wait),(m2,busy)]),67),(s([(m1,busy),(m2,wait)]),37)]

?- go(L).
L=[(s([(m1,wait),(m2,busy)]),64),(s([(m1,busy),(m2,wait)]),39)]

```

Comments:

This is an example of simulation modeling, adopting state transition diagrams as the representation formalism. States receive here individual names, and are not identified only by elementary components, as in the monkey-bananas problem (see Chapter 13). In a simulation there is a transition back from the goal state to the start state, so that the processing of the model may cycle through repeated iterations. Observe that planning (see Chapter 13) and simulation models share the characteristic that they involve reasoning about hypothetical sequences of activities. But, while planning models seem more appropriate where choice is the dominant characteristic, simulation seems more appropriate when stochastic concurrency dominates.

Note that ‘eran(X,M)’ is a built-in predicate returning exponential variables, X, having a mean of M.

Problem 166 [Lee 1985 b]

Verbal statement:

Consider again the case of the machine shop. Assume there are now three machines, m1, m2 and m3. There are two phases to the processing: the first phase

is done by machine m1; the second phase is done by either m2 or m3, depending on which is available first. Assume that processing times are exponentially distributed with means of 10, 30 and 15, respectively.

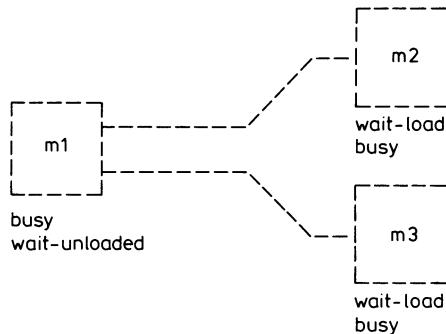


Fig.3

Adopt a Petri net diagram as the modeling paradigm:

- 1) Write a simple simulation program.
- 2) Refine the previous program by considering that there are multiple types of jobs, e.g. 'standard' and 'custom', and the choice between m2 and m3 is based on job type: m2 for standard jobs, and m3 for custom jobs.
- 3) Re-write the simulation logic that includes a boundary condition and collects data on the time spent on each activity.

Logic program 1:

```

/* model base */
start([(0,0,act(m1,b)),(_,0,act(m2,w1)),(_,0,act(m3,w1))]).

/* node a */
xtrans(T,[(_,_ ,act(m2,b))],[(_ ,T,act(m2,w1))]).

/* node b */
atrans(T,[(_,_ ,act(m1,wu)),(_,_ ,act(m2,w1))],
      [(T1,T,act(m1,b)),(T2,T,act(m2,b))]):-
      eran(X1,10),
      eran(X2,30),
      T1 is T-X1,
      T2 is T-X2.

/* node c */
xtrans(T,[(_,_ ,act(m1,b))],[(_ ,T,act(m1,wu))]).

/* node d */

```

```

atrans(T, [(_,_ ,act(m1,wu)), (_,_ ,act(m3,w1))],
       [(T1,T,act(m1,b)),(T3,T,act(m3,b))]):-
    eran(X1,10),
    eran(X3,15),
    T1 is T+X1,
    T3 is T+X3.

/* node e */

xtrans(T, [(_,_ ,act(m3,b))], [(_,_ ,act(m3,w1))]). 

/* Simulation */

go:- start(L), bump(0,L).

bump(T0,L0):-
    write('.'), 
    transit(T0,L0,T1,L1),
    bump(T1,L1).

transit(T,L0,T,L1):-
    atrans(T,WW,AA),
    remove(WW,L0,MM),!,
    append(AA,MM,L1).

transit(T,L0,T1,L1):-
    qsort(L0,[(T1,T0,A)|LL]),
    xtrans(T,[ (T1,T0,A)],AA),
    append(LL,AA,L1).

qsort([],[]).
qsort([H|T],S):-
    split(H,T,A,B),
    qsort(A,A1),
    qsort(B,B1),
    append(A1,[H|B1],S).

delete(X,L,LL):-
    append(L1,[X|L2],L), /* X is intermediate element of L */
    append(L1,L2,LL). /* LL is append of front, back */

remove([],M,M).
remove([X|L],M,MM):-
    delete(X,M,M1),
    remove(L,M1,MM).
append([],L,L).
append([X|L1],L2,[X|L3]):-
    append(L1,L2,L3).

```

Execution:

```

?- spy bump.
Spy-point placed on bump/2.

?- go.

** 1 Call:bump(0,
   [(0,0,act(m1,b)),(_44,0,act(m2,w1)),(_45,0,act(m3,w1))])?

```

```

** 2 Call:bump(0,
   [_44,0,act(m2,w1)),(_45,0,act(m3,w1)),(_251,0,act(m1,wu))])?

** 3 Call:bump(0,
   [(22,0,act(m1,b)),(50,0,act(m2,b)),(_45,0,act(m3,w1))])?

** 4 Call:bump(22,
   [(50,0,act(m2,b)),(_45,0,act(m3,w1)),(_789,0,act(m1,wu))])?

** 5 Call:bump(22,
   [(31,22,act(m1,b)),(31,22,act(m3,b)),(50,0,act(m2,b))])?

** 6 Call:bump(31,
   [(31,22,act(m3,b)),(50,0,act(m2,b)),(_805,22,act(m1,wu))])?

** 7 Call:bump(31,
   [(50,0,act(m2,b)),(_805,22,act(m1,wu)),(_1044,31,
   act(m3,w1))])?

** 8 Call:bump(31,
   [(41,31,act(m1,b)),(60,31,act(m3,b)),(50,0,act(m2,b))])?

** 9 Call:bump(41,
   [(50,0,act(m2,b)),(60,31,act(m3,b)),(_1687,31,act(m1,Wu))])?

** 10 Call:bump(50,
   [(60,31,act(m3,b)),(_1687,31,act(m1,wu)),(_1926,41,
   act(m2,w1))])?

** 11 Call:bump(50,
   [(82,50,act(m1,b)),(72,50,act(m2,b)),(60,31,act(m3,b))])? a
[Execution aborted]

```

In this example, if both m2 and m3 are waiting, the choice defaults to m2, by the ordering of the rules.

Logic program 2:

```

/* model base */

start([(0,0,act(m1,b(standard))),(_,0,act(m2,w1)),
       (_,0,act(m3,w1))]). 

/* node a */

xtrans(T,[(_,-,act(m2,b))],[(-,T,act(m2,w1))]). 

/* node b */

atrans(T,[(_,-,act(m1,wu(standard))),(_,-,act(m2,w1))],
      [(T1,T,act(m1,b(X))), (T2,T,act(m2,b))]):-
      eran(X1,10),
      eran(X2,30),
      T1 is T+X1,
      T2 is T+X2,
      draw(X,[standard,
              custom]). 

```

```

/* node c */
xtrans(T, [(_,_ ,act(m1,b(X))), [(_ ,T,act(m1,wu(X)))]]).

/* node d */
atrans(T, [(_,_ ,act(m1,wu(custom))), (_,_ ,act(m3,w1))],
        [(T1,T,act(m1,b(X))), (T3,T,act(m3,b))]):-
    eran(X1,10),
    eran(X3,15),
    T1 is T+X1,
    T3 is T+X3,
    draw(X,[standard,
             custom]).

/* node e */
xtrans(T, [(_,_ ,act(m3,b))], [(_ ,T,act(m3,w1))]).

/*utilities */
eran(A,B):- C is random, D is -ln(C), A is B*D.
draw A,B):- length (B,C), D is random,
            E is D*C, F is integer(E),
            G is F+1, element(A,G,B).
element(A,B,C):- nonvar(B), append(D,[A:E],C),
                 F is B-1, length(D,F).
element(A,B,C):- var(B), append(D,[A:E],C), length(D,F), B is F+1.

```

Execution:

```

?- spy bump.
Spy-point placed on bump/2.

?- go.

** 1 Call: bump(0,
   [(0,0,act(m1,b(standard))),(_44,0,act(m2,w1)),(_45,0,
   act(m3,w1))])?

** 2 Call: bump(0,
   [(_44,0,act(m2,w1)),(_45,0,act(m3,w1)),(_252,0,
   act(m1,wu(standard))])?

** 3 Call: bump(0,
   [(22,0,act(m1,b(standard))), (50,0,act(m2,b)),(_45,0,
   act(m3,w1))])?

** 4 Call: bump(22,
   [(50,0,act(m2,b)),(_45,0,act(m3,w1)),(_861,0,
   act(m1,wu(standard))])?

** 5 Call: bump(50,
   [(_45,0,act(m3,w1)),(_861,0,act(m1,wu(standard))),(_1100,22,
   act(m2,w1))])?

```

```

** 6 Call: bump(50,
   [(51,50,act(m1,b(custom))), (76,50,act(m2,b)), (_33,0,
   act(m3,w1))])?

** 7 Call: bump(51,
   [(76,50,act(m2,b)), (_33,0,act(m3,w1)), (_1287,50,
   act(m1,wu(custom))))]?)?

** 8 Call: bump(51,
   [(63,51,act(m1,b(custom))), (76,51,act(m3,b)), (76,50,
   act(m2,b))])?

** 9 Call: bump(63,
   [(76,50,act(m2,b)), (76,51,act(m3,b)), (_2133,51,
   act(m1,wu(custom))))]?)?

** 10 Call: bump(76,
   [(76,51,act(m3,b)), (_2133,51,act(m1,wu(custom))), (_2372,63,
   act(m2,w1))])?

** 11 Call: bump(76,
   [(_2133,51,act(m1,wu(custom))), (_2372,63,act(m2,w1)),
   (_2611,76,act(m3,w1))])?

** 12 Call: bump(76,
   [(81,76,act(m1,b(custom))), (80,76,act(m3,b)), (_2372,63,
   act(m2,w1))])?

** 13 Call: bump(80,
   [(81,76,act(m1,b(custom))), (_2372,63,act(m2,w1)), (_3307,76,
   act(m3,w1))])?

[execution aborted]

```

Logic program 3:

```

go(quit,History):- start(L),bump(Quit,0,L,[],History).

bump(Q,T,L,H,H):- T>=Q,!.
bump(Q,T0,L0,H0,HH):- 
   transit(T0,L0,T1,L1),!,
   DT is T1-T0,
   update(DT,L0,H0,H1),
   bump(Q,T1,L1,H1,HH).

update(0,_,H,H):- !.
update(_,[],H,H):- !.
update(DT,[(_,_,_)|L],H,[((TT,A)|HH)]):-
   delete((T,A),H,H1),!,
   TT is T+DT,
   update(DT,L,H1,HH).

update(DT,[(_,_,_)|L],H,[((DT,A)|HH)]):-
   update(DT,L,H,HH).

```

```

transit(T,L0,T,L1):-
    atrans(T,WW,AA),
    remove(WW,L0,MM),!,
    append(AA,MM,L1).
transit(T,L0,T1,L1):-
    qsort(L0,[(T1,T0,A)|LL]),
    xtrans(T,[(T1,T0,A)],AA),
    append(LL,AA,L1).

```

Execution:

```

?- go(1000,History).

History = [(748,act(m2,b)),
(641,act(m3,w1)),
(411,act(m1,wu(standard))),
(170,act(m1,b(standard))),
(366,act(m3,b)),
(116,act(m1,wu(custom))),
(310,act(m1,b(custom))),
(259,act(m2,w1))]

?- go(1000,History).

History = [(810,act(m2,b)),
(676,act(m3,w1)),
(406,act(m1,wu(standard))),
(225,act(m1,b(standard))),
(219,act(m2,w1)),
(353,act(m3,b)),
(243,act(m1,b(custom))),
(155,act(m1,wu(custom)))]

?- go(1000,History).

History = [(757,act(m2,b)),
(583,act(m3,w1)),
(439,act(m1,wu(standard))),
(190,act(m1,b(standard))),
(281,act(m2,w1)),
(455,act(m3,b)),
(213,act(m1,wu(custom))),
(196,act(m1,b(custom)))]

```

Comments:

A PERT diagram conveys concurrency of activities, but does not indicate choice between alternative courses of action. A state transition diagram indicates choice, but not concurrency. A Petri net is a combination of these two. Two types of nodes are included: choice nodes (drawn as circles), analogous to the nodes in state-trans-

sition graphs; and transition nodes (drawn as boxes), analogous to nodes in Pert graphs.

A Petri net diagram of the processing is as follows:

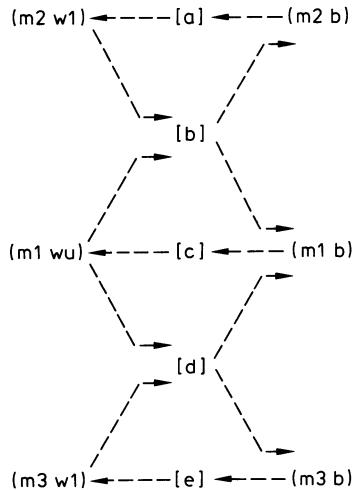


Fig.4

The flow of processing in a Petri net is described by means of ‘tokens’ that flow through the network. Tokens may reside at choice nodes, also called ‘places’. A transition node ‘fires’ by removing a token from each of its input places and depositing a (new) token on each of its output places. The state of the system is given by the location of tokens at each of the various places. (We assume that a place can have at most one token; this assumption is sometimes relaxed.)

Transition nodes in the Petri net become ‘trans’ assertions in the logic program. The predicate ‘atrans’ is used where the pre-conditions are waiting activities. The predicate ‘xtrans’ is used where the pre-conditions are busy activities.

For the program 2 the transitions differ only in the description of machine m1’s activities, which now record the type of job. The predicate ‘draw’ returns a random element from the list.

Chapter 19 Building Up with Prolog

Why use Prolog for writing interpreters, when C is so much better? The experience built up upon the DECsystem-10 Prolog interpreter and compiler establishes evidence that both programming languages are not comparable, in particular in what concerns machine efficiency. However, the following exercises put the case for Prolog as a tool for helping designers, i.e. as an aid in the learning process.

Problem 167 [L. Pereira et al. 1978]

Verbal statement:

Write a Prolog interpreter in Prolog, and illustrate the use of a variable goal.

In this mini-interpreter, goals and clauses may be represented as ordinary Prolog data structures (i.e. terms). Terms representing clauses may be specified using the unary predicate ‘clause’, e.g.

