tanulmányok 207/1988

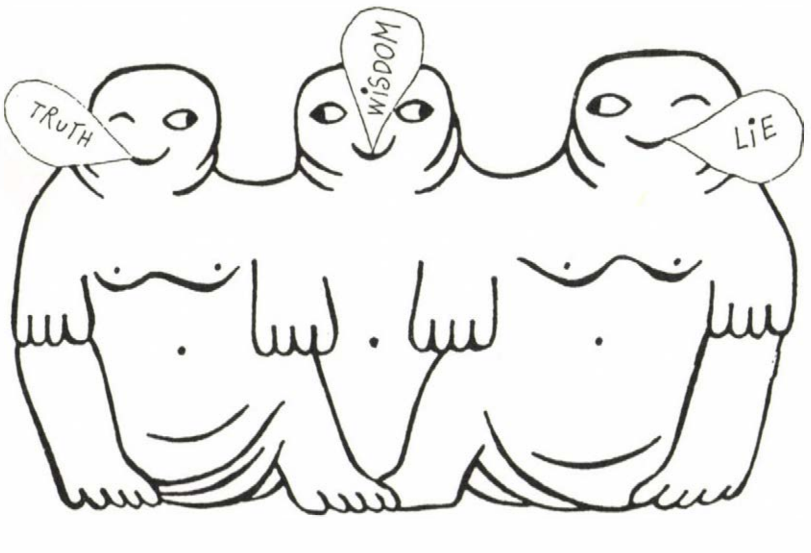
MTA Számitástechnikai és Automatizálási Kutató Intézet Budapest

Magyar Tudományos Akadémia Számitástechnikai és Automatizálási Kutató Intézete
Computer and Automation Institute Hungarian Academy of Sciences

LOGIC PUZZLES AND LOGIC PROGRAMMING I

Zsuzsa MÁRKUSZ

Gábor MÁRKUS



Tanulmányok 207/1988
Studies 207/1988

A kiadásért felelős:

KEVICZKY LÁSZLÓ

Főosztályvezető:

CSABA LÁSZLÓ

Illustrations by

Zsuzsa STUIBER

ISBN 963 311 249 4

ISSN 0324-2951

It's good rather than a pity if, eventually, your student feels: It's not so hard, I may as well have thought of it myself.

Laszlo Kalmar, *The Development of Mathematical Exactness from Visuality to Axiomatic Methods.*

Lecture, November 1941, Budapest

The aim of this report is to illustrate, via the solution of funny logic puzzles, some basic methods of logic programming and to point out the relation among the problem solving strategies of human reasoning, logic programming, and "conventional" algorithmic programming. Each puzzle is solved by a Prolog program, which demonstrates some useful Prolog programming techniques. Some puzzles are solved by Pascal programs as well, which provides an opportunity for comparing the techniques and strategies used in declarative logic programming with those used in algorithmic programming. Every Prolog and Pascal program in the report is original, none of them is published elsewhere. The programs are short and easily comprehensible for programmers as well as students of tertiary education.

Contents

Introduction

Funny logic puzzles have always challenged the artists and scientists of thinking: the philosophers and mathematicians; and the thorough investigation of paradoxes or seemingly paradoxical problems has often led to important results in logic. In the second half of the twentieth century, a new kind of device appeared to help humans think: the computers, which, via their mere existence, have been provoking certain challenge in a number of fields of life.

As a consequence of such challenges, besides the "conventional" methods of human reasoning, there have appeared new, computer oriented deductive techniques, such as the most recent one, logic programming. It was in the 70s when logic programming first appeared as a problem definition and problem solving method in artificial intelligence [4], [16], [24], [31], [32]. Since then it has become a useful tool in a number of fields, such as expert systems, computer aided design, and natural language understanding and translation. The best known and most widespread programming language of logic programming is Prolog, which is a subject in the curriculum of the computer science department of almost every university in the world.

Since applied logic programming combines the methods of mathematical logic and certain programming methodologies, it is quite natural to ask: *To what extent is logic programming adequate for solving logic puzzles?* We have found that question so interesting and inspiring that we have solved quite a number of puzzles and tried to find an answer to the question. This report presents a representative sample of the puzzles we studied; the complete set of puzzles is going to be compiled and presented as a subsequent report.

The logic puzzles solved in this report are collected from various sources. The texts of the puzzles are, however, often tailored to suit to the subject. Puzzles, such as the 8 Queens Problem, the family of cryptarithmetic puzzles of the type SEND+MORE=MONEY, the Problem of the Tower of Hanoi, etc., whose solutions via Prolog programs have already been published ([7], [8], [9], [30]) are deliberately left out. Every Prolog and Pascal program in the report is original, none of them is published elsewhere.

Each puzzle in this report is solved by a Prolog program, which demonstrates some useful Prolog programming techniques. Some puzzles are solved by Pascal programs as well, which provides an opportunity for comparing the techniques and strategies used in declarative logic programming with thos used in "conventional" algorithmic programming. As is expected, each puzzle can be solved by human reasoning without the help of any computer. Those solutions are also presented; and the differences between the typical problem solving approaches and styles of a "pure mathematician" and those of a "programmer" are discussed. In the case of certain puzzles, some exercises without solutions are also presented to complete the discussions. They are to highlight some mathematical or programming details or alternative approaches or techniques.

The texts of some Pascal programs are much longer than those of the Prolog programs for solving the same puzzles, which shows the compactness and expressive power of Prolog programs. Many Pascal programs are, on the other hand, much more efficient than the corresponding Prolog programs.

Though no special or higher knowledge is prerequisite to the use or understanding of this report, the reader is assumed to have read some Prolog and some Pascal book (such as [7], [8], [13], or [25]) and thus be familiar with the basics of those languages. Neither the solutions of the puzzles via reasoning require mathematical knowledge exceeding highschool mathematics.

The Prolog programs presented are written in the standard DEC-10 Prolog syntax and run in an MPROLOG environment on an IBM PC XT compatible personal computer, VARYTER-XT (640 KByte). The Pascal programs are written in Turbo Pascal and run on the same computer.

We really enjoyed solving logic puzzles by logic programming, it was so natural and easy and such a fun. We comletetly agree with Mr. Jacques Arsac [2], who, parapharasing the French proverb

Those who saw wood, warm twice,

said:

Those who write programs to solve puzzles, enjoy it twice.

Acknowledgements

The authors are grateful to friends and colleagues for the helpful and inspiring discussions we had about the subject. We are especially indebted to György Pollák, who provided us with the mathematical solution to one of the puzzles, and Judit Laufer, Éva Május, and Péter Somogyi, who made many useful comments on our work. We also acknowledge Jean-François Pommaret, Csudinka Csudutov, and Judit Laufer for their help in collecting the puzzles.

S. Markusz & G. Markus
Budapest, June 1988

THE CASE OF A JEALOUS BOYFRIEND

1

Kate and Mike were going to get married. They met on Friday afternoon, when Mike told Kate reproachfully he had tried to ring her up on Monday, Tuesday, Wednesday, and Thursday afternoon, but he could never find her at home.

"I have to devote some time to my friends," Kate said. "I've got only four of them, Olivia, Pat, Rose, and Sam. I spent an afternoon with each of them. I was at the hairdresser's with one of them, with another one, I went to the tailor's to have my skirt taken in, I ran into the third in the Library, and had a bit of rowing along the River with the fourth. Anyway, it's none of your business. Mind your business and leave me alone."

Mike was hurt a bit, and he felt something suspicious. He began to think over the argument:

(1) On the first three days of the week, he was by the River when he tried to give Kate a ring. Each of these days, Sam spent the whole afternoon at the Riverbank, too.

(2) Pat and Rose like each other. When they were talking this morning, they mentioned they had not been able to get to the hairdresser's for at least a week.

(3) To tell the truth, Olivia and Mike saw a film in a cinema together on Tuesday afternoon. Then she told him that originally she had been to go to the tailor's, but the tailor, who worked for both her and Kate, had left earlier that day.

(4) Kate's hairdresser works in the morning on Thursday, Friday, and Saturday. For Kate works in the mornings, she couldn't be at hers during the second half of the week.

(5) Pat or Rose never goes to the Library.

(6) The Library is closed on Thursdays.

Did Kate tell her boyfriend lies?

1.1 Solution

To figure out if Kate told a lie, Mike has to find a contradiction in Kate's argument, in which case she did tell a lie; if he cannot find any contradiction, then he should conclude that his girlfriend told the truth (or some uncontradictory lies).

Kate's argument states a one-to-one correspondence among the elements of the sets {Monday afternoon, Tuesday afternoon, Wednesday afternoon, Thursday afternoon}, {Olivia, Pat, Rose, Sam}, and {Hairdresser's, Library, Riverbank, Tailor's}. Therefore, in order to show that Kate told a lie, it is sufficient to find a friend of Kate's, for example, to which there is no suitable (afternoon, place) pair.

When Mike considers each piece of his information in turn, he can produce the following tables, where a table entry contains an X if a piece of information excludes the corresponding (friend, place) pair in that afternoon, otherwise the table entry is empty.

(The following abbreviations are used in the tables: Mon = Monday afternoon, Tue = Tuesday afternoon, Wed = Wednesday afternoon, Thu = Thursday afternoon; Oli = Olivia; Hair = Hairdresser's, Lib = Library, Riv = Riverbank, Tail = Tailor's.)

Mon	Hair	Lib	Riv	Tail
Oli				
Pat	X	X		
Rose	X	X		
Sam	X	X	X	X

Tue	Hair	Lib	Riv	Tail
Oli	X	X	X	X
Pat	X	X		X
Rose	X	X		X
Sam	X	X	X	X

Wed	Hair	Lib	Riv	Tail
Oli				
Pat	X	X		
Rose	X	X		
Sam	X	X	X	X

Thu	Hair	Lib	Riv	Tail
Oli	X	X		
Pat	X	X		
Rose	X	X		
Sam	X	X		

From the above tables it is obvious that there are only two possible cases for Tuesday afternoon:

i) Kate was at the Riverbank with Pat on Tuesday afternoon.
ii) Kate was at the Riverbank with Rose on Tuesday afternoon.