```
clause( (grandparent(X, Z):- parent(X, Y), parent(Y, Z)) ).
```

A unit clause may be represented by a term such as: -

```
( parent(john,mary):- true)
```

Logic program:

```
execute(true):- !.  
execute((P,Q)):- !, execute(P), execute(Q).  
execute(P):- clause((P:-Q)), execute(Q).  
execute(P):- P.
```

Problem 168 [L. Pereira 1978]

Verbal statement:

Extend the mini-interpreter considered in the previous problem, in order to deal with the “cut” symbol.

Logic program:

```
execute(true,_).
execute(!,_).
execute(!,cut).
execute((P,Q),V):- !, execute(P,C),
               (C==cut,V=cut;
                execute(Q,V)).
execute(P,V):- clause((P:-Q)),
              execute(Q,V),
              (V==cut,! ,fail;true).
execute(P,_):- P.
```

Problem 169 [L. Pereira et al. 1978]

Verbal statement:

Illustrate the use of the meta-predicates ‘var’ and ‘=..’. The procedure call ‘variables(*Term*,*L*,*I*)’ instantiates variable *L* to a list of all the variable occurrences in the term *Term*.

e.g. variables(d(U*V,X,DU*V+U*Dv),[U,V,X,DU,V,U,DV],[]).

Logic program:

```
variables(X,[X|L],L):- var(X),!.
variables(T,L0,L):- T =.. [F|A],variables1(A,L0,L).
variables1([T|A],L0,L):- variables(T,L0,L1),variables1(A,L1,L).
variables1([],L,L).
```

Problem 170 [L. Pereira et al. 1978]

Verbal statement:

Define conditional in Prolog.

Logic program:

```
:-op(1050,xfx,'->').
(P -> Q;R):- P,! ,Q.
(P -> Q;R):- ! ,R.
(P;Q):- P.
(P;Q):- Q.
(P,Q):- P,Q.
true.
```

```

:-repeat, read(X), (X=end -> true;
                      X -> write((Yes;)), fail.
                                         write((no;)), fail).

```

Execution:

```

24 > 23.
yes
23 > 23.
no
23 >= 23.
yes
end.

```

Problem 171 [Monteiro 1982]

Verbal statement:

Write a small interpreter (DTLOG) for distributed logic (DL), an extension of Horn clause logic (HCL) characterized by the following concepts: computing agents, configurations, transition rules, transitions, and an extension of the resolution principle to deal with the above notions.

COMPUTING AGENTS: These are ordinary HCL terms.

CONFIGURATIONS: Computing agents are organized into configurations, which are special terms. They are built up of computing agents and the operators ' \wedge ' (nullary) and '+', '.' (binary). These operators are called respectively the "null configuration" and the "concurrent" and "sequential" compositions; '+' and '.' are associative and have ' \wedge ' as neutral element. In writing DL programs for DTLOG we use 'skip', '< >' and '&' instead of ' \wedge ', '+' and '.' respectively.

TRANSITION RULES: Expressions of the form

$a_1, \dots, a_n \rightarrow c_1, \dots, c_n$

where $n > 0$, the a 's are agents and the c 's are configurations. In DTLOG we restrict n to $n = 1$ or $n = 2$, and call the transition rules unary or binary respectively. They are written

$a \rightarrow c$

or

$(a_1, a_2) \rightarrow (c_1, c_2)$

where a longer arrow is employed in binary transition rules merely to increase the efficiency of the interpreter. The logical status of a transition rule is that of an atomic formula. The user is expected to use them normally (but not necessarily) in the heads of clauses.

TRANSITIONS: Expressions of the form

$c \Rightarrow d$

where c, d are configurations. We allow

c^\wedge

as an abbreviation for $c = > \text{skip}$. In writing $c = > d$, DTLOG assumes that d contains no occurrence of ‘skip’ if $d = / = \text{skip}$. The logical status of a transition is that of an atomic formula. The user is expected to use them normally (but not necessarily) in the bodies of clauses. The operational interpretation of $c = > d$ is “execute c until it has the form d , applying suitable transition rules to c ”.

RESOLUTION PRINCIPLE: Distributed logic programs are, like HCL programs, sets of Horn clauses. To deal with the extra notions, the resolution principle has been suitably extended. We are only interested in linear top-down refutation strategies. The resolution of a negative clause and a definite clause to yield a new negative clause is as in Prolog, with the exception of goals of the form $c = > d$, as explained below.

Apply the interpreter to the following two examples:

Example 1:

(1) Our first example is a simplified version of the five philosophers problem. We only consider two philosophers (and two forks), since our main purpose is to specify a system with a deadlock. Each fork (0 or 1) is either up or down. Each philosopher (0 or 1) repeats for a specified number of life-cycles the actions of picking up a fork, picking up the other fork, putting down the first fork and putting down the second fork. The program follows.

Example 2:

(2) We are given two nonempty finite disjoint sets of integers, S_0 and T_0 , and are required to produce two sets S and T such that:

- (i) $S \cup T = S_0 \cup T_0$;
- (ii) $\#S = \#S_0$, $\#T = \#T_0$ ('#' stands for cardinal)
- (iii) every element of S is smaller than any element of T .

Logic program:

```
/* DTLOG : an interpreter for distributed logic */

?- op(240, xf, '^').    /* X^ is equivalent to X=>skip      */
?- op(230, xfx, '->').  /* unary transition rule predicate */
?- op(230, xfx, '-->'). /* binary transition rule predicate */
?- op(230, xfx, '>').   /* transition predicate           */
?- op(215, xfy, '<>').  /* concurrent composition        */
?- op(200, xfy, '&').    /* sequential composition         */
```

```

X^ :- X=>skip.
X=>X.
X=>Z :- step(X,Y), Y=>Z.

step(X,Y) :- top(X,T-[],C), exec(T), simplify(C,Y), compare(X,Y).
top(X1<>X2, T-T2, C1<>C2) :- !, top(X1, T-T1, C1), top(X2, T1-T2, C2).
top(X, [X->C|T]-T, C).

exec([]).
exec([X->C|T]) :- X->c, exec(T).
exec([X->C|T]) :- delete(Y->D, T, T1),
                  ((X,Y)-->(C,D); (Y,X)-->(D,C)),
                  exec(T1).
exec([X->X|T]) :- exec(T).

simplify(skip<>X, Y) :- !, simplify(X, Y).
simplify(X<>skip, Y) :- !, simplify(X, Y).
simplify(X<>V, Z) :- simplify(X, Y), simplify(V, W),
                     (W=skip, Z=Y; Z=(Y<>W)), !.
simplify(skip&X, X) :- !.
simplify(X&W, Z) :- simplify(X, Y), (Y=skip, Z=W; Z=(Y&W)), !.
simplify(X, X).

compare(X, Y) :- X\==Y.          /* avoids looping */

delete(X, [X|L], L).
delete(X, [Y|L], [Y|M]) :- delete(X, L, M).

philosophers(Lifespan) :-
    phil(0, Lifespan) <> phil(1, Lifespan) <> forkdown(0) <> forkdown(1)
                           => forkdown(0) <> forkdown(1).

phil(I, 0) -> skip.
phil(I, N) -> getfork(I) & getfork(J) & putfork(I) & putfork(J)
               & phil(I, M)
               :- N>0, M is N-1, J is (I+1) mod 2.

(getfork(I), forkdown(I)) --> (skip, forkup(I)).
(putfork(I), forkup(I)) --> (skip, forkdown(I)).

partition(S0, T0, S, T) :- procS(S0) <> procT(T0) => endS(S) <> endT(T).

procS(S0) -> exchgS(X, Y) & contS(X, Y, S) :- max(S0, X, S).
procT(T0) -> exchgT(X, Y) & contT(X, Y, T) :- min(T0, Y, T).

exchgS(X, Y), exchgT(X, Y)) --> (skip, skip).

contS(X, Y, S) -> endS([X|S]) :- X<Y.
contS(X, Y, S) -> procS([Y|S]) :- Y<X.

contT(X, Y, T) -> endT([Y|T]) :- X<Y.
contT(X, Y, T) -> procT([X|T]) :- Y<X.

```

```

max([W|S],X,[W|Q]) :- max(S,X,Q), W<X, !.
max([X|S],X,S).

min([V|T],Y,[V|R]) :- min(T,Y,R), Y<V, !.
min([Y|T],Y,T).

```

Comments:

For example 1:

(1) When presented with a user program, DTLOG expects to solve (execute) goals of the form $X => Z$, where X and Z are configurations. (A goal of the form X^{\wedge} is transformed into $X => \text{skip}$; any other goal is immediately transferred to the Prolog interpreter.) If X and Z are unifiable then the task terminates successfully. Otherwise it performs a transition step on X to obtain Y and tries to solve $Y => Z$. The transition step is accomplished by the predicate $\text{step}(X,Y)$.

(2) A transition step is subdivided in four stages:

(2.1) In the first stage, configuration X is split up into its “top” T and the “context” C in which T occurs in X (see example). The context C, however, has distinct variables in the place of holes. As a first approach, T is the list of all agents which occur in the top of X. As a second approach, each element in the list T is in fact a pair, written $A -> V$, formed by an agent A occurring in the top of X and a variable V which occupies in C the same position as occupied by A in X. This stage is accomplished by the predicate $\text{top}(X,T-[],C)$. (Note the use of difference lists, to avoid the explicit definition of the concatenation of two lists.) As an example, if we start with

$X = (a <> b) \& c <> d \& e$

then we obtain

$T = [a -> U, b -> V, d -> W] \quad C = (U <> V) \& c <> W \& e$

where U, V, W are distinct variables.

(2.2) In the second stage, the list T is picked up from the difference list $T-[]$ produced by top and executed, as specified by predicate $\text{exec}(T)$. To execute T means to execute every member of T. To execute the pair $A -> V$ means one of the following three things:

- (i) to execute the predicate $A -> V$;
- (ii) to find some other member $B -> W$ in T and to execute one of the predicates $(A,B) -> (V,W)$ or $(B,A) -> (W,V)$;
- (iii) to unify A and V (this corresponds to postponing the execution of A).

The strategy followed by DTLOG is to execute the members of T from left to right and try the execution alternatives (i-iii) in this order. Notice that after the execution of T the variables occurring in the top of C are instantiated to configurations.

Continuing with the previous example, suppose the only clauses involving the agents a, b, d were

$b -> c1. \quad (b,d) -> (c2,c3). \quad (a,b) -> (\text{skip},\text{skip}).$

The outlined execution strategy would give us the following instance of C:

$$C = (\text{skip} <> \text{skip}) \& c <> d \& e$$

- (2.3) The simplification stage eliminates from C all occurrences of 'skip' (unless C is itself 'skip'), the result being Y (remember skip is the neutral element for $<>$ and $\&$). In the previous example we would obtain

$$Y = c <> d \& e$$

- (2.4) The final stage checks whether the system specified by the DL program is in a deadlock situation or not. Deadlock occurs iff the only possibility for executing the members of T is (2.2-iii) (assuming there are no agents for which $a -> a$ or $(a,b) --> (a,b)$). This is true iff X and Y are identical. Hence the final stage in the execution of step(X,Y) consists in verifying that X and Y are not identical.
- (3) The system automatically recovers from a deadlock situation by backtracking. If for some reason this is not desired, the cut symbol '!' should be inserted after the call 'exec(T)'.
- (4) As a further facility, ordinary predicates can be turned into agents and inserted in configurations by writing a predicate P in the form agt(P). The following clause should then be added to the interpreter below:

```
exec([agt(P)->skip|T]) :- !, P, exec(T).
```

For example 2:

We assume there are an S-process and a T-process responsible for the S-sets and T-sets respectively. The S- and T-processes start by finding respectively the maximum value X and the minimum value Y of their input gets. These values are then exchanged and a check is made whether $X < Y$. If so, both processes stop and the exchange is not accepted. Otherwise the exchange is accepted and the processes repeat the actions already described.

Problem 172 [L. Pereira; Porto 1982]

Verbal statement:

An evaluator for pure Lisp in pure Prolog is required.

Logic program:

```
?- op(10,fx,'').  
  
lisp:- read(E), eval(E,[],R), write(R), nl, nl, lisp.  
  
eval(A,U,R):- atomic(A),  
           ( (integer(A); A=[]; A=true ), R=a;  
             assoc(A,U,[_,R]);  
             error ).
```

```

eval([quote,X],_,X).
eval(X,_,X).

eval([cond,[T,B]|L],U,R) :- eval(T,U,ET),
    (ET=true, eval(B,U,R);
     eval([cond|L],U,R)).

eval([cond],_,[]).
eval([list,[X|L]],U,[EX|EL]) :- eval(X,U,EX), eval([list,L],U,EL).
eval([list],_,[]).

eval([car,X],U,Y) :- eval(X,U,EX), (EX=[Y|_]; error).
eval([cdr,X],U,Y) :- eval(X,U,EX), (EX=[_|Y]; error).
eval([cons,X,Y],U,[EX|EY]) :- eval(X,U,EX), eval(Y,U,EY).
eval([atom,X],U,R) :- eval(X,U,EX), (atomic(EX), R=true; R=[]).
eval([equal,X,Y],U,R) :- X=Y;
    eval(X,U,EX),
    eval(Y,U,EY),
    (EX=EY, R=true; R=[]).

eval([F|L],U,R) :- assoc(F,U,P),
    (P=[_,EF], eval([EF|L],U,R); error).

eval([[lambda,V,E]|A],U,R) :- evalist(A,U,EA),
    pair(V,EA,P),
    append(P,U,W),
    eval(E,W,R).

eval([not,X],U,R) :- eval(X,U,EX), (EX=true, R=[]; R=true).
eval([and,X,Y],U,R) :- eval(X,U,EX), (EX=[], R=[]; eval(Y,U,R)).
eval([or,X,Y],U,R) :- eval(X,U,EX), (EX=[], eval(Y,U,R); R=EX).
eval([defun,N,A,E],_,N) :- assert(definition(N,[lambda,A,E])).
eval([eval,X],U,R) :- eval(X,U,EX), eval(EX,U,R).

/* extra notation */

eval([null,X],U,R) :- eval([equal,X,[]],U,R).
eval([if,C,A,B],U,R) :- eval([cond,[C,A],[true,B]],U,R).

/* association list */

assoc(X,_,[_,_]) :- definition(X,R).
assoc(X,[[Y,VY]|U],R) :- X=Y, R=[Y,VY]; assoc(X,U,R).

/* examples of defined functions */

definition(ff,[lambda,[X],[if,
    [atom,X],
    X,
    [ff,[car,X]]]]).
definition(alt,[lambda,[u],[if,
    [null,u],
    [],
    [if,
        [null,[cdr,u]],
        u,
        [cons,[car,u],[alt,[cdr,[cdr,u]]]]]]]).

```

```

/* utilities */

error:- write(error), tab(2), abort.

evalist([H|T],U,[EH|ET]):- eval(H,U,EH), evalist(T,U,ET).
evalist([],[],[]).

pair([X|Y],[U|V],[[X,U]|P]):- pair(Y,V,P).
pair([],[],[]).

append([H|T],L,[H|R]):- append(T,L,R).
append([],L,L).

```

Execution:

```

?- eval([ff,X],[[X,[[1,2],3]]],R).
R=1

?- lisp.
[alt,[1,2,3,4,5]].
[1,3,5]

```

Comments:

The relation ‘eval(E,U,R)’ takes an S-expression and evaluates it to R, in the context of association list U comprising two element lists pairing atoms to their associated values.

The evaluator features the Prolog system predicates indicated below:

X=Y	X unifies with Y
atom(A)	A is an atom
integer(I)	I is an integer
atomic(A)	A is an atom or integer

Prolog syntax is used for lists. As usual in Lisp ‘false’ is represented by the empty list, in this case ‘[]’. The predicate ‘equal’ is made primitive rather than the implementation oriented concept ‘eq’. Numeric functions and predicates are left out. Space may be recovered by garbage collection each cycle:

```

lisp:- repeat,solve((read(E),eval(E,[],R),
                      write(R),nl,nl)),fail.
solve(G):- G,!.

```

Where ‘repeat’ is a system predicate that always solves again. The predicate ‘assert’ is used as an optional convenience for storing functions interactively.

Problem 173 [L. Pereira 1982]**Verbal statement:**

Write an interpreter that realizes the demand driven computation process (Hansson et al. 1982).

Operationally an object language sentence specifies an algorithm by a network of communicating processes through unbound buffers. Networks are constructed by composition and recursion. In a sentence whose execution is controlled by a demand driven rule one demanding process will start the execution of the body of the sentence. If the network is cycle free any non-producer consumer can be the demanding process. If the network is cyclic, with one cycle or an overlapping number of cycles then any process in such cycles can be the demanding process. If the network is composed of disjoint cycles with one cycle acting as a producer to one or more cycles, then there must be at least one cycle which acts as non-producer consumer. Any process in such a cycle can be the demanding process. The only restriction is that there can be only one designated producer for any variable. This is reasonable since a problem would arise in deciding which producer should grant a demand. A situation may arise where many consumers may demand a partial result from a single producer. This may only occur when the producer of a result has local consumers. In this situation the consumers will be resumed in an innermost to outermost order before the procedure is reactivated.

Logic program:

```

?- op(230,xfx,=:).      /* functional relations */
?- op(240,fx , @).      /* access to program clauses */
?- op(240,fx , #).      /* evaluation */
?- op(254,xfx,<-).      /* functional relations' conditions */

/* User interface */

stream(R=:X):- s(R,X-X).

s(R,X-Z):- # R=:A, !, ((A=[];A=[_|T],list(T)),Z=A      ;
                           A=[V|T],Z=[V|Y],show(T,X),s(T,X-Y)).

show(T,X):- write(X),get0(C),(C=32,write(T),skip(10),nl;C=10),nl.

up_to(N,R=: [V|Y]):- N>0,#R=: [V|S],!,M is N-1,up_to(M,S=:Y).
up_to(_,_=:[]).

/* Interpreter */

#R=:S:- (list(R),R=S;@R=:A,#A=:S).
#(A,B):- #A,#B.
#G:- G.

list([]).
list([_|_]).

/* Access to non-unit and unit functional predicate clauses,
regular Prolog clauses, and system predicates */

@G:- (G<-C),#C.          /* non-unit clauses */
@G:- G.                   /* unit clauses, Prolog, and system */

```

```

@ conc(A,X)=: conc(EA,X)           :- # A=:EA.
@ select(N,A)=: select(N,EA)        :- # A=:EA.
@ sift(A)=: sift(EA)               :- # A=:EA.
@ filter(X,A)=: filter(X,EA)       :- # A=:EA.
@ merge(A,B)=: merge(EA,EB)         :- # A=:EA, #B=:EB.
@ mul(A,B)=: mul(A,EB)              :- # B=:EB.

/* Programs */

/* concatenation */

conc([],X)=: X.
conc([X|Y],U)=: [X|conc(Y,U)].

/* bounded buffer */

bounded_buffer(WS,RS)=: AS <- bmerge(WS,RS,0,S1),
                        buffer(S1,U-U)=:AS.

bmerge([write(X)|WS],RS,I,[write(X)|AS]):- I<5,K is I+1,
                                             bmerge(WS,RS,K,AS).
bmerge(WS,[read|RS],I,[read|AS]):- I>0,K is I-1,
                                         bmerge(WS,RS,K,AS).

bmerge(_,[],_,[]).

buffer([write(X)|S],V-[X|W])=: buffer(S,V-W).
buffer([read|S],[X|V]-W)=: [X|buffer(S,V-W)].
buffer([],_--[])=: [].

/* Infinite list of integers */

intfrom2=: inc(2).

inc(X)=: [X|inc(K)] <- K is X+1.

n_integer(N)=: Y <- intfrom2=:X,select(N,X)=:Y.

select(0,_)=: [].
select(N,[X|Y])=: [X|select(K,Y)] <- N>0, K is N-1.

/* Primes */

sift([X|Y])=: [X|sift(filter(X,Y))]. 

filter(X,[Y|Z])=: [Y|filter(X,Z)] <- YmodX=\=0.
filter(X,[Y|Z])=: [Y|filter(X,Z)] <- YmodX=:0.

/* Quicksort */

qs([])=: [].
qs([X|Y])=: conc(qs(Y1),[X|qs(Y2)]) <- part(X,Y,Y1,Y2).

part(X,[H|T],[H|S],R):- H=<X,part(X,T,S,R).
part(X,[H|T],S,[H|R]):- H>=X,part(X,T,S,R).
part(_,[],[],[]).

/* Cyclic network of agents */

```

```

p=: Y <- merge(mul(2,[1|Y]),merge(mul(3,[1|Y]),mul(5,[1|Y])))=:Y.

merge([X|Y],[U|V])=: [X|merge(Y,[U|V])] <- X<U.
merge([X|Y],[U|V])=: [U|merge([X|Y],V)] <- X>U.
merge([X|Y],[U|V])=: [X|merge(Y,V)] <- X=U.

mul(X,[Y|Z])=: [W|mul(X,Z)] <- W is X*Y.

```

Comments:

Predicate “stream” accepts a functional predicate goal ‘R’ and delivers a stream ‘X’ of the result, where difference lists are used to represent streams. After each value in the stream is produced it pauses, and displays the stream. If a <CR> is given it continues, if a <space> is given it shows the calls waiting to be demand driven and continues.

Example call: `stream(p=:X).`

Predicate “up_to” produces up to N values of a stream for a given call.

Example call: `up_to(3,conc([1,2],[3,4]))=:X.`

Predicate “#” evaluates any predicate call. If the predicate is functionally defined, it evaluates it recursively until a list is produced, where the head of the list, if any, contains the first result of evaluating the call, and the tail a call to a functional predicate for producing the next result. The call [] evaluates to the empty list. To do so, it uses predicate “@”, which picks up a clause for a functionally defined predicate and evaluates its body if there is one. However, if any argument is demand driven and is not yet evaluated, no clause can be picked up and “@” will evaluate the demand driven arguments, and return to “#” the call with its arguments evaluated.

Problem 174 [Corlett; Todd 1983]

Verbal statement:

Write a simple interpreter to enable selective coroutining of goals within a Prolog program.

(Acknowledgement: the following example is from R.A.Corlett and S.J.Todd of the Marconi Research Centre and reprinted with kind permission of GEC p.l.c.)

Logic program:

```

suspend.

system//(X,Y).

?- op(251,xfy,//).

true//P:- !, call(P).
(true,P)//Q:- !, (P//Q).

```

```
(suspend,P)//Q:- !,(Q//P).
(P;Q)//R:- !,(P//R;Q//R).
((P;Q),R)//S:- !,(P,R//S;Q,R//S).
((P,Q),R)//S:- !,(P,Q,R//S).
((P//Q)//R):- !,(P//Q//R).
(P,Q)//R:- !,(system(P) -> call(P),(Q//R);
clause(P,S),(S,Q//R)).
P//Q:- !,(system(P) -> call((P,Q));
clause(P,R),(R//Q)).
```

Execution:

An example of a problem for which the explicit insertion of suspension points is appropriate is the classic Eight Queens problem for which a simple sequential solution is given below:

```
solution(Perm):-
    permutation([1,2,3,4,5,6,7,8],Perm),safe(Perm).

permutation(L,[Q|M]):-
    remove(Q,L,L1),permutation(L1,M).
permutation([],[]).

remove(X,[X|L],L).
remove(X,[Y|L],[Y|M]:- remove(X,L,M).

safe([Queen|List]):-
    nodiagonal(Queen,List,1),safe(list).
safe([]).

nodiagonal(Q1,[Q2|List],N):-
    noattack(Q1,Q2,N),N1 is N+1,
    nodiagonal(Q1,List,N1).
nodiagonal(Q1,[],N).

noattack(Q1,Q2,N):-
    Q1>Q2,Diff is Q1-Q2,Diff\=N.

noattack(Q1,Q2,N):-
    Q2>Q1,Diff is Q2-Q1,Diff\=N.
```

A coroutining solution is:

```
suspend.

?- op(251,xfy,//).