First, Mike supposes that **Kate was at the Riverbank with Pat on Tuesday afternoon** (Case i)). This implies that no one else could be at the Riverbank with Kate in any other afternoon and that Kate could not be at any other place with Pat in any other afternoon. Thus, Mike has the following tables, where a + sign denotes Mike's assumption, and - signs are placed into the entries that are excluded by the assumption.

Mon	Hair	Lib	Riv	Tail
Oli			-	
Pat	X	X	-	-
Rose	X	X	-	
Sam	X	X	X	X

Tue	Hair	Lib	Riv	Tail
Oli	X	X	X	X
Pat	X	X	+	X
Rose	X	X	-	X
Sam	X	X	X	X

Wed	Hair	Lib	Riv	Tail
Oli			-	
Pat	X	X	-	-
Rose	X	X	-	
Sam	X	X	X	X

Thu	Hair	Lib	Riv	Tail
Oli	X	X	-	
Pat	X	X	-	-
Rose	X	X	-	
Sam	X	X	-	

As for Sam, these tables show only one possibility: **Kate was at the tailor's with Sam on Thursday afternoon.** And this implies that no one else could be at the tailors's with Kate in any other afternoon. Having updated the tables again, Mike realizes that there is no afternoon remained for Rose to be with Kate, that is, **Kate could not be at any place with Rose in any afternoon.**

Following exaclty the same track, Mike can arrive at an analoguous conclusion in Case ii): If he supposes that **Kate was at the Riverbank with Rose on Tuesday afternoon**, then he concludes that **Kate could not be at any place with Pat in any afternoon.**

The conclusions in the two cases together mean that **Kate**'s argument is contradictory; consequently, she **did tell her boyfriend lies.**

Remark: If we assume that once one is at the Riverbank, he does notice anyone else who is at the Riverbank, too, or is rowing along the bank (which is not unreasonable at all if a particular spot is understood by Riverbank), then it is much easier to find a contradiction in Kate's argument. In fact, in that case there is no friend of Kate's who could be together with her on Tuesday afternoon.

1.2 Prolog program

There is no doubt, Kate is a rather able girl, but she has overlooked an important fact: her boyfriend, Mike, can program in Prolog, so he can easily check the consistency of her argument. After the sharp conversation, the jealous boyfriend jumpes up, goes home, sit down at his personal computer, and writes a Prolog program. A PC is fair, it has no sentiments, it is thus wise to ask its "opinion" about the case. First, Mike records some data, the relevant days and places and Kate's friends, as Prolog facts. Then he lists the impossible meetings, that is, the day-friend-place triplets that are excluded by his infomation. The clauses in definition **impossible** correspond to the constraints in the puzzle in almost a one-to-one manner.

The Prolog program easily generates all the meetings not excluded by definition **impossible**; those are the possible meetings. Mike's task is easy now: he should find four possible meetings, one for each day, one for each place, and one for each friend of Kate's. If he can find such meetings, then Kate's argument is consistent: she might have told him the truth. If, on the other hand, he cannot find such meetings, Kate told him lies for sure.

By performing that check for Mike, the program undoubtfully proves that Kate has told Mike lies. All in all, Mike has a fantastic luck: this simple program has prevented him from marrying a girl who is not sincere even before the wedding.

```
%              The Case of a Jealous Boyfriend

dynamic(meeting/3).

start:-
   environment,
   possible, out, check,
   nl, write("There is no contradiction; "),
   write("Kate may have told her boyfriend the truth."), nl,
   retractall(meeting(_, _, _)).
start:-
   nl, write("There is a contradiction; "),
   write("Kate told her boyfriend lies."), nl,
   retractall(meeting(_, _, _)).

environment:- set_state(evaluation_limit, 50000).

possible:-
   nl, write("Mike's information says that Kate could be"),
   nl, nl,
   day(DAY), friend(PARTNER), place(PLACE),
   not impossible(DAY, PARTNER, PLACE),
   write("  with "), write(PARTNER), write(" at the "),
   write(PLACE), write(" on "), write(DAY), nl,
   assert(meeting(DAY, PARTNER, PLACE)), fail.
possible.

out:-
   nl,
   write("Now it is to check if the above list contradicts "),
   write("Kate's argument."), nl.

check:-
   meeting(monday,X1,Y1),    meeting(tuesday,X2,Y2),
   meeting(wednesday,X3,Y3), meeting(thursday,X4,Y4),
   consistent(X1,X2,X3,X4),
   consistent(Y1,Y2,Y3,Y4).

consistent(X1,X2,X3,X4):-
   X4 =/= X3, X4 =/= X2, X4 =/= X1,
   X3 =/= X2, X3 =/= X1, X2 =/= X1.

friend(olivia).
friend(pat).
friend(rose).
friend(sam).

day(monday).
day(tuesday).
day(wednesday).
day(thursday).
```

```
place(hairdressers).
place(tailors).
place(library).
place(riverbank).

impossible(DAY, sam, ANYPLACE):- first_half_of_week(DAY).
                                                      /* 1 */
impossible(DAY, pat, hairdressers).                   /* 2 */
impossible(DAY, rose, hairdressers).                  /* 2 */
impossible(tuesday, olivia, ANYPLACE).                /* 3 */
impossible(tuesday, ANYFRIEND, tailors).              /* 3 */
impossible(DAY, ANYFRIEND, hairdressers):-
                        second_half_of_week(DAY).     /* 4 */
impossible(DAY, pat, library).                        /* 5 */
impossible(DAY, rose, library).                       /* 5 */
impossible(thursday, ANYBODY, library).               /* 6 */

first_half_of_week(monday).
first_half_of_week(tuesday).
first_half_of_week(wednesday).

second_half_of_week(thursday).
```

----------- **output** -----------

```
? start.

Mike's information says that Kate could be

   with olivia at the hairdressers on monday
   with olivia at the tailors on monday
   with olivia at the library on monday
   with olivia at the riverbank on monday
   with pat at the tailors on monday
   with pat at the riverbank on monday
   with rose at the tailors on monday
   with rose at the riverbank on monday
   with pat at the riverbank on tuesday
   with rose at the riverbank on tuesday
   with olivia at the hairdressers on wednesday
   with olivia at the tailors on wednesday
   with olivia at the library on wednesday
   with olivia at the riverbank on wednesday
   with pat at the tailors on wednesday
   with pat at the riverbank on wednesday
   with rose at the tailors on wednesday
   with rose at the riverbank on wednesday
   with olivia at the tailors on thursday
   with olivia at the riverbank on thursday
   with pat at the tailors on thursday
   with pat at the riverbank on thursday
   with rose at the tailors on thursday
   with rose at the riverbank on thursday
   with sam at the tailors on thursday
   with sam at the riverbank on thursday
```

```
Now it is to check if the above list contradicts Kate's argument.

There is a contradiction; Kate told her boyfriend lies.
Yes
```

A closer look into the program

The program is so simple and transparent, it needs hardly any explanation. Having set the environment, the program generates and displays every possible meeting. Then it tries to find four possible meetings, one for each friend, one for each place, and one for each day. The actual generation and diplay is peformed by calling predicate **possible**. On generating the possible meetings, predicate possible utilizes the inherent unify-and-backtrack mechanism of Prolog: First, a particular day (**day(DAY)**), a particular friend (**friend(PARTNER)**), and a particular place (**place(PLACE)**) are chosen. Then an attempt is made to prove that that particular triplet is impossible. If it fails to be impossible, then it is assumed to be possible (negation as failure), and it is displayed and recorded as a dynamic clause: **assert(meeting(DAY, PARTNER, PLACE))**; finally, predicate **fail** forces backtracking. If the triplet being investigated is impossible, then backtracking commences at that stage: the program tries out another particular place, if any, and the process goes on as usual. Notice that, eventually, when it exhaustively investigated all possibilities, the first clause of definition **possible** fails in finding another day beyond the last. At this point control goes on to the second clause of **possible**, which, being always true, turns failure into success. Such techniques are often used in the programs of this report.

Predicate **check** tries to find four required meetigs among the possible ones. It takes four possible meetings, one for each day, first and then checks if they are allowed or consistent. Via backtracking, it checks all candidate sets of required meetings until the first set is found, when and only when, it succeeds. In consitency check, built-in predicate =/= is used. It exactly means *not equal* if equal and not are defined as follows:

```
equal(X, X).

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

equal(X, Y) yields *true* if and only if *X* and *Y* are unifiable, and it actually performs unification if either *X* or *Y* is an uninstantiated variable. Built-in predicate = corresponds to **equal**, while built-in predicate =/= corresponds to **not equal**.

Unfortunately, the concepts of equality and negation are not so easy as one wishes they were. We will discuss some points concerning them later in the report; and for more details and thorough discussion, we refer the interested reader to [24] and [28].

The program is an excellent example for transparency: its structure directly follows Mike's thoughts. Let's have just one example.

> *Kate's hairdresser works in the morning on Thursday, Friday, and Saturday. For Kate works in the mornings, she couldn't be at hers during the second half of the week.*

This piece of information translates into the following two clauses:

```
impossible(DAY, ANYFRIEND, hairdressers) :-
        second_half_of_week(DAY).

second_half_of_week(thursday).
```

Notice that both *DAY* and *ANYFRIEND* are variables. Since *ANYFRIEND* is an unconstrained variable, it actually means any friend. *DAY* is not unconstrained, it actually means any day of the second half of the week only.

The above example also presents another issue, the problem of database consistency. Obviously, the second half of the week consist of more than one day. Therefore, we should rather have a three-clause definition

```
second_half_of_week(thursday).
second_half_of_week(friday).
second_half_of_week(saturday).
```

shouldn't we? No, we should not, or, rather, must not. Although there is no problem as far as the solution of the puzzle is concerned: the result would remain the same with the latter definition of the **second_half_of_week** and the extra computation and storage required by the two extra clauses, defining irrelevant data, is negligible (note, however, that such extra computation may not be negligible in other cases). The real problem is that those extra clauses make the database of the program inconsistent. To show this, it is enough to ask questions about the second half of the week, which now should consist of three days, Thursday, Friday, and Saturday. Obviously, the following two goals should succeed:

```
? second_half_of_week(friday).

? day(friday), second_half_of_week(friday).
```

But the first one succeeds, while the second one fails. To overcome this inconsistency, we should add two extra clauses:

```
day(friday).
day(saturday).
```

Now the database of the program is consistent, the complete database, however, defines another set of constraints, a less restrictive one, and therefore another puzzle. And thus it is only a coincidence that the final output of the completed program remains the same: "There is a contradiction."

Built-in predicates used in the program

There are many Prolog dialects all over the world with a lot of common predicates implemented under diferent names. We would like to help the reader adapt the programs in this report to his own implementation; that is why we list the built-in predicates used in the Prolog programs.

In the Prolog program for *The Case of a Jealous Boyfriend*, we used the following built-in predicates:

nl, write, retractall, not, assert, fail, =/=, set_state

Built-in predicate **set_state** occurs in a number of programs, defining a reasonable call limit for those programs. In the MPROLOG environment, the default value is 10,000, which should be increased in some cases. For doing so, we always use a separate predicate **environment**, such as

```
environment:- set_state(evaluation_limit, 50000).
```

which assigns 50,000 to environment parameter **evaluation_limit** (see also Appendix A).

Reasoning versus Prolog programming

If one wants to solve the puzzle via reasoning, without the help of a computer, the strategy implemented in the program is not really adequate: there are too many constraints in the puzzle for a human being to cope with. In such cases one should look for ways of transforming the problem into a (sequence of) simpler problem(s), in order to achieve success faster.

If, for instance, there is a day with a lot of restrictions, such as Tuesday, then it is worth investigating the question: Is it possible for Kate to meet any of her friends on that particular day at all? If it is impossible, we have solved the original problem. If it is possible, then it is most likely that there are only a few possible meetings (in fact, only two meetings are possible on Thuesday: with Pat or Rose at the Riverbank), which helps one reduce the number of possible meetings on other days, too. It is clear from the constraints that Olivia is the one about whose whereabouts Mike has the least information; therefore, one had better examine Kate's possible meetings with the other girls first. Having a closer look into Pat's, Rose's, and Sam's time schedules, one soon realizes that it is impossible to arrange the three girls to be at three different places on three different days. The above is an instance of a general problem solving strategy: one tries to reduce the search space as soon as possible by examining the conditions of the given problem, and concentrating on the most promising subproblem. Intuition plays an important role in the solution of problems via reasoning.

The above problem solving strategy is none the less adequate for solving problems via programs. But, due to some nonhuman features of computers, the selection and handling of subproblems may be a bit relaxed: the search space may be much larger, the algorithm may be less sophisticated, it may contain more mechanical segments. Notice the instances of these points in the bodies of clauses **possible**, **check**, and **consistent**. Although, of course, a computer too has limitations (see Section 8), it can be a useful aid for humans in solving various problems, and in solving logic puzzles in particular. And since Prolog programs can follow human thoughts fairly closely, they seem to be rather effective problem solving aids.

Exercises

E1.1 Change the body of clause **check** so that no meeting be chosen in vain, i.e., check consistency as soon as possible (after the second, third, and fourth predicate **meeting**); change predicate **consistent** accordingly. The new version seems to be more efficient. Is it really more efficient? At what cost?

E1.2 Having understood the solution of the puzzle, one can see how the order of database clauses affect the performance of the program. Change the order of clauses in definitions **place**, **friend**, and **day** to speed up the program.
(Notice that it is nothing but reducing the search space.)

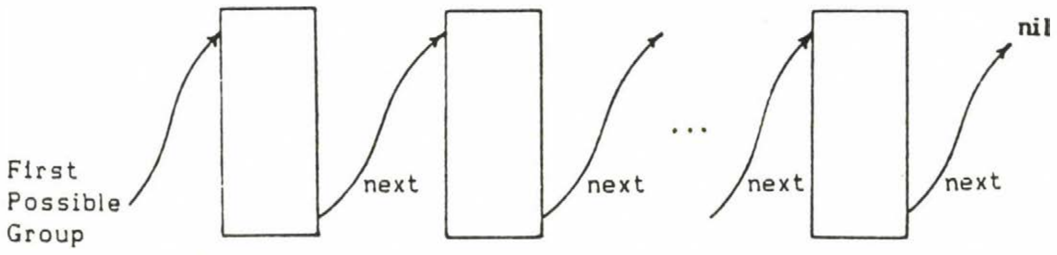
1.3 Pascal program

On the surface, the problem solving strategy of the Pascal program is essentially the same as that of the Prolog program: for it is most adequate, the Pascal program simulates the choosing-backtracking strategy of Prolog--there are only minor differences (see the exercises at the end of this section). Although the underlying algorithm is virtually identical, the organization of the Pascal program differs from that of the Prolog program. The most obvious difference is that the Pascal program requests the user to enter the relevant data items, if they are not supplied. Then, using those data, it generates a list of impossible combinations. In contrast to this, the Prolog program contains the impossible combinations, as well as the other relevant constants, declaratively (definitions **impossible**, **friend**, and **day**) or inline (names *Kate* and *Mike*).

The Prolog program then generates all possible combinations and then tries to find a different combination for each day among them. The Pascal program, on the other hand, tries to find a possible combination for each day in turn, without having generated the set of possible combinations.

A closer look into the program

First, the Pascal program sets the initial state of the solution of the puzzle via **procedure Initialize**. **procedure Initialize** checks if input data are supplied in a text file (**function Exist**). If there exists a relevant data file, it reads the data from that file (**procedure GetDataFromFile**). Otherwise it requests the user to enter the relevant data, echoes and checks the data read and stores them in a data file in order to save the user's effort of inputting them when he reruns the program (**procedure GetDataFromKeyboard** and **procedure GetOneItem**). On accepting the data items in either way, the program records them and generates the chained list of impossible combinations (**procedures AddFriendEtc**, **AddPlaceEtc**, and **AddDay**).



The data to be entered in order to solve the puzzle are shown below along the trace of the man-machine communication (the user answers are underlined for emphasis).

```
Please enter the name of the girl.
> Kate
Kate
Please enter the name of her boyfriend.
> Mike
Mike
Please enter the names of the days.
> Monday
Monday
> Tuesday
Tuesday
> Wednesday
Wednesday
> Thursday
Thursday
Please enter the names of the friends.
> Olivia
Olivia
> Pat
Pat
> Rose
Rose
> Sam
Sam
Please enter the names of the places.
> Hairdressers
Hairdressers
> Library
Library
> Tailors
Tailors
> Riverbank
Riverbank

Please enter the impossible groups [friend, place of action, day].
Allowed answers:
   Friends: Sam  Rose  Pat  Olivia  AnyFriend
   Places of Actions: Riverbank  Tailors  Library  Hairdressers  AnyPlace
   Days: Thursday  Wednesday  Tuesday  Monday  AnyDay
Friend: Sam
Place of Action: AnyPlace
Day: Monday

Is there any more impossible group? (y/n): y

Friend: Sam
Place of Action: Anyplace ?  > AnyPlace
Day: Tuesday
```

```
Is there any more impossible group? (y/n): y

Friend: Sam
Place of Action: AnyPlace
Day: Wednesday

Is there any more impossible group? (y/n): y

Friend: Pat
Place of Action: Hairdressers
Day: Anyday

Is there any more impossible group? (y/n): y

Friend: Rose
Place of Action: Hairdressers
Day: Anyday

Is there any more impossible group? (y/n): y

Friend: Olivia
Place of Action: AnyPlace
Day: Tuesday

Is there any more impossible group? (y/n): y

Friend: AnyFriend
Place of Action: Tailors
Day: Tuesday

Is there any more impossible group? (y/n): y

Friend: AnyFriend
Place of Action: Hairdressers
Day: Thursday

Is there any more impossible group? (y/n): y

Friend: Pat
Place of Action: Library
Day: AnyDay

Is there any more impossible group? (y/n): y

Friend: Rose
Place of Action: Library
Day: AnyDay

Is there any more impossible group? (y/n): y

Friend: AnyFriend
Place of Action: Library
Day: Thursday

Is there any more impossible group? (y/n): n
```

----------- **output** -----------

```
Kate has told Mike lies.
```

After the initialization, the main program starts to generate and test the candidate combinations for each day. A possible combination or group consists of a day (at the top level), a friend (at the next level), and a place [of action] (at the bottom level). Values are chosen in the order given at input according to the depth-first search strategy. A new candidate combination is generated in the loop of the main program; initial values for the two upper level components are also chosen on the spot. **procedure GenerateGroups** instantiates the bottom level component and checks if the candidate is impossible. The procedure "generates" new candidates only, i.e., it examines candidates not yet investigated. First it calls **procedure SetCurrentCounters**, which sets the current values of the backtrack pointers of the two lower level components, and then finds values of lower level components to obtain new candidates (**procedures ChooseAnotherPlace and ChooseAnotherFriend**) utilizing the required one-to-one correspondence among the values of components at different levels. Whenever a candidate is found, it is checked against the impossible groups (**procedure CheckCandidate**). If it proves to be impossible, the algorithm backtracks: it tries to find a new place, and if there is no more place to be chosen, it tries to find a new friend. The backtracking at the two lower levels is implemented by the loop in **procedure GenerateGroups**. The backtracking at the top level is of another sort: the candidate group, which has proved to be the outcome of a wrong guess, has to be deleted and backtracking has to be continued at the bottom level of the previous candidate, if any. This action is performed and controlled by **procedure WrongGuess**. The program, that is, the loop in the main program, stops as soon as a complete set of possible combinations is found or when all combinations proved to be impossible.