((P//Q)//R):- !,(P//Q//R).

solution(Perm):-
    permutation([1,2,3,4,5,6,7,8],Perm)//safe(Perm).

permutation(L,[Q|M])//R:- 
    remove(Q,L,L1),(R//permutation(L1,M)).
```

```

permutation([],[])//Q:- !,call(Q).

remove(X,[X|L],L).
remove(X,[Y|L],[Y|M]) :- remove(X,L,M).

safe([Queen|List])//P:-  

    P//(nodiagonal(Queen,List,1)//safe(List)).  

    safe([]).

nodiagonal(Q1,[Q2|List],N)//P:-  

    noattack(Q1,Q2,N), N1 is N+1,  

    (P//(nodiagonal(Q1,List,N1))).  

nodiagonal(Q1,[],N)//P:- !,call(P).

noattack(Q1,Q2,N) :- Q1>Q2,Diff is Q1-Q2,Diff\=N.  

noattack(Q1,Q2,N) :- Q2>Q1,Diff is Q2-Q1,Diff\=N.

```

Comments:

In the program the goal ‘system(P)’ succeeds if P is a call to a built-in predicate and is used to avoid errors from accessing the body of system predicates.

The infix operator, ‘//’, has a declarative reading of ‘and’, and coroutines the processes it separates. Transfer of control out of the active process is effected by the explicit inclusion of suspension points defined by the presence of a ‘suspend’ goal. The interpreter as given makes no provision for ‘if then-else’ constructs or cuts. The interpreter might also be extended to handle multiple levels of active coroutines, but this requires a more detailed analysis of the flow of control than is in fact required.

Problem 175 [Viccarri, personal communication 1986]

Verbal statement:

Write an interpreter able to correct Prolog programs. The interpreter must perform lexical and syntactical analyses of the Prolog terms, when there are no ambiguity situations. In case of errors, a message must be supplied to the user explaining the mistakes, and two cases may occur:

- 1) the program is repaired;
- 2) the user helps in repairing it.

Logic program:

```

prolog :-  

    op(1200,fx,?-),  

    colour(32),  

    write('For exit write end.'),  

    colour(37),  

    nl,

```

```

( abolish('L' / 1)
; true
),
assert('L'(13)),
repeat,
send(end),
!.

send(A) :-
    nl,
    lexical(B),
    ( B = [token(end, name)|C],
      A = end
    ; analyse(D,E,F,B,[ ]),
      !,
      nl,
      write(D),
      write(.),
      nl,
      retract(event(G)),
      ok_to_execute(G,D),
      !,
      A = continue
    ).

ok_to_execute(fact,A) :-
    assertz(A).

ok_to_execute(goal,A) :-
    retract(goal1(B)),
    execrec(B).

ok_to_execute(A,B).

inic_flags :-
    ( error(A),
      retract(error(A)),
      assert(error(0))
    ; assert(error(0))
    ),
    ( operator(B),
      retract(operator(B)),
      assert(operator(0))
    ; assert(operator(0))
    ),
    ( ok(C),
      retract(ok(C)),
      assert(ok(1))
    ; assert(ok(1))
    ),

```

```

( count_arg(D),
  retract(count_arg(D)),
  assert(count_arg(0))
; assert(count_arg(0))
),
( event(E)
; assert(event(aaa))
).

replace(A,0,B,C,A,KK,WW) :- KK=X, WW=Y, ! .
replace(A,B,C,D,E,KK,WW) :-
  arg(B,A,F),
  name(F,G),
  verify(G,H),
  member(F,C,I),
  ( H == 1,
    argrep(A,B,I,E)
  ; true
  ),
  J is B - 1,
  replace(E,J,C,C,KK,WW).
replace(A,B,C,D,E,KK,WW) :-
  arg(B,A,F),
  name(F,G),
  verify(G,H),
  ( H == 1,
    argrep(A,B,I,E),
    col_list(C,F,E,B,D),
    J is B - 1,
    replace(E,J,D,D,E,KK,WW)
  ; J is B - 1,
    replace(A,J,C,C,E,KK,WW)
  ).

col_list(A,B,C,D,E) :-
  arg(D,C,F),
  ( var(F),
    list(A,B,F,E)
  ; G is D - 1,
    col_list(A,B,C,G,E)
  ).

list(A,B,C,D) :-
  append(A,[B,C],D).
list(A,B,C,D) :-
  ( C = E,
    empty_list(A,E,D)
  ; C = F,
    bracket_open(F,G),

```

```
retract(ok(H)),  
I is H + 1,  
assert(ok(I)),  
exprlist(A,B,G,J),  
bracket_close(B,J,D)  
).  
  
verify([A|B],C) :-  
A > 64,  
A < 91,  
C is 1.  
verify([A|B],C) :-  
A == 95,  
C is 1.  
verify([A|B],C) :-  
C is 0.  
  
execrec(A) :-  
execute(A),  
answer(A).  
execrec(A).  
  
execute(true) :-  
!.  
execute((A , B)) :-  
!,  
execute(A),  
execute(B).  
execute(A) :-  
functor(A,B,C),  
!,  
( system(B/C),  
A,  
nl  
; clause(A,E),  
execute(E)  
).  
  
answer(A) :-  
nl,  
colour(33),  
write('answer= '),  
colour(37),  
write(A),  
write(.),  
!,  
fail.  
  
analyse(A,B,C,D,[]) :-  
inic_flags,
```

```

term(A,B,C,D,D,[[]),
error(E),
E =< 0.
analyse(A,B,C,D,[[]) :-
retract(new_list(E)),
inic_flags,
disconnect,
term(A,B,C,E,E,[[]).
analyse(A,B,C,D,E) :-
write('I do not understand.'),  
nl.
lexical(A) :-
connect,  
read_in(B),  
!,  
remove_blanks(B,C),  
examine(C,A),  
!.
read_in([A|B]) :-
get0(A),
( A = 13,  
  B = []  
 ; read_in(B)
).
examine([],[]) :-
!.
examine(A,[B|C]) :-
gettken(B,A,D),
remove_blanks(D,E),
examine(E,C).
gettken(token(A,name),B,C) :-
name(A,B,C),
!.
gettken(token(A,limits),B,C) :-
limits(A,B,C),
!.
gettken(token(A,variable),B,C) :-
variable(A,B,C),
!.
gettken(token(A,anonymous),B,C) :-
underscore(A,B,C).
gettken(token(A,number),B,C) :-
number(A,B,C).
gettken(token(A,string),B,C) :-
string(A,B,C),
!.
```

```

remove_blanks([A|B],C) :-  

    true,  

    A =< 32,  

    remove_blanks(B,C).  

remove_blanks(A,A).  
  

name(A,B,C) :-  

    ( B = D,  

      words(A,D,C)  

    ; B = E,  

      ( E = [91,93|C],  

        true,  

        A = []  

      ; E = F,  

        ( F = G,  

          symbols(A,G,C)  

        ; F = H,  

          ( H = I,  

            alone(A,I,C)  

          ; H = J,  

            ( J = K,  

              enclose_quotes(A,K,C)  

            ; J = [123,125|C],  

              true,  

              A = '{ }'  

            )  

          )  

        )  

      )  

    ).  
  

enclose_quotes(A,[39|B],C) :-  

    true,  

    charact(D,B,C),  

    name(A,[39|D]).  
  

charact([A|B],[39|C],C) :-  

    true,  

    A = 39,  

    B = [],  

    !.  

charact([A|B],[A|C],D) :-  

    true,  

    charact(B,C,D).  
  

words(A,B,C) :-  

    small(D,B,E),  

    rest_words(F,E,C),  

    name(A,[D|F]).
```

```

rest_words([A|B],C,D) :-  

    other_charact(A,C,E),  

    rest_words(B,E,D).  

rest_words([],A,A).  

other_charact(A,B,C) :-  

    letter(A,B,C),  

    !.  

other_charact(A,B,C) :-  

    digits(A,B,C),  

    !.  

other_charact(A,B,C) :-  

    underscore(A,B,C).  

symbols(A,B,C) :-  

    car_simb(D,B,E),  

    rest_simb(F,E,C),  

    name(A,[D|F]).  

rest_simb([A|B],C,D) :-  

    car_simb(A,C,E),  

    rest_simb(B,E,D).  

rest_simb([],A,A).  

car_simb(A,[A|B],B) :-  

    true,  

    !.  

belong(A,[35,36,38,42,43,45,47,58,60,61,62,63,64,92,94,96,126]).  

alone(A,[B|C],C) :-  

    true,  

    belong(B,[33,37,59]),  

    name(A,[B]).  

number(A,B,C) :-  

    int(D,B,[46|E]),  

    true,  

    int(F,E,C),  

    name(G,D),  

    name(H,F),  

    length(F,I),  

    I > 0,  

    A is G + H / 10 I.  

number(A,B,C) :-  

    int(D,B,C),  

    name(A,D).  

int([A|B],C,D) :-  

    digits(A,C,E),  

    int(B,E,D).  

int([],A,A).

```

```

variable(A,B,C) :-  

  ( B = D,  

    capital(E,D,F),  

    rest_words(G,F,C),  

    name(A,[E|G])  

  ; B = H,  

    underscore(I,H,J),  

    rest_words([E|G],J,C),  

    name(A,[I,E|G])  

  ).  

variable(A,[token(A,variable)|B],B).  

string(A,[34|B],C) :-  

  true,  

  rest_string(D,B,C),  

  name(A,[34|D]).  

string(A,[token(A,string)|B],B).  

rest_string([34],[34|A],A).  

rest_string([A|B],[A|C],D) :-  

  true,  

  rest_string(B,C,D).  

small(A,[A|B],B) :-  

  true,  

  A > 96,  

  A < 123.  

capital(A,[A|B],B) :-  

  true,  

  A > 64,  

  A < 91.  

letter(A,B,C) :-  

  capital(A,B,C),  

  !.  

letter(A,B,C) :-  

  small(A,B,C).  

digits(A,[A|B],B) :-  

  true,  

  A > 47,  

  A < 58.  

underscore(95,[95|A],A).  

limits(',',[44|A],A) :-  

  true,  

  !.  

limits('(',[40|A],A) :-  

  true,  

  !.

```

```

limits(',')',[41|A],A) :-  

    true,  

    !.  

limits(.,[46|A],A) :-  

    true,  

    !.  

limits('[',[91|A],A) :-  

    true,  

    !.  

limits(']',[93|A],A) :-  

    true,  

    !.  

limits('!',[33|A],A) :-  

    true,  

    !.  

limits('{',[123|A],A) :-  

    true,  

    !.  

limits('!',[124|A],A) :-  

    true,  

    !.  

limits('}',[125|A],A).  

belong(A,[A|B]) :-  

    !.  

belong(A,[B|C]) :-  

    A @> B,  

    belong(A,C).  

term(A,B,C,D,E,F) :-  

    subterm(A,B,C,D,1200,E,F),  

    !.  

subterm(A,B,C,D,1200,E,E) :-  

    error(F),  

    F > 0,  

    !.  

subterm(A,B,C,D,1200,E,F) :-  

    term(A,B,C,D,1200,E,F),  

    !.  

subterm(A,B,C,D,999,E,F) :-  

    term(A,B,C,D,0,E,F),  

    !.  

term(A,B,C,D,1200,E,F) :-  

    op(G,H,fx,E,I),  

    retract(operator(J)),  

    assert(operator(1)),  

    subterm(K,B,C,D,1200,I,F),  

    assert(goal1(K)),

```

```

append([G],[K],L),
A =.. L,
retract(event(M)),
assert(event(goal)),
!.
term(A,B,C,D,1200,E,F) :-  

    subterm(G,B,C,D,999,E,H),
    stop(H,F),
    A = G,  

    !.
term(A,B,C,D,1200,E,F) :-  

    subterm(G,B,C,D,999,E,H),
    ( H = I,  

      op(J,K,xfx,I,L)
    ; H = M,  

      ( M = N,  

        op(J,K,yfx,N,L)
      ; M = O,  

        ( O = P,  

          op(J,K,xfy,P,L)
        ; O = Q,  

          comma(J,D,Q,L)
        )
      )
    ),
retract(operator(R)),
assert(operator(1)),
subterm(S,C,T,D,1200,L,F),
append([J],[G,S],U),
A =.. U,
retract(event(V)),
assert(event(rule)),
!.
term([],A,B,C,O,[],[]).
term(A,B,C,D,O,E,F) :-  

    functor(G,E,H),
    sig_brac_open(H,I),
    arguments(D,J,O,I,K),
    sig_brac_close(D,K,F),
    append([G],J,L),
    M =.. L,
    cont_arg(N),
    replace(M,N,O,P,Q,KK,WW),
    A = KK,
    C = WW,
    retract(event(R)),
    assert(event(fact)).

```

```

term(A,B,C,D,O,E,F) :-  

  ( E = G,  

    atoms(A,G,F)  

  ; E = H,  

    ( H = I,  

      variable(A,I,F)  

  ; H = J,  

    ( J = K,  

      anonymous(A,K,F)  

  ; J = L,  

    ( L = M,  

      list(A,D,M,F)  

  ; L = N,  

    ( N = O,  

      string(A,O,F)  

  ; N = P,  

    ( P = Q,  

      sig_brac_open(Q,R),  

      subterm(A,B,C,D,1200,R,S),  

      sig_brac_close(D,S,F)  

  ; P = T,  

      curly_bracket_open(T,U),  

      subterm(A,B,C,D,1200,U,V),  

      curly_brackets_close(D,V,F)  

    )  

  )  

) )  

) )  

) ).  

op(A,B,C,[token(A,name)|D],D) :-  

  true,  

  nonvar(A),  

  current_op(B,C,A).  

arguments(A,B,C,[],[]).  

arguments(A,B,C,D,E) :-  

  subterm(F,A,999,D,G),  

  rest_arg(A,H,C,G,E),  

  ( H = [],  

    B = [F]  

  ; B = [F|H]  

  ).  

arguments(A,B,C,D,E) :-  

  error(F),  

  ( F = 0,  

    retract(error(F)),  

    ...
  ).  


```

```

assert(error(1)),
error(arguments,D,A)
; true
).

rest_arg(A,B,C,D,E) :-  

    retract(ok(F)),
    G is F + 1,
    assert(ok(G)),
    H is C + 1,
    comma(I,A,D,J),
    retract(count_arg(K)),
    assert(count_arg(H)),
    arguments(A,B,H,J,E).
rest_arg(A,[],B,C,C).