```
program Jealous (input, output, lst, Data, Fil);
  const
    ItemNo = 4; ItemNoPlusOne = 5; WordLength = 20;
    FileName = 'JEALOUS.DTA';
  type
    Word = string[WordLength];
    Words = array [0..ItemNo] of Word;
    GroupPtr = ^Groups;
    Groups = record Friend : Word;
                    PlaceOfAction : Word;
                    Day : Word;
                    next : GroupPtr
             end;
  DataType = text;
  ExtItems = 0..ItemNoPlusOne;
var
  Girl, Boy: Word;
  Days, Friends, PlacesOfActions: Words;
  FirstImpossibleGroup, Impossible: GroupPtr;
  FirstPossibleGroup, Possible: GroupPtr;
  DayCount, FriendCount: ExtItems;
  FriendChosen: Boolean;
  Data: DataType;

procedure InitializeProblem (var Girl, Boy: Word;
           var Days, Friends, PlacesOfActions: Words;
           var FirstImpossibleGroup, Impossible: GroupPtr;
           var Data: DataType);
 { initialize the constraints of the puzzle }
  type
    Name = string[30];
  var
    i: 1..ItemNoPlusOne;
    ItemGotten, A_Day, A_Friend, A_Place: Word;

  procedure GetOneItem (var Items: Words; var Item: Word);
   { accept a data item }
  var
    j: -1..ItemNoPlusOne;
    OK: Boolean;
  begin  { GetOneItem }
    OK := false;
    while not OK do
    begin
      read(ItemGotten);
      j := -1;
      repeat  j := j + 1
      until (Items[j] = ItemGotten) or (j = ItemNo+1);
      if j <= ItemNo then OK := true
                     else write(' ?  > ')
    end;
    writeln;
    Item := ItemGotten
  end;  { GetOneItem }
```

```
procedure AddFriendEtc (var FirstImpossibleGroup,
                            Impossible: GroupPtr);
  { generate impossible groups with special respect to
                                              field A_Day }

  procedure AddPlaceEtc (var FirstImpossibleGroup,
                             Impossible: GroupPtr);
  { generate impossible groups with special respect to
                                            field A_Place }

    procedure AddDay (var FirstImpossibleGroup,
                          Impossible: GroupPtr);
      { actually generate the impossible groups }
      begin  { AddDay }
        new(Impossible);
        with Impossible^ do
        begin
          Friend := A_Friend;
          PlaceOfAction := A_Place;
          Day := A_Day;
          next := FirstImpossibleGroup
        end;
        FirstImpossibleGroup := Impossible
      end;  { AddDay }

    begin  { AddPlaceEtc }
      if A_Day = 'AnyDay' then
        for i := 1 to ItemNo do
        begin
          A_Day := Days[i];
          AddDay(FirstImpossibleGroup, Impossible)
        end
      else AddDay(FirstImpossibleGroup, Impossible)
    end;  { AddPlaceEtc }

  begin  { AddFriendEtc }
    if A_Place = 'AnyPlace' then
      for i := 1 to ItemNo do
      begin
        A_Place := PlacesOfActions[i];
        AddPlaceEtc(FirstImpossibleGroup, Impossible)
      end
    else AddPlaceEtc(FirstImpossibleGroup, Impossible)
  end;   { AddFriendEtc }

procedure GetDataFromKeyboard (var Girl, Boy: Word;
         var FirstImpossibleGroup, Impossible: GroupPtr;
         var Data: DataType);
  { accept data from keyboard, echo and store the items
    gotten and generate impossible groups }
  var
    more: char;
    continue: Boolean;
```

```
begin  { GetDataFromKeyboard }
  writeln('Please enter the name of the girl.');
  write('> ');  readln(Girl);  writeln(Girl);
  writeln(Data, Girl);
  writeln('Please enter the name of her boyfriend.');
  write('> ');  readln(Boy);  writeln(Boy);
  writeln(Data, Boy);
  writeln('Please enter the names of the days.');
  for i := 1 to ItemNo do
    begin
    write('> '); readln(ItemGotten); writeln(ItemGotten);
    Days[i] := ItemGotten; writeln(Data, ItemGotten)
    end;
  writeln('Please enter the names of the friends.');
  for i := 1 to ItemNo do
    begin
    write('> '); readln(ItemGotten); writeln(ItemGotten);
    Friends[i] := ItemGotten; writeln(Data, ItemGotten)
    end;
  writeln('Please enter the names of the places.');
  for i := 1 to ItemNo do
    begin
    write('> '); readln(ItemGotten); writeln(ItemGotten);
    PlacesOfActions[i] := ItemGotten;
    writeln(Data, ItemGotten)
    end;
  FirstImpossibleGroup := nil;
  writeln;
  write('Please enter the impossible groups ');
  writeln('[friend, place of action, day].');
  writeln('Allowed answers:');
  write('    Friends: ');
  for i := ItemNo downto 0 do write(Friends[i], '  ');
  writeln;
  write('    Places of Actions: ');
  for i := ItemNo downto 0 do
    write(PlacesOfActions[i], '  ');
  writeln;
  write('    Days: ');
  for i := ItemNo downto 0 do write(Days[i], '  ');
  writeln;
  continue := true;
  while continue do
  begin
    write('Friend: ');
    GetOneItem(Friends, A_Friend);
    writeln(Data, A_Friend);
    write('Place of Action: ');
    GetOneItem(PlacesOfActions, A_Place);
    writeln(Data, A_Place);
    write('Day: ');
    GetOneItem(Days, A_Day);  writeln(Data, A_Day);
```

```
      if A_Friend = 'AnyFriend' then
        for i := 1 to ItemNo do
        begin
          A_Friend := Friends[i];
          AddFriendEtc(FirstImpossibleGroup, Impossible)
        end
      else AddFriendEtc(FirstImpossibleGroup, Impossible);
      more := ' ';
      writeln;
      write('Is there any more impossible group? (y/n): ');
      read(more);
      while not ( (more = 'y') or (more = 'n') ) do
      begin
        write(' ?  > '); read(more)
      end;
      writeln;  writeln;
      if more = 'n' then continue := false
    end
  end;  { GetDataFromKeyboard }

procedure GetDataFromFile (var Girl, Boy: Word;
         var FirstImpossibleGroup, Impossible: GroupPtr;
         var Data: DataType);
 { retrieve data form text file
   and generate impossible groups }
  begin  { GetDataFromFile }
    readln(Data, Girl);
    readln(Data, Boy);
    for i := 1 to ItemNo do readln(Data, Days[i]);
    for i := 1 to ItemNo do readln(Data, Friends[i]);
    for i := 1 to ItemNo do readln(Data, PlacesOfActions[i]);
    FirstImpossibleGroup := nil;
    while not eof(Data) do
    begin
      readln(Data, A_Friend);
      readln(Data, A_Place);
      readln(Data, A_Day);
      if A_Friend = 'AnyFriend' then
        for i := 1 to ItemNo do
        begin
          A_Friend := Friends[i];
          AddFriendEtc(FirstImpossibleGroup, Impossible)
        end
      else AddFriendEtc(FirstImpossibleGroup, Impossible);
    end
  end;  { GetDataFromFile }
```

```
function Exist (Filename: Name): Boolean;
 { check if a file exists }
  var
    Fil: file;
  begin  { Exist }
    assign(Fil, Filename);
    {$I-}
    reset(Fil);
    {$I+}
    if IOresult <> 0 then Exist := false
                     else Exist := true
  end;  { Exist }

begin  { InitializeProblem }
  Days[0] := 'AnyDay';
  Friends[0] := 'AnyFriend';
  PlacesOfActions[0] := 'AnyPlace';
  assign(Data, FileName);
  if Exist(FileName) then
  begin
    reset(Data);
    GetDataFromFile(Girl, Boy, FirstImpossibleGroup,
                                       Impossible, Data);
    writeln;
    writeln('     >> Data in file ', FileName,
                                         ' are read. <<')
  end
  else
  begin
    rewrite(Data);
    GetDataFromKeyboard(Girl, Boy, FirstImpossibleGroup,
                                       Impossible, Data)
  end;
  close(Data)
end;  { InitializeProblem }

procedure ChooseAnotherFriend (var FirstPossibleGroup:
                                              GroupPtr;
                               var FriendCount: ExtItems;
                               var Friends: Words;
                               var FriendChosen: Boolean);
{ by taking another friend, find a new candidate group }
  var
    OK: Boolean;
    A_Friend: Word;
    CurrentPossibleGroup: GroupPtr;
  begin  { ChooseAnotherFriend }
    CurrentPossibleGroup := FirstPossibleGroup;
    OK := false;
    while (CurrentPossibleGroup <> nil) and
          (FriendCount < ItemNo) do
```

```
      begin
        FriendCount := FriendCount + 1;
        A_Friend := Friends[FriendCount];
        CurrentPossibleGroup := FirstPossibleGroup;
        OK := true;
        repeat
          if CurrentPossibleGroup^.Friend = A_Friend then
            OK := false;
          if OK then
            CurrentPossibleGroup := CurrentPossibleGroup^.next
        until not OK or (CurrentPossibleGroup = nil)
      end;
      if OK then FriendChosen := true
            else FriendChosen := false
    end;   { ChooseAnotherFriend }

procedure GenerateGroups (var FirstPossibleGroup,
                              FirstImpossibleroup: GroupPtr;
                          var DayCount, FriendCount: ExtItems;
                          var Friends, PlacesOfActions: Words);
  { fill out the frame of a group to suit to the constraints }
  var
    Loop: (CYCLE, EXIT_OK, EXIT_BACK);
    FriendChosen, PlaceChosen, Collision, Go: Boolean;
    PlaceCount: ExtItems;
    CurrentPossibleGroup: GroupPtr;

  procedure SetCurrentCounters (var FirstPossibleGroup:
                                                 GroupPtr;
                      var FriendCount, PlaceCount: ExtItems;
                      var Friends, PlacesOfActions: Words);
      { set counters FriendCount and PlaceCount
        to point to the values in the FirstPossibleGroup }
    begin  { SetCurrentCounters }
      FriendCount := 0;
      PlaceCount := 0;
      if FirstPossibleGroup^.Friend <> Friends[0] then
        repeat  FriendCount := FriendCount + 1
        until (FriendCount = ItemNo) or
              (Friends[FriendCount] =
                                FirstPossibleGroup^.Friend);
      if FirstPossibleGroup^.PlaceOfAction <>
                                PlacesOfActions[0] then
        repeat  PlaceCount := PlaceCount + 1
        until (PlaceCount = ItemNo) or
          (PlacesOfActions[PlaceCount] =
                          FirstPossibleGroup^.PlaceOfAction)
    end;  { SetCurrentCounters }
```