exprlist([A|B],C,D,E) :-  

    ( D = F,  

        subterm(A,C,999,F,G),
        vertical_bar(G,H),
        rest_list(B,C,H,E)
    ; D = I,  

        subterm(A,C,999,I,J),
        comma(K,C,J,L),
        exprlist(B,C,L,E)
    ).
exprlist([A],B,C,D) :-  

    subterm(A,B,999,C,D).

rest_list(A,B,C,D) :-  

    ( C = E,
        variable(A,E,D)
    ; C = F,
        anonymous(A,F,D)
    ).

rest_list(A,B,C,D) :-  

    error(E),
    ( E = 0,
        retract(error(E)),
        assert(error(1)),
        error(rest_list,C,B)
    ; true
    ).

atoms(A,B,C) :-  

    ( B = [token(A,name)|C],
        true
    ; B = [token(A,number)|C],
        true
    ).
```

```

anonymous(A,[token(B,anonymous)|C],C) :-  

    true,  

    A = '_'.  

stop([token(.,limits)|A],A).  

stop([],A) :-  

    error(B),  

    ( B = 0,  

      error(stop,C),  

      examine([46],D),  

      stop(D,E)  

    ; true  

    ).  

sig_brac_open([token('(',limits)|A],A).  

curly_bracket_open([token('{',limits)|A],A).  

bracket_open([token('[',limits)|A],A).  

vertical_bar([token('!',limits)|A],A).  

empty_list([], [token([],name)|A],A).  

sig_brac_close(A,[token(')',limits)|B],B).  

sig_brac_close(A,B,B) :-  

    error(C),  

    C > 0.  

sig_brac_close(A,B,C) :-  

    operator(D),  

    D = 0,  

    error_sig1(A,B).  

sig_brac_close(A,B,C) :-  

    operator(D),  

    D > 0,  

    count_arg(E),  

    E = 0,  

    error_sig1(A,B).  

sig_brac_close(A,B,B) :-  

    operator(C),  

    C > 0,  

    count_arg(D),  

    D > 0,  

    nl,  

    colour(31),  

    write('The right parenthesis is missing.'),  

    nl,  

    write('Please, rewrite the term:'),  

    nl,  

    colour(37),  

    disconnect,

```

```
lexical(E),
assert(new_list(E)),
retract(error(F)),
assert(error(1)),
retract(count_arg(G)),
assert(count_arg(0)).  
  
error_sig1(A,B) :-  
    retract(error(C)),  
    assert(error(1)),  
    error(sig_brac_close,B),  
    examine([41],D),  
    subtract(A,B,E),  
    append(E,D,F),  
    append(F,B,G),  
    assert(new_list(G)).  
  
bracket_close(A,[token(']',limits)|B],B).  
bracket_close(A,B,C) :-  
    error(D),  
    ( D = 0,  
      retract(error(D)),  
      assert(error(1)),  
      error(bracket_close,B),  
      examine([93],E),  
      subtract(A,B,F),  
      append(F,E,G),  
      append(G,B,H),  
      assert(new_list(H))  
    ; true  
    ).  
  
curly_brackets_close(A,[token('{',limits)|B],B).  
curly_brackets_close(A,B,C) :-  
    error(D),  
    ( D = 0,  
      retract(error(D)),  
      assert(error(1)),  
      error(curly_brackets_close,B),  
      examine([125],E),  
      subtract(A,B,F),  
      append(F,E,G),  
      append(G,B,H),  
      assert(new_list(H))  
    ; true  
    ).  
  
comma(A,B,C,C) :-  
    error(D),  
    D > 0.
```

```

comma('!',A,B,C) :-  

    error(D),  

    D = 0,  

    ok(E),  

    E = 0,  

    error_list(bracket_open,B,A).  

comma(A,B,[token(C,limits)|D],D) :-  

    true,  

    C == ',',  

    A = ',,'.  

comma(A,B,[token(C,limits)|D],E) :-  

    true,  

    C == '!',  

    A = '!',  

    retract(ok(F)),  

    assert(ok(0)),  

    comma(A,B,D,E).  

error(A,B) :-  

    msg(A,C),  

    take_out(B,D),  

    disconnect,  

    print([C,' --> '|D]),  

    nl.  

error(A,B,C) :-  

    !,  

    msg(A,D),  

    take_out(B,E),  

    disconnect,  

    print([D,' --> '|E]),  

    nl,  

    subtract(C,B,F),  

    ask,  

    lexical(G),  

    append(F,G,H),  

    assert(new_list(H)).  

error_list(A,B,C) :-  

    count_arg(D),  

    D = 0,  

    msg(A,E),  

    write(E),  

    nl,  

    examine([91],F),  

    subtract(C,B,G),  

    last(H,G),  

    subtract(G,[H],I),  

    nl,

```

```

append([H],B,J),
last(K,I),
subtract(I,[K],L),
append([K],J,M),
append(L,F,N),
append(N,M,O),
assert(new_list(O)),
retract(error(P)),
assert(error(1)).
error_list(A,B,C) :-
    write('The left bracket is missing.'),  

    nl,  

    write('Please rewrite the term:'),  

    nl,  

    disconnect,  

    lexical(D),  

    assert(new_list(D)),  

    retract(error(E)),  

    assert(error(1)).

take_out([],[]) :-  

    !.  

take_out([token(A,B)|C],[A|D]) :-  

    take_out(C,D).

msg(sig_brac_close,'The right parenthesis is missing. I myself  

arrange it.') :-  

    !.  

msg(bracket_close,'The right bracket is missing. I myself  

arrange it.') :-  

    !.  

msg(curly_brackets_close,'The right curly bracket is missing. I  

myself arrange it.') :-  

    !.  

msg(rest_list,'The tail of the list must be represented by  

variable.') :-  

    !.  

msg(arguments,'Missing arguments') :-  

    !.  

msg(bracket_open,'The left bracket is missing. I myself arrange  

it.') :-  

    !.  

msg(stop,'The end point is missing. I myself arrange.').

ask :-  

    print('Arrange the error and rewrite the rest of the term:  

').

```

```

print(A) :-  

    colour(31),  

    nl,  

    retract('L'(B)),  

    remove(B),  

    ( A = [C|D],  

        format(A)  

    ; write(A),  

        colour(37)  

    ),  

    ( B >= 23,  

        assert('L'(13))  

    ; E is B + 1,  

        assert('L'(E))  

    ).  

disconnect :-  

    ( 'L'(A),  

        A =:= 13  

    ; assert('L'(13))  

    ).  

connect :-  

    retract('L'(A)),  

    remove(A),  

    ( A >= 23,  

        assert('L'(13))  

    ; B is A + 1,  

        assert('L'(B)),  

        colour(37)  

    ).  

format([]) :-  

    nl,  

    !.  

format([A|B]) :-  

    write(A),  

    format(B).  

remove(A) :-  

    ( A =:= 13  

    ; true  

    ).  

sequence([]).  

sequence([A|B]) :-  

    put(A),  

    sequence(B).  

escape :-  

    put(27).

```

```
colour(A) :-  
    escape,  
    name('[', B),  
    sequence(B),  
    name(A, C),  
    sequence(C),  
    name(m, D),  
    sequence(D).
```

Comments:

The program ‘lexical/1’ reads the terms written by the user and executes a lexical analysis. The syntactical analysis is done by ‘analyse/3’. In case of errors occurring during the writing, they are detected, notified and corrected by ‘error/3’. When there is an ambiguous situation the user is called by ‘error/2’ to help the correction. The lexical analyzer is also able to separate the tokens of each term through ‘gettoken/3’. There are several possibilities of tokens: name (functor and atom), limits (‘;’, ‘(’, ‘)’, ‘:’, ‘[’, ‘]’, ‘!’, ‘{’, ‘}’, ‘|’), variables, anonymous (‘_’), numbers (integer and floating point) and strings. The syntactical analysis of each term is done along two steps: analysis of the operators part by ‘subterm/5 (1200)’ and analysis of the following part by ‘subterm/5 (999)’. As a matter of fact, each term can be a generator, a functor followed by a left parenthesis, arguments, a right parenthesis, or stop or a comma, an atom, a list and a string. After error detection and correction a new list (R) is generated which is re-processed by the program. When the terms are goals or facts they are executed by ‘execute/1’. Interaction is supported by ‘send/1’ till the user writes ‘end’.

Acknowledgments

First of all, we would like to thank the authors who have offered us their programs and those from whom we have borrowed the examples included herein. In developing this book over a number of years, we have benefited from discussions with many colleagues, including Luis Moniz Pereira, Fernando Pereira, Salvador Pinto de Abreu, António Porto, Rosa Viccari, students José Silva, Margarida Ferreira, Maria de Jesus Fraga, Clara Guerra, Luis Almeida and António Rodrigues in courses at Universidade de Lisboa, and students Isabel Nunes, Alice Nunes and Olga Santos during training periods at LNEC. We are especially grateful to Pavel Brazdil, Robert Corlett, Jan Sebelik, Alain Grumbach and Peter Swinson for extended commentaries on earlier drafts of this book during the reviewing period. We will be pleased for any comments and corrections regarding these examples, and for any additional programs to be included in a future edition.

Finally, thanks are due to LNEC where the work behind this book has been carried out since 1979 when the idea of designing the first book of one of us (H.C.), "How to solve it with Prolog", was brought up. The interaction with anonymous referees of Springer-Verlag allowed us also to refine and to improve the new book's shape by taking into account the user's point of view. We are very grateful to Gillian Hayes who generously contributed her time and expertise to this book by making all the necessary copy-editing changes during the proofreading process. The editing of all the manuscript was a painful job, and only with the help of Ermelinda Ribeiro was it possible to finish it.

Appendix 1 Prolog Implementations

Everything began with the Marseille interpreter. In fact, in the early seventies Colmerauer's group presented the first Prolog interpreter. From these times to our days more than ten years have passed and we have traveled a long way. Very important marks along this way are the Prolog implementations of Edinburgh. These implementations are based on a two volume paper by David Warren, "Implementing Prolog. Compiling predicate logic programs", published in 1977. For the first time a compiler was proposed and a new and clearer syntax adopted.

This implementation for the DECsystem-10 became almost the standard for Prolog, and almost all the implementations that have been put forward since then use the same syntax and the same set of built-in predicates. And, the concept of structure-sharing has been adopted for the internal representation.

In the late seventies and early eighties a lot of new implementations appeared. Some of them are inspired by the Marseille version and others are inspired by the DECsystem-10 Prolog. However, some of them have important innovations. Among them we would like to mention MU-Prolog, from Melbourne University, in which some "traditional" problems, namely negation, are solved.

An evolution in the languages used for the implementations has also occurred. While the Marseille interpreter used FORTRAN the newest versions use C and PASCAL. The analysis of this evolution is also very interesting. Briefly, we can say that the evolution of the concepts present in the Prolog implementations is parallel to the evolution of computer science, therefore these implementations require higher and higher level languages.

A lot of research is now under way in several Artificial Intelligence communities in what concerns the programming environment and database access for Prolog. These aspects will permit a more sophisticated software development environment for Prolog, therefore contributing to the maturity of the language. In other words, the Prolog years have just begun and the Japanese Fifth-Generation Project has opened a large new horizon.

Computer System	Programming Language	Implementation	Year	Authors	Site or University
IBM 360	ALGOL-W	interpreter	1970	Roussel	Marseille
IBM 360	FORTRAN	interpreter	1972	Battani Melloni	Marseille
ICL-1903/A, ICL-105/E	CDL	interpreter	1975	Szeredi et al.	Budapest

Computer System	Programming Language	Implementation	Year	Authors	Site or University
IBM 370/158 (VM/CMS), IBM 3031,4341	ASSEMBLY	interpreter	1976	Roberts	Waterloo
Solar Exorciser "	ASSEMBLY	interpreter	1976	P.Roussel	Marseille
IBM 370	PASCAL CDL	interpreter interpreter	1976 1976	Bruynooghe Szeredi et al.	Louvaine Budapest
ICL 4/70, ICL 1903A	CDL	interpreter	1976	Szeredi et al.	Budapest
Honeywell Bull 66/20	CDL	interpreter	1976	Szeredi et al.	Budapest
EMG 840	CDL	interpreter	1976	Szeredi et al.	Budapest
ODRA 1304	CDL	interpreter	1976	Szeredi et al.	Budapest
RIAD R22	CDL	interpreter	1977	Szeredi et al.	Budapest
DEC-10/20	MACRO-10 and PROLOG	compiler and interpreter	1977	Warren F.Pereira L.Pereira C.Mellish	Edinburgh and Lisbon
PDPII/60 (UNIX)	ASSEMBLY	interpreter	1978	Nakashima	Edinburgh
FACOM230/38 EXORCISER (M6800)	PASCAL CANDIDE PASCAL UCSD	interpreter interpreter (PROLOG II)	1978 1979	Colmerauer Kanoui Caneghan Damas	Tokyo Marseille
ICL 4/75 EMAS	IMP	interpreter	1979	Dwiggins	Edinburgh
PDP-11, CDC6600	FORTH and SNOBOL	compiler	1979	McCabe	Los Angeles
IBM/370	PASCAL ¹	interpreter	1979	Kluzniak Sibert Robinson Komorowski	London (IC) Warsaw Syracuse
CDC6000	PASCAL LISP	interpreter interpreter (LOGLISP)	1979 1980	Bende Futo	Uppsala
SIEMENS 7.755, IBM3031, VAX 11/780	CDL2	interpreter (T-PROLOG)	1980	Donz	Budapest
IBM 43XX (VS/CMS)	PASCAL	interpreter (FOLL-PROLOG)	1981	Colmerauer Caneghan Kanoui	Grenoble
APPLEII, VAX 11/780 (VMS), IBM-PC,	CANDIDE	interpreter (PROLOG II)	1981		Marseille

¹ This version differs significantly from other systems in the control primitives.

Computer System	Programming Language	Implementation	Year	Authors	Site or University
LISA, SM 90, MICROMEGA					
NORD-100	LISP	interpreter	1981	Fogelholm	Stockholm
HITACM200H,	UTILISP	interpreter (PROLOG/KR)	1981	Nakashima	Tokyo
FACOM M180II					
"	MACLISP	interpreter (DURAL)	1981	Goto	Tokyo
"	LISP	interpreter (YAQ)	1981	Carlsson	Uppsala
Z80	ASSEMBLY	interpreter	1981	McCabe	London
North Star					
PDP-11,	ASSEMBLY	interpreter (MICRO-PROLOG)	1981	Goodall	Oxford
LSI-11					
-	LISP	interpreter (UNIFORM: LISP + PROLOG + ACT1)	1981	Kahn	Uppsala
VAX 11/750	C	interpreter (PROLOG-C)	1981	Bruynooghe	Louvain
NEC, VAX 11/780 (UNIX)	C	interpreter	1982	Yamamoto	Kawasaki
VAX11/780 (UNIX)	C	interpreter	1982	Yokota	Tokyo
"	C	interpreter (EPILOG)	1982	Porto	Lisbon
VAX11/780 (UNIX 4.1 BSD) or Eunice under VMS M68000	C	interpreter (C-PROLOG)	1982	F. Pereira L. Damas L. Byrd	Edinburgh
VAX 11/780 (VMS), Z8000	POP-11 + ASSEMBLY	compiler (POPLOG)	1982	Mellish	Sussex
PRIME 750	PASCAL	interpreter (PORTABLE PROLOG)	1982	Bulmer	York
-	CDL2	interpreter (MPROLOG)	1982	Szeredi	Budapest
-		interpreter (PARALOG)	1982	Aida	Tokyo
-	LISP	interpreter (PROLISP)	1982	Zanon	Lannion
VAX 11/780 (UNIX), PCS QU68000, Perkin Elmer (UNIX3200), IBM PC (MS-DOS, CP/M-86)	C	interpreter (IF/PROLOG)	1983	Szeredi Futo	Munich
LMI (Lisp Machine)	LISP	interpreter (LOGIS)	1983	Gloess	Campiègne

Computer System	Programming Language	Implementation	Year	Authors	Site or University
Z80 (CP/M-80), 380 Z, 8088/86 (MS-DOS/ PC DOS)	-	interpreter (MICRO-PROLOG3)	1983	McCabe	London
SM90 (UNIX,SOL) Lisp Machine	PASCAL	interpreter	1983	Hentinger	Paris
VAX 11/780 (UNIX, VMS)	FRANZLISP	compiler (LM-PROLOG) interpreter (LISLOG)	1983	Carlsson Kahn Bourgault Dincbas Le Pape Barberge	Uppsala Lannion
-	PASCAL (MULTICS)	interpreter (PROLOG/CNET)	1983	Nilsson	Issy-les- Moulineaux
IBM 370 (CMS)	PASCAL	interpreter	1983	Szpakowicz	Warsaw
-	MACLISP	interpreter (FOOLOG)	1983	Greussay	Paris
-	VLISP	interpreter (LOVLISP)	1983	Lloyd	Melbourne
VAX11/780 (UNIX), M68000	C	interpreter (MU-PROLOG 2.5)	1983	Furukawa	Tokyo
Lisp Machine (ELIS)	LISP	interpreter (MANDALA) interpreter (TAO: LISP + PROLOG + SMALLTALK)	1983	Takeuchi	Tokyo
VAX-11 (UNIX), Z80, M6809, INTEL8089 Victor, NORSK- DATA	C	interpreter (UNSW-PROLOG Version 4)	1983	Sammut	Kensington
IBM-PC	ASSEMBLY	interpreter (D-PROLOG)	1984	Donz	Paris
Burroughs Alegre 86700 (MCP)	ASSEMBLY	interpreter (PROLOG/Sc)	1984	C.Pereira	São Carlos
IBM 43xY/VM, HB68/MULTICS, SCL2900/VME-8, MICROMEGA 32 (UNIX), Burroughs B 6700 (MCP), IBM 30XY/VM-CMS	PASCAL	interpreter (PROLOG-CRISS)	1984	Low	Porto
VAX 11/7XY (UNIX 4.2, VMS), SUN(M68000)-2 MEGA-FRAME	C	interpreter (Quintus)	1984	Donz et al.	Grenoble Stanford

Computer System	Programming Language	Implementation	Year	Authors	Site or University
VAX 11/7XY	C	interpreter (FROG)	1984	Ducasse Faget Grumbach	Marcoussis
SUN-2, SUN-3, APOLLO, VAX	C	interpreter + compiler + environment	1986	BIM Research Department	Belgium
MAC, SUN(M68000), VAX (UNIX, VMS)	C	interpreter + compiler	1987	L. Damas	Portugal

Efficiency of Prolog Implementations

Computer			LIPS
APPLE II		interpreter	10
EXORCISER	(M6800)	interpreter	33
Z-80	MICRO-PROLOG	interpreter	240
SOLAR		interpreter	250
IBM-PC	PROLOG-2	interpreter	500
DEC-10	FORTRAN	interpreter	500
IBM-PC	PROLOG-2	compiler	2000
VAX 11/780	C-PROLOG	interpreter	3000
SUN-2	Quintus	compiler	20000
VAX 11/780	Quintus	compiler	23000
IBM 43	Waterloo	interpreter	25000
PSI	fifth generation	computer	25000
DEC-10	Edinburgh	compiler	30000
IBM 3033	Waterloo	interpreter	33000
DEC-20	Edinburgh	compiler	43000
SUN-3	Quintus	compiler	80000
SUN-3	BIM	compiler	200000

Appendix 2 Commercial Products

A number of commercial products around the programming language Prolog appeared recently on the market. As an example, we present a tentative list:

Name	Firm (Country)
AAIS PROLOG	Advanced AI Systems (USA)
ALS PROLOG 1.0	Applied Logic Systems (USA)
ARITY/PROLOG	Arity Corporation (USA)
BIM _ PROLOG	Belgium Institute of Management (B)
HP PROLOG	Hewlett-Packard Co. (USA)
IF/PROLOG	Springer Software (FRG) and Interface Computer GmbH (FRG)
LIGHTNING FAST PROLOG	Advanced AI Systems (USA)
LPA PROLOG	Logic Programming Associates (GB)
MAC _ PROLOG	Logic Programming Associates (GB)
MICRO-PROLOG	Logic Programming Associates (GB)
MPROLOG	Epsilon GmbH (FRG) and Logicware Inc. (CDN)
POPLOG	Systems Designers (GB)
PROLOG-1	Expert Systems International (GB)
PROLOG-2	Expert Systems International (GB)
PROLOG II	Prolog IA (F)
PROLOG V	Chalcedony Software (USA)
PROLOG-86	Solution Systems (USA)
QUINTUS PROLOG	Quintus (USA)
SD-PROLOG	Systems Designers plc (GB) and Systems Designers International, Inc. (USA)
SIGMA-PROLOG	Logic Programming Associates (GB)
TOY PROLOG	Public Domain Software (GB)
TRILOGY 1.15	Complete Logic Systems (USA)
TURBO PROLOG	Borland International (USA)
YAP	Portu (P)

Appendix 3 Selection of Some Historical Prolog Applications

Problem Domain	Authors	Year
I) Pure Research		
Plane geometry	Welham	1976
Mechanics		
	Coelho & Pereira	1976
Symbolic calculus	Bundy et al.	1979
Natural language understanding		
	Kanoui	1973
	Bergman & Kanoui	1975
	Kanoui	1975
	Colmerauer	1971
	Pasero	1972
	Colmerauer & Kanoui	1973
	Roussel & Pasero	1973
	Colmerauer	1974
	Colmerauer	1975
	Guizol	1975
	Pasero	1976
	Dahl	1977
	Pique	1978
	Mellish	1978
	Pereira & Warren	1979
	Milne	1979
	Coelho	1979
	Mellish	1981
	Pereira & Warren	1981
	Piquet & Sabatier	1982
	Colmerauer	1982
	Filgueiras	1983
Speech understanding	Batani & Meloni	1975
Learning	Brazdil	1978
Knowledge engineering: Chess	Emden	1980
Robotics		
	Bratko	1982
	Hileyan	1982
	Warren	1974
	Giannesini	1978
Database management	Pereira & Martins	1976