```
procedure ChooseAnotherPlace (var FirstPossibleGroup:
                                          GroupPtr;
                              var PlaceCount: ExtItems;
                              var PlacesOfActions: Words;
                              var PlaceChosen: Boolean);
  { by taking another place, find a new candidate group }
  var
    OK: Boolean;
    A_Place: Word;
  begin  { ChooseAnotherPlace }
    CurrentPossibleGroup := FirstPossibleGroup;
    OK := false;
    while (CurrentPossibleGroup <> nil) and
          (PlaceCount < ItemNo) do
    begin
      PlaceCount := PlaceCount + 1;
      A_Place := PlacesOfActions[PlaceCount];
      CurrentPossibleGroup := FirstPossibleGroup;
      OK := true;
      repeat
        if CurrentPossibleGroup^.PlaceOfAction = A_Place
        then OK := false;
        if OK then
          CurrentPossibleGroup := CurrentPossibleGroup^.next
      until not OK or (CurrentPossibleGroup = nil)
    end;
    if OK then PlaceChosen := true
          else PlaceChosen := false
  end;   { ChooseAnotherPlace }

procedure WrongGuess (var FirstPossibleGroup: GroupPtr;
                      DayCount: ExtItems; var Go: Boolean);
 { delete the latest group, which proved to be wrong,
                                     and step back a day }
  var
    WrongGroup: GroupPtr;
  begin  { WrongGuess }
    if FirstPossibleGroup <> nil then
    begin
      WrongGroup := FirstPossibleGroup;
      FirstPossibleGroup := WrongGroup^.next;
      dispose(WrongGroup);
      DayCount := DayCount - 1;
      Go := true
    end
    else  Go := false
  end;  { WrongGuess }

procedure CheckCandidate (var FristPossibleGroup,
                              FirstImpossibleGroup: GroupPtr;
                          var Collision: Boolean);
      { check if a candidate group is impossible }
  var
    CurrentImpossibleGroup: GroupPtr;
  begin  { CheckCandidate }
    Collision := false;
    CurrentImpossibleGroup := FirstImpossibleGroup;
    while not Collision and
         (CurrentImpossibleGroup <> nil) do
    begin
      with CurrentImpossibleGroup^ do
        if (Day     = FirstPossibleGroup^.Day)    and
           (Friend = FirstPossibleGroup^.Friend) and
           (PlaceOfAction  =
                    FirstPossibleGroup^.PlaceOfAction)
        then  Collision := true;
      CurrentImpossibleGroup :=
                          CurrentImpossibleGroup^.next
    end
  end;  { CheckCandidate }

begin  { GenerateGroups }
  SetCurrentCounters(FirstPossibleGroup, FriendCount,
                     PlaceCount, Friends, PlacesOfActions);
  Loop := CYCLE;
  while Loop = CYCLE do
  begin
    ChooseAnotherPlace(FirstPossibleGroup, PlaceCount,
                          PlacesOfActions, PlaceChosen);
    if PlaceChosen then
    begin
      FirstPossibleGroup^.PlaceOfAction :=
                          PlacesOfActions[PlaceCount];
      CheckCandidate(FirstPossibleGroup,
                       FirstImpossibleGroup, Collision);
      if not Collision then Loop := EXIT_OK
    end
    else
    begin
      ChooseAnotherFriend(FirstPossibleGroup, FriendCount,
                                 Friends, FriendChosen);
      if FriendChosen then
      begin
        FirstPossibleGroup^.Friend := Friends[FriendCount];
        PlaceCount := 0;
        FirstPossibleGroup^.PlaceOfAction :=
                           PlacesOfActions[PlaceCount]
      end
      else  Loop := EXIT_BACK
    end
  end;
  if Loop = EXIT_BACK then
  begin
    WrongGuess(FirstPossibleGroup, DayCount, Go);
    if Go then GenerateGroups(FirstPossibleGroup,
                             FirstImpossibleGroup,
                             DayCount, FriendCount,
                             Friends, PlacesOfActions)
  end
end;  { GenerateGroups }
```

```
begin  { Jealous }
  InitializeProblem(Girl, Boy, Days, Friends, PlacesOfActions,
                    FirstImpossibleGroup, Impossible, Data);
  FirstPossibleGroup := nil;
  DayCount := 0;
  repeat  { generate and test candidate groups }
    DayCount := DayCount + 1;
    new(Possible);
    with Possible^ do
    begin
      Day := Days[DayCount];
      if FirstPossibleGroup = nil then FriendCount := 1
      else
      begin
        FriendCount := 0;
        ChooseAnotherFriend(FirstPossibleGroup, FriendCount,
                            Friends, FriendChosen);
        if not FriendChosen then
        begin
          writeln('    Error in the algorithm--main.');
          repeat until keypressed
        end
      end;
      Friend := Friends[FriendCount];
      PlaceOfAction := PlacesOfActions[0];
      next := FirstPossibleGroup
    end;
    FirstPossibleGroup := Possible;
    GenerateGroups(FirstPossibleGroup, FirstImpossibleGroup,
                   DayCount, FriendCount, Friends,
                   PlacesOfActions)
  until  (DayCount = 0) or (DayCount = ItemNo);
  writeln;
  if DayCount = 0 then
    writeln(Girl, ' has told ', Boy, ' lies.')
  else
    writeln(Girl, ' may not have told ', Boy, ' lies.')
end.   { Jealous }
```

----------- **output** -----------

```
      >> Data in file JEALOUS.DTA are read. <<

 Kate has told Mike lies.
```

Exercises

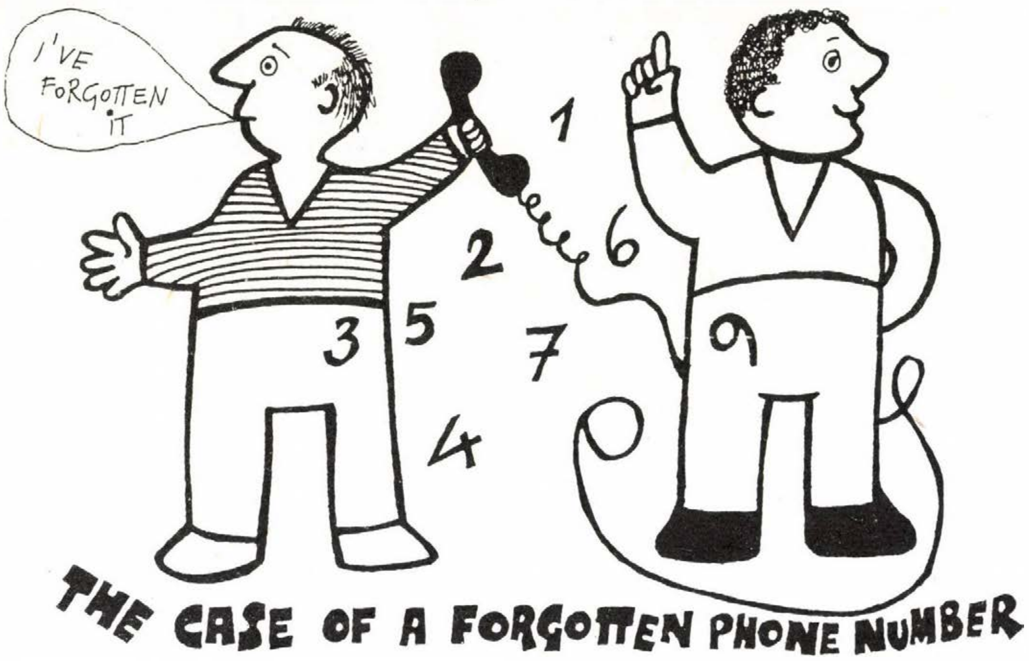
E1.3 Notice that the program utilizes some nonstandard features of Turbo Pascal, such as the *string type*, the built-in function *keypressed*, special external file handling. Rewrite the program so that it suit to your Pascal implementation.
(Hint: Although they make the use of the program convenient, the file handling fragments are not essential. *String*s are usually implemented as *packed arrays of char*acters.)

E1.4 As in the case of the Prolog program, the order of input data items (days, friends, places, *and* impossible combinations) does affect the performance of the program. Find a better input order.

E1.5 As is mentioned above, the actual algorithms programmed in Prolog and Pascal are different.
a) Rewrite the Prolog program so that it implement the algorithm of the Pascal program.
b) Rewrite the Pascal program so that it implement the algorithm of the Prolog program.
Argue for and against the versions obtained.

E1.6 The Pascal program presented tries to simulate the choosing-backtracking strategy of Prolog. The most severe restriction is the utilization of the static number of possible values (the algorithm does not allow to add or delete a friend, for instance) and the utilization of the one-to-one correspondence among the components of possible groups. Rewrite the program to get rid of the above restrictions. Is it worth making a distinction between the static (in the above sense) and dynamic (as the opposite of static) sets of clauses? Why?
(Hint: Use chained lists of records.)

E1.7 Unlike the Prolog program, the Pascal program collects the data relevant to the solution of the puzzle from the user. Rewrite the Prolog program so that it contain a similar interactive session. Given your Prolog implementation, how can you utilize external files to improve convenience?
(Hint: Use the constructor functor =.. to form clauses and/or use lists [instead of clauses]. Remember the importance of the order of clauses inside a definition.)



THE CASE OF A FORGOTTEN PHONE NUMBER

2

At a party, someone suggested that they should give Frank a ring. Unfortunately, there was no one at present who knew his phone number. All they could collect was some little bits of information:

(1) He had a six-figure number.

(2) The second half of the number, that is, the number formed by the last three figures, was equal to four times the first one.

(3) The two figures in the middle of the number were identical.

(4) The second figure was equal to twice the first.

(5) And the third figure in the phone number was two times the second one or two plus the second one.

What was Frank's phone number?

2.1 Solution

The first half as well as the second half of a six-figure number is a three-figure number. Therefore, from information (2) it follows that the first figure is not greater than 2--otherwise the second half of the number, which is four times the first half of it, has four figures.

Now, from information (4) it follows that the first two figures in the phone number can only be 00, 12, or 24.

Then from information (5) it follows that the third figure in the number is even, since the second one is even and the third one is obtained by multiplying the second one by two or incrementing it by two. Therefore, the phone number cannot start with 24. Since in that case the first half of it would be at least 240 and at most 249, and thus the second half of the number, which is four times the first half of it, would be at least 960 and at most 996, that is, the fourth figure would be 9 anyway, which, being odd, cannot be equal to the third figure (information (3)).

The phone number cannot start with 00 either, since in that case, according to information (5), the third figure would be either 0 or 2. If the third figure were also 0, then the phone number would consist of six zeros, which is not consisdered a valid phone number (though all requirements are fulfilled in that case). If the third figure were 2, then the phone number would be 002-008, which violates requirement (3).

Hence, the phone number can only start with 12, in which case the third figure is 4, by either part of information (5), and the second half of the number is 4*124=496, which satisfied requirements (3) as well.

Therefore, the only phone number that satisfies all requirements is **124-496**. That **is Frank's phone number.**

2.2 Prolog program

The people at the party tried to reconstruct Frank's phone number from various bits of information. When we start to write a program to help them, we cannot know how accurate those little bits of information are or if they are sufficient for us to determine the phone number. That being the case, we have to handle three possible cases:

- More than one phone numbers are possible.
- Exactly one phone number is possible.
- The pieces of information do not determine a phone number.

Having studied the conditions carefully, we realize that the first digit determines all other digits: they can simply be computed. Therefore, the program takes new and new values for the first digit on backtracking until the complete set of digits is exhausted, and records the different phone numbers obtained. The strategy of the solution via reasoning is more or less the same. A man, however, knows and utilizes a number of properties of integers, such as integers are either even or odd, there are simple rules for the parity of the results of arithmetic operations, the integers are sorted, etc. These properties can, of course, be incorporated into the program, but the effort, however little it is, is not worthwhile: the simple version of the program is reasonably fast. Similarly, it is easier to list the ten digits than to generate them. Note, however, that it may be crucial to program such background knowledge in other cases (c.f. Section 7).

The program does not utilize the special advantages of Prolog: it uses hardly any backtracking, does not unify complex structures, etc. Therefore, its algorithm can easily be programmed in any other language as well.

A closer look into the program

As is mentioned above, the program tries out the possible values of the first digit, *D1*, in turn. Once the first digit is selected, the second and the third ones (*D2* and *D3*) are computed using the rules in the puzzle. Then the program constructs the first half of the candidate phone number by calling the **half(D1, D2, D3, FIRST)** predicate. *FIRST=0* is not allowed, since in that case the phone number would consist of six zeros, which is not considered a valid phone number. On having an allowed value of *FIRST*, the program computes the second half (*SECOND*) of the phone number, and checks if it satisfies the requirements (**half(D3, D5, D6, SECOND)**).

The program calls predicate **half** twice. At the first call, the first three arguments of the predicate are bound to decimal digits, in which case the fourth argument is unified with the integer formed by those digits if the fourth argument is a free variable. If the fourth argument is also bound, then it is tested if that argument is unifiable with the integer formed by the three digits (the first clause in the definition of **half**). The second clause in the definition of **half** works essentially in the opposite direction: if the fourth argument is a three-digit integer, then the first three arguments are unified with or compared to its digits, depending if an argument is free or bound. When predicate **half** is called at the second time, **half(D3, D4, D5, SECOND)**, the second clause in the definition is activated and the first argument is compared to the first digit of *SECOND*, while the second and the third arguments are unified with the second and the third digits of *SECOND*, respectively.