	Coelho	1976
II) Practical Programs		
Compiler writing	Colmerauer	1975
Interpreter writing	Warren	1977
	Pereira & Porto	1979
	Byrd	1979
	Pereira	1982

Problem Domain	Authors	Year
Distributed logic interpreter	Monteiro	1982
Computer utilities	Battani & Meloni	1975
Travel agent problem	Mellish	1976
	Silva & Cotta	1978
Computer catalogue	Dahl	1976
Plotter programs generation	Darvas	1976
Pollution control	Darvas	1976
Pesticide information	Darvas	1976
Statistics	Darvas	1976
Flat design	Markusz	1977
Building design	Markusz	1980
		1981
Architecture	Rodriguez	1978
	Swinson	1980
		1981
Layout generation of dimensional rectangular spaces	Laginha & Coucello	1981
Civil engineering legislation	Cotta & Silva	1978
Drug design aids	Darvas	1978
Drug interaction prediction	Darvas & Futó & Szeregi	1978
Carcinogenic activity	Darvas	1978
GT-42 Picture book	Santos	1979
Distribution of Portuguese families through a scale of income	Pereira	1979
Pre-registration appointments	Townshend	1979
Library manager	Coelho	1979
Intelligent analyst	Dwiggins	1979
List of equipment	Simoes	1980
Program writing	Gaspar	1980
Algebra of relational composition	Brainbridge & Skuce	1980
Fault finder system	Hammond	1980
Gas heater faults		
Car engine faults		
Automatic analysis and synthesis in CAD	Dincbas	1980
Modeling machine parts	Molnar & Markus	1981
Geographical information system	Warren & Pereira	1981
Electronic CAD	Pasztome-Varga	1981
Production control system	Markus, Molnar & Szelke	1981
Fixture design	Farkas et al.	1982
Specification support system	Farkas et al.	1982
Environmental resources evaluation	Pereira et al.	1982
Calculation of physicochemical properties of organic compounds	Darvas	1982
Migration decision-making	Roach	1982
Database system	Pique & Sabatier	1982
French public administration	Girau et al.	1982
Translator Edinburgh Prolog form into Micro-Prolog form	Townsend	1982
Configurer for DEC (VAX's)	McDermott	1982
Tracing package	Spacek	1982
Fast neutron reactor diagnostic system	Parcy	1982

Problem Domain	Authors	Year
Garden store assistant	Walker & Porto	1983
Database design	Parsaye	1983
Scheduling meetings	Saint-Dizier	1983
Diagnosis and treatment of cardiac arrhythmics	Mozetic	1983
Logic circuit synthesis	Uehara & Kawato	1983
Automatic model-building for regression analysis	Darvas	1983
Reactor fault diagnostic	Mizoguchi	1983
House manager	Hiroyuki	1984
Search of cancer therapy literature	Pollitt	1984
Tactical data fusion	Pecora	1984

References

- ANDREKA, H.; NEMETI, I. [1976] The Generalised Completeness of Horn Predicate-Logic as a Programming Language, DAI Research Report No.21, Univ. of Edinburgh, Univ. d'Aix-Marseille
- APT, K. R.; EMDEN, M. H. van [1980] Contributions to the Theory of Logic Programming Research Report CS-80-12, Univ. of Waterloo
- BATTANI, G.; MELONI, H. [1973] Interpreteur du Langage de Programmation PROLOG Rapport de DEA d'Informatique Appliquée
- BATTANI, G.; MELONI, H. [1974] Un Bel Example de PROLOG en Analyse et Synthese Groupe de IA, UER Luminy Univ. d'Aix-Marseille
- BATTANI, G.; MELONI, H. [1975] Mise en Œuvre des Contraintes Phonologiques Syntaxiques et Semantiques dans un Système de Comprehension Automatique de la Parole Groupe de IA, UER Luminy Univ. d'Aix-Marseille
- BAXTER, L. [1980] The Versatility of Prolog Sigplan Notices, Vol. 15, no. 12
- BERGMAN, M.; KANOUI, H. [1973] Application of Mechanical Theorem Proving to Symbolic Calculus Groupe de IA, UER Luminy Univ. d'Aix-Marseille
- BERGMAN, M.; KANOUI, H. [1975] SYCOPHANTE: Système de Calcul Formel et d'Integration Symbolique sur Ordinateur Groupe de IA, UER Luminy Univ. d'Aix-Marseille
- BOWEN, L. M.; BYRD, D. L.; CLOCKSIN, W. F. [1985] A Portable Prolog Compiler in The Proceed. of the Logic Programming Workshop
- BOYER, R. S.; MORE, J. S. [1972] The Sharing of Structure in Theorem-Proving Programs DAI Memo no. 47 Univ. of Edinburgh and Machine Intelligence 7.
- BRAMER, M. [1980] Reasoning About Knowledge: the "S and P" Problem AISB Quarterly No.37
- BRATKO, I. [1986] Prolog Programming for Artificial Intelligence Addison-Wesley
- BRAZDIL, P. [1978] Experimental Learning Model Proceed. of AISB78
- BRAZDIL, P. [1984] Use of Derivation Trees in Discrimination Proc. of ECAI
- BRUYNOOGHE, M. [1975] The Inheritance of Links in a Connection Graph Report CW2 Applied Mathematics and Programming Dept. Katholieke Univ. Leuven (Belgium)
- BRUYNOOGHE, M. [1976] An Interpreter for Predicate Logic Programs Part I : Basic Principles Report CW 10 Applied Mathematics and Programming Dept. Katholieke Univ. Leuven (Belgium)
- BRUYNOOGHE, M. [1977] An Interface Between Prolog and Cyber-EDMS in Logic and Data Bases Workshop Toulouse
- BRUYNOOGHE, M. [1978] Intelligent Backtracking for Horn Clause Logic Programs Applied Mathematics and Programming Dept. Katholieke Univ. Leuven (Belgium)
- BRUYNOOGHE, M. [1979 a] Analysis of Dependencies to Improve the Behaviour of Logic Programs Report CW19 Univ. Catholique de Louvaine
- BRUYNOOGHE, M. [1979b] Solving Combinatorial Search Problems by Intelligent Backtracking Report CW18 Univ. Catholique de Louvaine
- BURNHAM, W. D.; HALL, A. R. [1985] Prolog Programming and Applications Macmillan
- CAMPBELL, J. A. [1984] Implementations of Prolog Ellis Horwood
- CLARK, K. L.; KOWALSKI, R. [1977a] Predicate Logic as a Programming Language Department of Computing and Control Imperial College
- CLARK, K. L.; TARNLUND, S. A. [1977b] A First-Order Theory of Data and Programs Proc. IFIP 77

- CLARK, K. L.; McCABE, F. [1980] Prolog : a Language for Implementing Expert Systems Imperial College and Machine Intelligence 10 1982
- CLARK, K. L. [1981] An Introduction to Logic Programming Imperial College
- CLOCKSIN, W. F.; MELLISH, C. S. [1981] Programming in Prolog Springer-Verlag Third Edition (1987)
- CLOCKSIN, W. F. [1985a] Design and Simulation of a Sequential Prolog Machine New Generation Computing 3, 101-120
- CLOCKSIN, W. F. [1985b] Implementation Techniques for Prolog Data Bases Software - Practice and Experience Vol. 5, no. 7, 669-675
- COELHO, H.; PEREIRA, L. M. [1976] GEOM: A PROLOG Geometry Theorem Prover LNEC
- COELHO, H. [1977] Natural Language and Data Bases LNEC
- COELHO, H. [1979] A Program Conversing in Portuguese Providing a Library Service Ph. D. Thesis Univ. of Edinburgh
- COELHO, H.; COTTA, J. C. [1981] Prolog -- a Tool for Logic Programming and Problem Solving DECUS Symposium
- COELHO, H. [1982] Prolog: a Programming Tool for Logical Domain Modeling Proc. of the IFIP/IIASA Working Conference on Processes and Tools for Decision Support
- COELHO, H. [1983a] Prolog for Databases Pocitace a umela inteligencia (Computers and Artificial Intelligence) no. 1
- COELHO, H. [1983b] The Art of Knowledge Engineering with Prolog INFOLOG Research Report RR06, DEIOC/BESCL
- COELHO, H.; RODRIGUES, A. J.; SERNADAS, A. [1984a] Towards Knowledge Based INFOLOG Specifications Decision Support Systems Journal, Vol. 1, no. 2
- COELHO, H. [1984b] Logic Programming at Work: the Case of a Civil Engineering Environment Proc. of the 3rd International Conference on Artificial Intelligence and Information-Control Systems of Robotics; Computers and Artificial Intelligence Journal, Vol. 4, no. 2, 115-124 (1985)
- COELHO, H.; RODRIGUES, A. J. [1984c] Knowledge Architecture for Management Environments Proc. of the Working Conference on Knowledge Representation for Decision Support Systems
- COELHO, H. [1984d] Logic Programming Teaching: Aids and Ideas Proc. of the International Conference on Artificial Intelligence, Methodology, Systems and Applications
- COELHO, H. et al. [1985a] How to Solve it with Prolog LNEC, 5th. edition
- COELHO, H. [1985b] Logic Programming Bibliography LNEC
- COELHO, H. [1986a] Library Manager: a Case Study in Knowledge Engineering in Intelligent Information Systems, Roy Davies (ed.), Ellis Horwood
- COELHO, H.; MITTERMEIR, R. [1986b] A Survey of Formal Knowledge Modelling LNEC, Memoria no. 661
- COELHO, H.; PEREIRA, L. M. [1986c] Automated Reasoning in Geometry Theorem Proving with Prolog Journal of Automated Reasoning, Vol. 2, 329-390
- COLMERAUER, A. et al. [1973] Un Système de Communication Homme-Machine en Français Groupe de IA, UER Luminy Univ. d'Aix-Marseille
- COLMERAUER, A. [1974] Programmation en Langue Naturelle Groupe de IA, UER Luminy Univ. d'Aix-Marseille
- COLMERAUER, A. [1975] Les Grammaires de Metamorphose Groupe de IA, UER Luminy Univ. d'Aix-Marseille
- COLMERAUER, A. [1977] Un Sous-Ensemble Interessant du Français Groupe de IA, UER Luminy Univ. d'Aix-Marseille
- COLMERAUER, A. [1979a] Sur les Bases Théoriques de Prolog Groupe de IA, UER Luminy Univ. d'Aix-Marseille
- COLMERAUER, A.; PIQUE, J. [1979b] About Natural Logic Groupe de IA, UER Luminy Univ. d'Aix-Marseille
- COLMERAUER, A.; KANOUI, H.; CANEGHEM, M. [1979c] Etude et Réalisation d'un Système Prolog Groupe de IA, UER de Luminy Univ. d'Aix-Marseille
- COLMERAUER, A. [1982] Prolog and Infinite Trees in Clark and Tarnlund (eds.), Logic Programming, Academic Press
- COLMERAUER, A. [1985] Prolog in 10 Figures Communications of the ACM, Vol. 28, no. 12, pp. 1296-1310

- COLMERAUER, A. [1986] Prolog: un Langage pour l'Intelligence Artificielle... et la Cinquième Generation Minis et Micros no. 183
- CORLETT, R. A.; TODD, S.J. [1983] A Basic Interpreter for Coroutining Logic Programming Newsletter, 5
- COTTA, J. C.; SILVA, A. P. [1978] Interação com Bases de Dados LNEC
- COTTA, J.C. [1980] Experiência de Utilização da Lingua Natural no Acesso a Bases de Dados Comunicação ao 1o. CPI80
- DAHL, V.; SAMBUC, R. [1976] Un Système de Banque de Données en Logique du Premier Ordre, en Vue de sa Consultation en Langue Naturelle Groupe de IA, UER Luminy Univ. d'Aix-Marseille
- DAHL, V. [1977a] Some Experiences on Natural Language Question-Answering Systems in Logic and Data Bases, Workshop Toulouse
- DAHL, V. [1977b] Un Système Deductif d'Interrogation de Banques de Données en Espagnol Groupe de IA, UER Luminy Univ. d'Aix-Marseille
- DARVAS, F.; FUTO, I.; SZEREDI, P. [1978] Logic Based Program System for Predicting Drug Interactions Int. J. Bio-Medical Computing(9)
- DELIYANNI, A.; KOWALSKI, R. A. [1977] Logic and Semantic Networks Department of Computing and Control Imperial College in Logic and Data Bases Workshop Toulouse
- ELCOCK, E. W. [1983] The Pragmatics of Prolog: Some Comments Proceed. of the Logic Programming Workshop
- EMDEN, M. van [1974a] First-Order Predicate Logic as a High-Level Program Language DAI, MIP-R-106 Univ. of Edinburgh
- EMDEN, M. van [1974b] The Semantics of Predicate Logic as a Programming Language DAI, Memo 73 Univ. of Edinburgh
- EMDEN, M. van [1975] Programming with Resolution Logic Research Report CS-75-30 DCS, Univ. of Waterloo, Canada
- EMDEN, M. van; KOWALSKI, R. [1976a] Verification Conditions as Representations for Programs Research Report CS-76-03 DCS, Univ. of Waterloo, Canada
- EMDEN, M. van [1976b] Unstructured Systematic Programming Research Report CS-76-09 DCS, Univ. of Waterloo, Canada
- EMDEN, M. van [1976c] A Proposal for an Imperative Complement to PROLOG Research Report CS-76-39 DCS, Univ. of Waterloo, Canada
- EMDEN, M. van [1976d] Deductive Information Retrieval on Virtual Relational Databases Research Report CS-76-42 DCS, Univ. of Waterloo, Canada
- EMDEN, M. van [1976e] Logic Programs for Querying Relational Databases Working Paper DCS, Univ. of Waterloo, Canada
- EMDEN, M. van [1977] Computation and Deductive Information Retrieval Research Report CS-77-16 DCS, Univ. of Waterloo, Canada
- EMDEN, M. van [1978] Relational Programming Research Report CS-78-48 DCS, Univ. of Waterloo, Canada
- EMDEN, M. van [1980] Chess-endgame Advice: a Case Study in Computer Utilization of Knowledge Research Report CS-80-05 DCS, Univ. of Waterloo
- ENNALS, R. [1982] Teaching Logic as a Computer Language to Children AISB Newsletter no. 45
- ENNALS, R. [1984] Beginning with Micro-Prolog Ellis Horwood, 2nd. edition
- FARKOS, Zs.; SZEREDI, P. [1983] Getting Started with Prolog Institute for Coordination of Computer Techniques
- FIKES, R.E.; NILSSON, N. [1971] STRIPS: A New Approach to the Application of Theorem Proving in Problem Solving Artificial Intelligence, Vol. 2
- FUTO, I.; DARVAS, F.; CHOLNOKY, E. [1977a] Practical Applications of an Artificial Intelligence Language Preprints of the Second Hungarian Computer Science Conference, Budapest
- FUTO, I.; SZEREDI, P.; DARVAS, F. [1977b] Some Implemented and Planned PROLOG Applications in Logic and Databases Workshop Toulouse
- GENESERETH, M. R.; NILSSON, N. [1987] Logical Foundations of Artificial Intelligence Morgan Kaufmann
- GIANNESINI, F.; KANQUI, H.; PASERO, R.; CANEGHEM, M. [1985] Prolog InterEditions