```
%         The Case of a Forgotten Phone Number

dynamic(phone/2).

start:-
   digit(D1),
   D2 is 2*D1, digit(D2),
   ( D3 is 2*D2 ; D3 is 2+D2 ), digit(D3),
   half(D1,D2,D3, FIRST), FIRST =/= 0,
   SECOND is 4*FIRST,  SECOND < 1000,
   half(D3,D5,D6, SECOND),
   remember(FIRST, SECOND),
   fail.
start:-
   number_of_results(N),
   out(N).

half(D1,D2,D3, N):-
   digit(D1), digit(D2), digit(D3),
   Y1 is 100*D1,   Y2 is 10*D2,    N is Y1+Y2+D3.
half(D1,D2,D3, N):-
   integer(N), N > 0,
   N < 1000,
   D1 is N div 100,   Y  is N mod 100,
   D2 is Y div  10,   D3 is Y mod  10.

remember(F, S):-
   phone(F, S), !, fail.
remember(F, S):-
   assert(phone(F, S)).

number_of_results(many):-
   phone(F, S), phone(F1, S1),
   ( F =/= F1 ; S =/= S1 ), !.
number_of_results(1):-
   phone(F, S), !.
number_of_results(0).

out(many):-
   nl, write("The phone number is not unique, "),
   write("the folks have to make some trials."), nl, nl,
   write("The possible numbers are:"), nl,
   out.
out(1):-
   retract(phone(F, S)),
   nl, write("Frank's phone number is:  "),
   write(F), write("-"), write(S), nl, nl.
out(0):-
   nl,
   write("The pieces of information do not "),
   write("determine a phone number."),
   nl.
```

```
out:-
   retract(phone(F, S)),
   tab(10), write(F), write("-"), write(S),  nl,
   out.
out:- nl.

digit(0).
digit(1).
digit(2).
digit(3).
digit(4).
digit(5).
digit(6).
digit(7).
digit(8).
digit(9).
```

----------- **output** -----------

```
? start.

Frank's phone number is:  124-496

Yes
```

Once we have found a phone number, we record it, that is, we assert it as a dynamic clause **phone(F, S)**, where *F* is the integer formed by the first half of the phone number and *S* is the integer formed by the second half of it. (Note that a six-digit integer would be too big to be representable.) As we do not know how many solutions we will have, we should generate all possible phone numbers. But as we are interested only in the different solutions, we must not record duplicates. Predicate **remember** does exactly that for us: first it checks if the a phone number has already been recorded, and stores the solution found most recently if and only if it has not been recorded yet.

The output of the program depends on the number of the solutions: we have prepared different texts for each possible case. On displaying a phone number *F-S*, the program deletes the corresponding clause **phone(F, S)**. Although it seems to be unnecessary, this kind of "garbage collection" becomes important as soon as we want to re-run the program. That is why each Prolog program presented in this report deletes all dynamic clauses generated.

Symbol **;**, which denotes the permissive **or** of logic within one clause, is worth mentioning here, because it appears at several places in the program. Using this symbol properly, we can write more concise and more elegant programs. For example, the condition in the puzzle

> *And the third figure in the phone number was two times the second one or two plus the second one.*

naturally translates into the Prolog subgoal

```
        (D3 is 2*D2 ; D3 is 2+D2)
```

The effect of this subgoal could be more difficult to achieve without ;.

Built-in predicates used in the program

integer, =/=, is, *, +, <, fail, div, mod, assert, !, nl, write, tab, retract.

Exercise

E2.1 Built-in predicate **write** requires exactly one argument; it displays the value of that argument. To force a line feed and carriage return, we have to use built-in predicate **nl**. Therefore, if we have to display a number of items on several lines, several items a line, and usually we have to do so, then it is rather disappointing to use that huge amount of single-argument **write** predicates and the **nl** predicates. To overcome such problems, write definitions **write** and **writeln** which accept 0 to 6 arguments, for instance, and **writeln** performs line feed and carriage return as well. Rewrite the program using these new predicates and enjoy the convenience provided.

2.3 Pascal program

The strategy the Pascal program follows while solving the puzzle is similar to the problem solving strategy used in the mathematical reasoning and in the Prolog progam. The Pascal program **PhoneNumber** investigates each possible value of the first digit, *D1*, in turn, generates further digits (*D2, D3,* and *D4*) as well as the first half (*FirstValue*) and the second half (*SecondValue*) of the phone number, if necessary, and checks them against the constraints given in the puzzle. Whenever a phone number, that is, a pair (*FirstValue, SecondValue*), satisfying all constraints is found, it is recorded as two consecutive entries of array *Results* if and only if it is the first occurrence of that phone number (**function Duplicate**). This kind of a technique is forced by the integer representation of Turbo Pascal: *maxint* = 2^{15} = 32768. (Notice that the same problem appears in the Prolog

```
program PhoneNumber (output);
  const
    TwiceMaxNoResults = 10;
  type
    Digit = 0..9;
    ThreeDigitCard = 0..999;
    TimesOrPlus = (times, plus);
  var
    FirstValue, SecondValue: ThreeDigitCard;
    FirstTimes4: 0..9999;
    D1, D2, D3, D4: Digit;
    WhichOne: TimesOrPlus;
    TwiceNoSolutions, Count: 0..TwiceMaxNoResults;
    Temp: 0..18;
    TempReal: real;
    Results: array [1..TwiceMaxNoResults] of ThreeDigitCard;

  procedure Display3 (Number: ThreeDigitCard);
   {display an integer between 0 and 999 with leading zeros}
    begin  { Diplay3 }
      if Number > 99 then write(Number:0)
      else if Number > 9 then write('0', Number:0)
      else if Number > 0 then write('00', Number:0)
      else write('000')
    end;  { Display3 }

  function Duplicate (Number1, Number2: ThreeDigitCard):
                                                          Boolean;
     { check if a result has already been encountered }
    begin { Duplicate }
      Duplicate := false;
      Count := 1;
      while (Count < TwiceNoSolutions) and
            ( (Results[Count] <> Number1) or
              (Results[Count+1] <> Number2) )
        do Count := Count + 2;
      if Count < TwiceNoSolutions then Duplicate := true
    end;  { Duplicate }

  begin  { PhoneNumber }
    TwiceNoSolutions := 0;
    for WhichOne := times to plus do
      for D1 := 0 to 9 do
        if 2*D1 < 10 then
        begin
          D2 := 2*D1;
          if WhichOne = times then Temp := 2*D2
                              else Temp := 2+D2;
      if Temp < 10 then
      begin
        D3 := Temp;
        FirstValue := 100*D1 + 10*D2 + D3;
        FirstTimes4 := 4 * FirstValue;
        if FirstTimes4 < 1000 then
        begin
          SecondValue := FirstTimes4;
          D4 := SecondValue div 100;
          if D4 = D3 then
            if not ((FirstValue = 0) and
                    (SecondValue = 0)) then
              if not Duplicate(FirstValue, SecondValue)
              then
              begin
                if TwiceNoSolutions > 0 then writeln('or');
                write('Frank''s phone number is:',
                                  D1:3, D2:0, D3:0, '-');
                Display3(SecondValue);
                writeln;
                if TwiceNoSolutions = TwiceMaxNoResults
                then
                begin
                  writeln;
                  writeln('Re-set the maximum number of ',
                                             'solutions--',
                             (TwiceMaxNoResults div 2):0,
                                        ' is too small.');
                  writeln
                end
                else
                begin
                  Results[TwiceNoSolutions+1] := FirstValue;
                  TwiceNoSolutions := TwiceNoSolutions + 2;
                  Results[TwiceNoSolutions] := SecondValue
                end
              end
        end
      end
    end;
  if TwiceNoSolutions > 2 then
  begin
    writeln;
    writeln('The phone number is not unique, ',
            'the folks have to make some trials.');
    writeln('The possible numbers are:');
    Count := 1;
    while Count < TwiceNoSolutions do
    begin
      Display3(Results[Count]);     write('-');
      Display3(Results[Count+1]);  writeln;
      Count := Count + 2
    end
  end
```