- GRUMBACH, A. [1983] Knowledge Acquisition in Prolog Proc. of the 1st International Logic Programming Conference

- HAMMOND, P. [1980] Logic Programming for Expert System Technical Report DOC 82/4 Imperial College
- HANSSON, A.; HARIDI, S.; TÄRNLUND, S.-Å. [1982] Properties of a Logic Programming Language in "Logic Programming", K. Clark and S. A. Tärnlund eds. Academic Press and also report 8/81 Computing Science Dept. Uppsala University
- HANUS, M. [1986] Problemlösen mit Prolog B.G. Teubner
- HOGGER, C.J. [1984] Introduction to Logic Programming Academic Press
- HORN, A. [1951] On Sentences Which are True of Direct Unions of Algebras J. Symbolic Logic, 16, 14-21
- HUSTLER, A. [1982] Programming Law in Logic Research Report CS - 82-13, Univ. of Waterloo
- KANOUI, H. [1973a] Some Aspects of Symbolic Integration via Predicate Logic Programming Proceed. of IJCAI
- KANOUI, H. [1973b] Application de la Démonstration Automatique aux Manipulations Algébriques et a l'Intégration Formelle sur Ordinateur Groupe de IA, UER Luminy Univ. d'Aix-Marseille
- KANOUI, H. [1975] Rapport d'Activité du G.I.A. (Intégration Symbolique) Groupe de IA, UER Luminy Univ. d'Aix-Marseille
- KANOUI, H.; BERGMAN, M. [1977] Generalized Substitutions Groupe de IA, UER Luminy Univ. d'Aix-Marseille
- KANOUI, H.; CANEGHEM, M. van [1979] Implementing a Very High Level Language on a Very Low Cost Computer Groupe de IA, UER Luminy Univ. d'Aix-Marseille
- KLUZNIAK, F.; SZPAKOWICZ, S. [1981] PROLOG for Programmes, an Outline of a Teaching Method in Papers in Logic Programming I Stanislaw Szpakowicz (ed.) Univ. of Warsaw
- KLUZNIAK, F.; SZPAKOWICZ, S. [1985] Prolog for Programmes Academic Press
- KOWALSKI, R.; KUEHNER, D. [1971] Linear Resolution with Selection Function AI Journal 2
- KOWALSKI, R. [1973a] An Improved Theorem-Proving System for First-Order Logic DAI, Memo 65 Univ. of Edinburgh
- KOWALSKI, R. [1973b] Predicate Logic as Programming Language DAI, Memo 70 Univ. of Edinburgh and Proc. IFIP 74 North-Holland
- KOWALSKI, R. [1974a] A Proof Procedure Using Connection Graphs DAI, Memo 74 Univ. of Edinburgh and JACM Vol. 22 no. 4 – Oct/1975
- KOWALSKI, R. [1974b] Logic for Problem Solving DAI, Memo 75 Univ. of Edinburgh
- KOWALSKI, R. [1976a] Algorithm = Logic + Control Department of Computing and Control Imperial College
- KOWALSKI, R. [1976b] Logic and Data Bases Department of Computing and Control Imperial College
- KOWALSKI, R. [1977a] Logic as Programming Language Visit to N. America, 25 March–29 April 1977 Department of Computing and Control Imperial College
- KOWALSKI, R. [1977b] General Laws in Data Description in the Proceed. of the Logic and Databases Workshop, Toulouse
- KOWALSKI, R. [1979] Logic for Problem Solving North-Holland
- KOWALSKI, R. [1981a] Logic as a Database Language Imperial College
- KOWALSKI, R. [1981b] Prolog as a Logic Programming Language AICA Congress
- KOWALSKI, R. [1985a] Logic-Based Open Systems Imperial College
- KOWALSKI, R. [1985b] The Limitations of Logic Imperial College
- LEE, N. S.; ROACH, J. W. [1986] A Prolog Shell for Implementating Expert Systems AT&T Bell Laboratories
- LEE, R. M.; COELHO, H.; COTTA, J. C. [1985a] Temporal Inferencing on Administrative Data-Bases Information Systems Journal, Vol. 10, no. 2, 197–206
- LEE, R. M. [1985b] A Logic Programming Approach to Building Planning and Simulation Models Department of General Business Working Paper 85/86-3-2 University of Texas
- LLOYD, J. W. [1984] Foundations of Logic Programming Springer-Verlag
- MCDERMOTT, D. [1980] The Prolog Phenomenon Sigart Newsletter no. 72
- MALER, O.; SCHERZ, Z.; SHAPIRO, E. [1984] A New Approach for Introducing Prolog to Naive Users Weizmann Institute of Science
- MALPAS, J. [1987] Prolog: a Relational Language and Its Applications Prentice-Hall

- MARCUS, C. [1986] Prolog Programming: Applications for Data Base Systems, Expert Systems, and Parsers Addison-Wesley
- MARKUSZ, Z. [1977] How to Design Variants of Flats Using Programming Language Prolog Based on Mathematical Logic Information Processing, B. Gilchrist (ed.) IFIP - 1977, North-Holland
- MELLISH, C.S. [1977] An Approach to the GUS Travel Agent Problem Using Prolog DAI Univ. of Edinburgh
- MELLISH, C.S. [1978a] Syntax - Semantics Interaction in Natural Language Parsing DAI, Working Paper no. 31 Univ. of Edinburgh
- MELLISH, C. [1978b] Preliminary Syntactic Analysis and Interpretation of Mechanics Problems Stated in English DAI Working Paper no. 48 Univ. of Edinburgh
- MELONI, H. [1976] Prolog - Mise en Route de l'Interpréteur et Exercices Groupe de IA, UER Luminy Univ. d'Aix-Marseille
- MONTEIRO, L. [1982] A Small Interpreter for Distributed Logic Logic Programming Newsletter, 3
- MOORE, J.S. [1973] Computational Logic: Structure Sharing and Proof of Program Properties - Parts I & II DAI Memo 67 Univ. of Edinburgh
- NAISH, L. [1986] Negation and Control in Prolog Springer-Verlag
- NILSSON, M.; CAMPBELL, J.A. [1980] A Multinational Answer to McCarthy's "S and P" Problem AISB Quarterly No.37
- ODETTE, L. [1987] How to Compile Prolog AI Expert, Vol. 2, no. 8
- O'KEEFE, R. [1983] Prolog Compared with Lisp? Sigplan Notices, Vol. 18, no. 5
- PAIN, H.; BUNDY, A. [1979] What Stories Sould We Tell Novice Programmers? Univ. of Edinburgh
- PARSAYE, K. [1983] Database Management, Knowledge Base Management and Expert System Development in Prolog Proceed. of the Logic Programming Workshop
- PEREIRA, F.; WARREN, D. H. [1978] Definite Clause Grammars Compared with Augmented Transition Network DAI Univ. of Edinburgh and Artificial Intelligence 13, 3, 1980
- PEREIRA, F. [1979] Extrapolation Grammars DAI, Working paper no.59 Univ. of Edinburgh
- PEREIRA, F. [1982] C-Prolog User's Manual EdCAAD
- PEREIRA, F. [1983] Logic for Natural Language Analysis SRI International
- PEREIRA, L. M. [1977] Prolog, uma Linguagem de Programação em Logica LNEC
- PEREIRA, L. M. [1978a] Prolog, Linguagem de Resolução de Problemas em Logica (Part 1 & 2) in Informatica, Vol. 2, no. 4 1978, Vol. 2, no. 6, 1979
- PEREIRA, L. M.; MONTEIRO, L. F. [1978b] The Semantics of Parallelism and Co-Routining in Logic Programs LNEC and Colloquium on Mathematical Logic in Programming, Salgotarjan, Hungary North-Holland
- PEREIRA, L. M.; PEREIRA, F.; WARREN, D. [1978c] User's Guide to DECsystem 10 PROLOG LNEC
- PEREIRA, L. M. [1979a] Cálculo da Distribuição das Famílias Portuguesas por Escalões de Rendimento LNEC
- PEREIRA, L. M. [1979b] Backtracking Intelligently in AND/OR trees Universidade Nova de Lisboa
- PEREIRA, L. M.; COELHO, H. [1979 c] A Lógica, Instrumento de Comunicação em Português com o Computador LNEC
- PEREIRA, L. M.; PORTO, A. [1979 d] Intelligent Backtracking and Sidetracking in Horn Clause Programs - the Theory Universidade Nova de Lisboa, Lisbon
- PEREIRA, L. M.; PORTO, A. [1979 e] Intelligent Backtracking and Sidetracking in Horn Clause Programs - the Implementation Universidade Nova de Lisboa, Lisbon
- PEREIRA, L. M.; PORTO, A. [1980] Selective Backtracking for Logic Programs Departamento de Informática, CIUNL no. 1/80 Univ. Nova de Lisboa
- PEREIRA, L. M.; PORTO, A. [1982a] Pure Lisp in Pure Prolog Logic Programming Newsletter, 3
- PEREIRA, L. M. [1982b] A Prolog Demand Driven Computation Interpreter Logic Programming Newsletter, 4
- PLAISTED, D.A. [1984] The Occur-Check Problem in Prolog New Generation Computing 2, 309-322
- PORTO, A. [1981] A Prolog Program for the 'S-P Problem' Logic Programming Newsletter, 1

- ROBINSON, J.A. [1965] A Machine-Oriented Logic Based on the Resolution Principle J. of the ACM, Vol. 12
- ROBINSON, J.A. [1980a] The Logic Basis of Programming by Assertion and Query Syracuse University
- ROBINSON, J.A.; SIBERT, E. E. [1980b] Logic Programming in Lisp Syracuse University
- ROBINSON, J.A.; SIBERT, E. E. [1980c] LOGLISP - an Alternative to Prolog Syracuse University
- ROGERS, J. [1987] A Turbo-Prolog Primer Addison-Wesley
- ROSS, P. [1982] Teaching Prolog to Undergraduates AISB Newsletter no. 45
- ROUSSEL, P. [1975] PROLOG, Manuel de Reference et d'Utilisation Univ. d'Aix-Marseille
- SAMMUT, R. A.; SAMMUT, C. A. [1983] Prolog: a Tutorial Introduction The Australian Computer Journal, Vol. 15, no. 2
- SANTOS, M. J. [1979] Interface Grafica em Prolog LNEC
- SCHERZ, Z.; MALER, O.; SHAPIRO, E. [1986] The Use of Logic Programming in Education in J. Moonen and T. Plomp (eds.), Proceed. of the First European Conference on Education and Information Technology (EURIT86) Pergamon Press
- SCHLOBOHM, D. A. [1984] Introduction to Prolog Robotics Age November
- SEBELIK, J. [1983] A Problem Description in Prolog Logic Programming Newsletter, 5
- SHAPIRO, E. Y. [1982] Alternation and the Computational Complexity of Logic Programs Research Report 239, Yale University
- SHAPIRO, S. [1987] Encyclopedia of Artificial Intelligence John Wiley & Sons
- SILVA, A. P.; COTTA, J. C. [1978] The Travel Problem Revisited LNEC
- SOMEREN, M. W. van [1985] Beginners' Problems in Learning Prolog Univ. of Amsterdam
- STERLING, L.; SHAPIRO, E. [1986] The Art of Prolog MIT Press
- SWINSON, P. S. G. [1981] Logic Programming: a Computing Tool for the Architect of the Future Proc. of the Logic Programming Workshop
- SWINSON, P. S. G.; PEREIRA, F. C. N.; BIJL, A. [1983a] A Fact Dependency System for the Logic Programmer Computer-Aided Design Journal, Vol. 15, no. 4
- SWINSON, P. S. G. [1983b] Prolog: a Prelude to a New Generation of CAAD Computer-Aided Design Journal, Vol. 15, no. 6
- SZEREDI, P. [1977] Prolog - a Very High Level Language Based on Predicate Logic Proceedings of the Second Hungarian Computer Science Conference, Budapest
- SZPAKOWICZ, S. [1981] Papers in Logic Programming IIInfUW Reports, Institute of Informatics, Univ. of Warsaw
- TARNLUND, S. [1975] Logic Information Processing Report TRITA-IBADB - 1034 Dept. of Inf. Processing Univ. of Stockholm
- TARNLUND, S. [1976a] A Logical Basis for Data Bases Report TRITA - IBADB no. 1029 DCS - Royal Institute of Technology Stockholm in Logic and Data Bases - Workshop Toulouse
- TARNLUND, S. [1976b] Programming as a Deductive Method Report TRITA - IBADB - 1031 Dept. of Inf. Processing Univ. of Stockholm
- TARNLUND, S. [1977] Horn Clause Computability BIT 17 (1977) 215-226
- TARNLUND, S. [1978] An Axiomatic Data Base Theory Dept of Inf. Processing Univ. of Stockholm
- TAYLOR, J.; DU BOULAY, B. [1986] Studying Novice Programmers: Why They May Find Learning Prolog Hard in Rutkowska, J. and Crosk, C. (eds.), The Computer and Human Development
- TOWNSHEND, H. [1979] PRAMS Scheme and Prolog Computer Program DAI Draft Univ. of Edinburgh
- WALKER, A.; PORTO, A. [1983] KB01: A Knowledge Based Garden Store Assistant Proceed. of the Logic Programming Workshop
- WALKER, A.; McCORD, M.; SOWA, J.; WILSON, W. [1987] Knowledge Systems and Prolog: a Logical Approach to Expert Systems and Natural Language Processing Addison-Wesley
- WARREN, D. H. D. [1974a] WARPLAN - a System for Generating Plans DAI, Memo 76 Univ. of Edinburgh
- WARREN, D. H. D. [1974b] Prolog notes Notes from Lectures in November 74 (unpublished)
- WARREN, D. H. D. [1975a] A User's Guide to PROLOG Supervisor SVW (Marseille Version) DAI (unpublished) Univ. of Edinburgh

- WARREN, D. H. D. [1975b] Examples for WARPLAN and PROLOG: the Car Assembly Problem and Two Scene Analysis Problem (unpublished)
- WARREN, D. H. D. [1976a] Generating Conditional Plans and Programs AISB
- WARREN, D. H. D. [1976b] Machine Code Synthesis (SRC proposal) DAI (unpublished) Univ. of Edinburgh
- WARREN, D. H. D. [1977a] Implementing PROLOG - Compiling Predicate Logic Programs Vols. 1 & 2 - DAI, Research Report no. 39 Univ. of Edinburgh
- WARREN, D. H. D. [1977b] Logic Programming and Compiler writing DAI, Report no. 44 Univ. of Edinburgh
- WARREN, D.; PEREIRA, L. M.; PEREIRA, F. [1977c] PROLOG - The Language and Its Implementation Compared with LISP LNEC and SIGART/SIGPLAN Symposium, August 1977 Rochester
- WARREN, D. H. D. [1979] Prolog on the DECsystem-10 DAI Research Paper no. 127 Univ. of Edinburgh
- WARREN, D. H. D. [1981a] Efficient Processing of Interactive Relational Database Queries Expressed in Logic Univ. of Edinburgh
- WARREN, D. H. D. [1981b] The 'S-P Problem' Revisited Logic Programming Newsletter, 2
- WARREN, D. H. D. [1981c] Higher-Order Extensions to Prolog - Are They Needed? DAI Research Paper no. 154, Univ. of Edinburgh
- WARREN, D. H. D. [1983a] Applied Logic - Its Use and Implementation as a Programming Tool Ph.D. Thesis (Univ. of Edinburgh, 1977) Technical Note 290, SRI International
- WARREN, D. H. D. [1983b] An Abstract Prolog Instruction Set Technical Note 300, SRI International
- WELSH, J.; ELDER, J. [1979] Introduction to Pascal Prentice-Hall