```
    else
    if TwiceNoSolutions < 2 then
      begin
        writeln;
        writeln('The pieces of information given ',
                'do not determine a phone number.');
        writeln;
        writeln('(Note that 000-000 is not a valid phone number.)')
      end
  end.  { PhoneNumber }
```

----------- **output** -----------

```
Frank's phone number is:  124-496
```

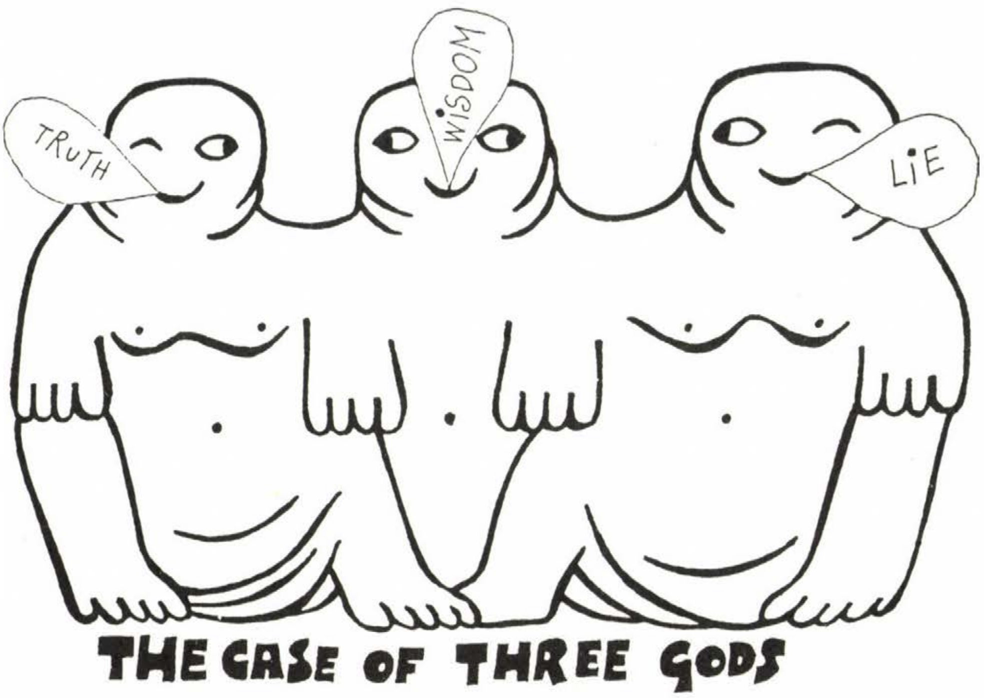
---

program as well.) If array *Results* happens to be too small, the program signals the fault and asks for re-setting the size of the array. One might think that there cannot appear multiple phone numbers, since the first digit is sufficient to generate the complete phone number. The relation between the second and the third digits is, however, disjuntcive, and if $D2=2$, then $D3=4=2*2=2+2$ is obtained in both ways. This is not a theoretical possibility, for the second digit is, in fact, 2 in the only result, and thus that result is generated twice.

The program tries to obtain the second half of the phone number using the relation $4*FirstValue = SecondValue$. This number, however, might have four digits; that is why a four-digit integer, *FirstTimes4*, is used.

The program displays an answer to the question; if there are more than one distinct phone number solutions, the program lists all of them. In displaying the resulting phone number(s), **procedure Display3** is used, which displays an integer between 0 and 999 as three characters, with leading zeros if necessary.

As we can see, the Pascal program for solving this puzzle is very simple, easy to understand, and easy to write. It solves the puzzle very efficiently. The solution of the puzzle by writing a Pascal program requires about the same effort as the systematic investigation-exclusion in the mathematical solution.



3

In an oracle, there sat three gods, the God of Truth, the God of Lie and the God of Wisdom. As they were sitting side-by-side, they were quite alike, one could tell none of them from any one of the others. But everyone knew the God of Truth always told truths, the God of Lie always told lies, and the God of Wisdom sometimes told truths and sometimes told lies.

Once a philosopher arrived at the place to find out the identity of the gods. He asked the god sitting on the left hand side, "Who is sitting on thy side, Mighty God?"

"He is the God of Truth," the god said with dignity.

Then the philosopher asked the god sitting in the middle, between the two others, "Who art thou, Glorious God?"

"I am the God of Wisdom," was the answer.

Finally, the philosopher asked the god sitting on the right hand side, "And who is sitting on thy side, my Lord in the Heavens?"

"The God of Lie," said the god.

How could then the philosopher identify the gods?

3.1 Solution

As for the god sitting between the others, the philosopher has two different answers. That god cannot be the God of Truth, since he told he was the God of Wisdom. He cannot be the God of Wisdom either, since in that case the others, including the God of Truth, would lie. Hence, **the god sitting in the middle** can only **be the God of Lie.** Then **the god sitting on the right hand side is the God of Truth** and **the god sitting on the left hand side is the God of Wisdom.**

3.2 Prolog program

This puzzle essentially differs from the previous ones in that not every statement is necessarily true. If each statement might be true or false or partially true, then there would be hardly any chance to solve the puzzle. Furtunately, the statements in puzzles of the kind can be grouped: there are true statements, there are false satements, and there are statements that may be either true or false, which are most likely to occur in everyday life. In this puzzle we have one statement of each sort; each statement states something about the god sitting in the middle. To identify the gods, the philosopher has to find a one-to-one correspondece between the gods and their positions:

```
truth  lie  wisdom      left  middle  right
```

The method we have implemented in the Prolog program is as follows. We suppose that a god is the god of something, then check if the assumptions made so far contradict the information given in the puzzle. If there is no contradiction, then we either take another god or we are ready, we have identified the gods. Otherwise we have to change our last assumption and try to assign the positions for the gods in another way. This process will end sooner or later, hopefully in a heavenly harmony of statements and identities of gods.

A closer look into the program

In order to fully understand the program, we should have a look at the database first. We have three gods, who are recorded in definition **god**, for example, **god(truth)**. We know the truth value or certainty level of statements made by the individual gods, this information is recorded in definition **god_says**. "The God of Truth always tells truth," for example, translates into the clause **god_says(truth, is_sure)**. The last item in the database is definition **said_middle_is**, which records the statements the gods made (about the identity of the middle god) as answers to the philosopher's questions. Clause **said_middle_is(right, lie)** means, for example, that the god sitting at the *right* hand side said the god sitting in between the others is the God of *Lie*.

Following the problem solving algorithm explained briefly above, we investigate various statements about the identity of the god sitting in between the others. These statements are at different levels of certainty (*is_sure, may_be, cannot_be*), depending on the way they are made; they are recorded, at least temporarily, as dynamic clauses **about_middle**. If, for example, the <u>God</u> of Wisdom <u>says</u> <u>something</u>, then it *may be* true, or if <u>we</u> <u>suppose</u> that "the middle god is the God of Lie," for instance, then we have to accept that statement for *sure*.

After these preliminary thoughts, we can concentrate on the actual algorithm. First, we take a god and suppose that he is sitting in between the others: predicate **suppose_middle(GOD)** records, temporarily, our assumption by calling predicate **temporary(GOD, is_sure)**; then it calls predicate **validity(middle, GOD)** in order to find and record what the god we have chosen to be in the middle said about himself and at which level of certainty. (Predicate **temporary** will be discussed in details later in the section.) At this stage, clauses **about_middle** represent our information derived from the text of the puzzle *and* our assumption. It is now time we checked the consistency of our information, that is, we should check if we have contradictory clauses **about_middle**. The check is actually performed by predicate **contradiction**. We can have a contradiction in two ways: we have two true statements (*is_sure*) that state different things (first clause) or we have two statements, a true and a false one (*is_sure* and *cannot_be*), that state the same thing (second clause). If our information is consistent, we go on and choose a god for the left hand side position. Then we call predicate **validity(left, L)** to find and record what that god said about the one in the middle and at which level of certainty. And if the information gathered so far is consistent, we take the remaining god, place him at the right hand side position, and check the situation as above. If we still cannot find any contradiction, we have identified the gods.

```
%                      The Case of Three Gods

dynamic(about_middle/2).

start:-
   god(M),
   suppose_middle(M),  not contradiction,
   god(L), M =/= L,
   validity(left, L),  not contradiction,
   god(R), R =/= M,  R =/= L,
   validity(right, R), not contradiction,
   nl, write("The gods were sitting in the oracle as follows:"),
   nl, nl, out(L), out(M), out(R), nl, nl,
   retractall(about_middle(_, _)).
start:-
   nl,
   write("The philosopher cannot figure out "),
   write("the identity of the gods."), nl, nl.

suppose_middle(GOD):-
   temporary(GOD, is_sure),
   validity(middle, GOD).

temporary(GOD, CERTAINTY):-
   asserta(about_middle(GOD, CERTAINTY)).
temporary(GOD, CERTAINTY):-
   retract(about_middle(GOD, CERTAINTY)), !, fail.

validity(PLACE, GOD_OF):-
   nonvar(PLACE), nonvar(GOD_OF),
   god_says(GOD_OF, CERTAINTY),
   said_middle_is(PLACE, GOD),
   temporary(GOD, CERTAINTY).

contradiction:-
   about_middle(X, is_sure),
   about_middle(Y, is_sure),
   X =/= Y.
contradiction:-
   about_middle(X, is_sure),
   about_middle(X, cannot_be).

out(X):- write("     God of "), write(X).

god(truth).
god(lie).
god(wisdom).
```

```
god_says(truth,  is_sure).
god_says(lie,    cannot_be).
god_says(wisdom, may_be).

said_middle_is(left,   truth).
said_middle_is(middle, wisdom).
said_middle_is(right,  lie).
```

----------- **output** -----------

```
? start.

The gods were sitting in the oracle as follows:

     God of wisdom     God of lie     God of truth

Yes
```

If the set of clauses **about_middle** prove to be contradictory at any stage of the above algorithm, the program backtracks, deletes the most recently asserted clauses and, taking the next possible value of variables *R*, *L*, or *M*, tests another branch of the search tree. The temporary assertion and retraction, which have key roles in the program, are performed by predicate **temporary**. When the problem solving process goes ahead and calls predicate **temporary**, its first clause asserts a new clause at the beginning of a dynamic definition. When the process backtracks, on the other hand, the second clause of **temporary** is activated, which deletes the first clause in the dynamic definition, that is, the most recently asserted one, and backtracking goes on. Obviously, such a definition may be very useful in many other programs, too. That is why some Prolog implementations have "backtrackable" **assert** and **retract** as built-in predicates. For example MPROLOG provides built-in predicates

add_statement_b(P) and **del_statement_b(P)**

Using these predicates, we can write more straightforward and more concise programs.

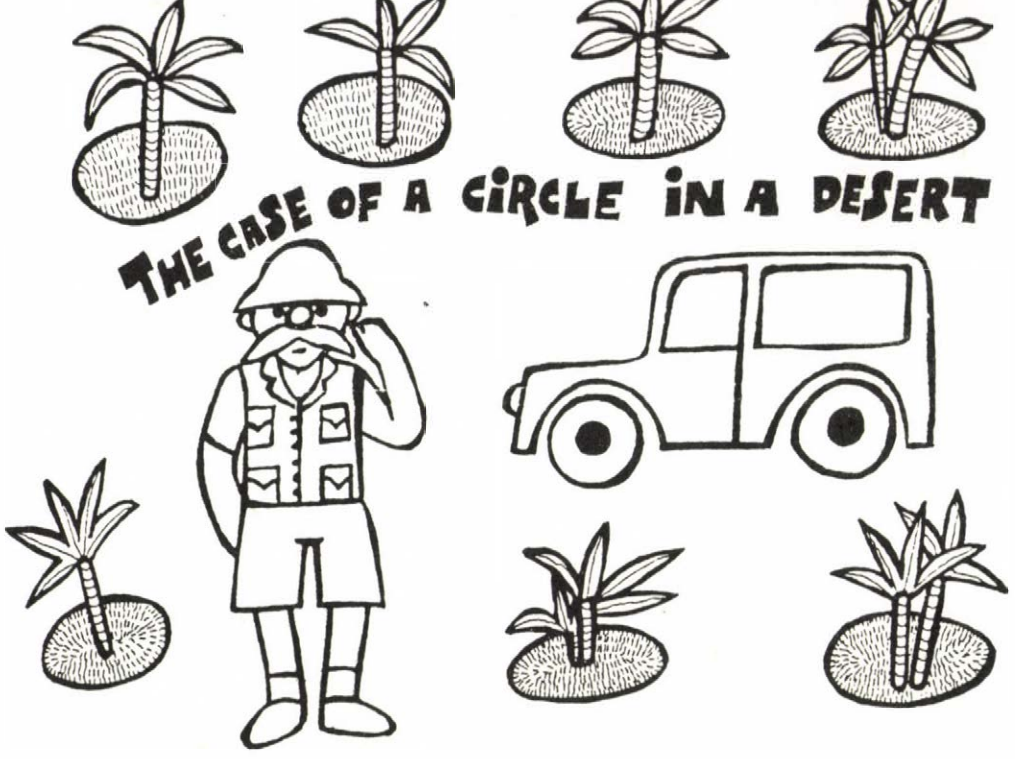
On having a look at the mathematical solution and the Prolog program, one might ask: Is it worth writing a program for an easy problem like that? To tell the truth, the mathematical solution is, in fact, simpler than the program. But it is due to the small number of conditions. In other puzzles of the same kind, we have to cope with more conditions--if we can. It is, however, relatively easy to modify the program to handle many more conditions.

Built-in predicates used in the program

not, =/=, nl, write, retractall, retract, asserta, nonvar, !, fail.

Exercise

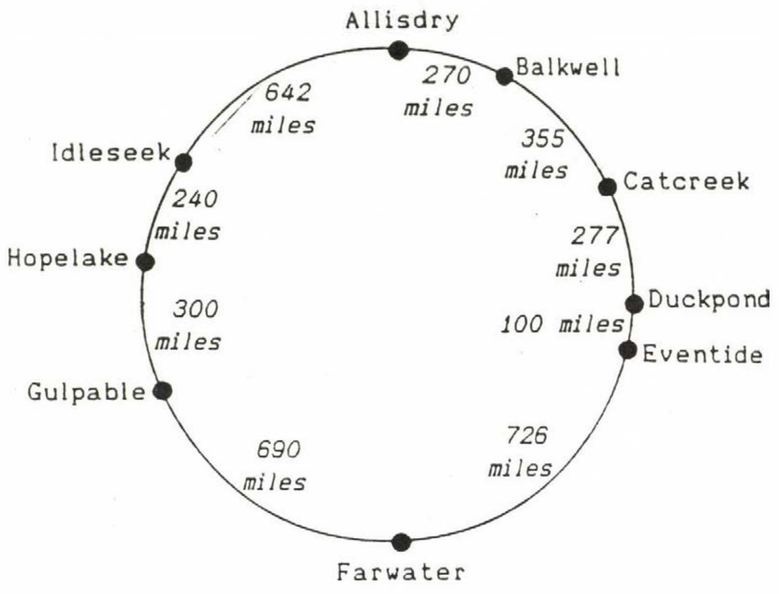
E3.1 Suppose you have "backtrackable" version of predicates **assert** and **retract**. Rewrite the program using those predicates.



4

Kiwi is a prosperous travel agency specialized for long-distance air trips. The secret of its fortune is very simple: it offers tours around the wonderful oases of the Nowhere Land Desert. If one wants to have a go, he chooses an oasis to start from, and he and his range rover are taken by a helicopter to that oasis. Then the helicopter returns, and he starts his drive around. By the end of the term, agreed upon with the agent before the trip, he has to come back to that oasis, for then the helicopter fetches him and his range rover and the journey is over. The price includes food as well as gas used during the journey.

When Mr. Prudent went to the Kiwi Travel Agency, he got a map of the desert (a sketch of the map is shown below). The map and the illustrated brochure, exploring some wonders of the desert and promising many more, convinced him: he should give it a try. But when he went back to the Agency a few days later, the agent told him there had arisen some "tiny" difficulties.



(1) "There are only a few gallons of gas left in the oases:

11 gallons in Allisdry,	*1 gallon in Eventide,*
14 gallons in Balkwell,	*28 gallons in Farwater,*
11 gallons in Catcreek,	*20 gallons in Gulpable,*
32 gallons in Duckpond,	*2 gallons in Hopelake,*

and 25 gallons in Idleseek.

(2) "The helicopter has to carry a lot of food to supply the oases, so it can take your range rover with no gas in it only."

"No problem," said Mr. Prudent, "I'll take some gallons from the helicopter's gas."

(3) "Unfortunately, it's impossible," the agent said. "The gas for a helicopter is of a quite different kind. You can't use it. But don't worry. How far can you get with a gallon of gas in the tank?"

(4) "25 miles or so," said Mr. Prudent wearing a bit of a long face.

(5) "Look," the agent said. "Your range rover can take all the gas you need for the whole drive, can't it?" Mr. Prudent nodded. "O.K. You wanna drive around, don't you? Then, eventually, you can start at any oasis. Well, tell me which way round you want to drive, and we'll be taking you to an oasis from which you can drive around."

"Do you think there is enough gas to finish a circle?"

"Sure, there is."

"No, thanks. I can't take the risk," said Mr. Prudent and left. 'Now I should go to the Vulture Agency to find a flight over Nowhere Land. Then I can at least have a look at the desert from high above,' he thought on his way home.

Was Mr. Prudent too cautious when he did not trust the agent?

4.1 Solution

To answer the question, we should check if it is possible to drive around with Mr. Prudent's range rover in any way round, and if it *is* possible, we should find the starting oasis and the direction to be followed.

To avoid any unnecessary work, first we should check if the combined amount of gas in all oases is enough for a complete circle. If it is not enough, then it is, of course, impossible to drive around. In the puzzle, the combined amount of gas in all oases as well as the amount of gas needed to complete a circle is 144 gallons; so we are not so lucky, we have to keep on working.

Once the desired direction is fixed, we have to find an oasis in which there is enough gas for the range rover to reach the next oasis. If there are such oases, any one of them can be a starting oasis and we have to try them out in turn until we find a complete circle or there are no more candidates. Otherwise, there is no oasis to start from that way round.

When we reach the next oasis, we combine the gas remained and the gas that is originally in that oasis, and try to drive on from that oasis to the next one. If we do not have enough gas to drive to the next oasis, then we cannot complete that circle. Otherwise we test the next oasis similarly.

If we cannot complete a circle in one direction, then, naturally (see Exercise E4.1), we have to try to find a complete circle driving in the other direction. And we have to do this even if we have found a complete circle in the direction fixed first, since the agent said that Mr. Prudent feel free to choose any direction, the agency would find an oasis for his trip to start from. Notice, however, that one complete circle found each way round is enough.

Let's choose the clockwise direction first and try to find a starting oasis. With a little bit of calculation, we get that the candidates are: Allisdry, Duckpond, Farwater, and Gulpable. And soon we have: Starting from Allisdry oasis, Mr. Prudent cannot get farther than Catcreek, i.e., he cannot reach the next oasis, Duckpond. Starting from Duckpond oasis, he can drive as far as Eventide, but he cannot reach the next oasis, Farwater. Fortunately, **starting from Farwater oasis, Mr. Prudent succeeds in driving around in clockwise direction, using up the whole amount of gas in the oases.** Although there is another oasis, Gulpable, to start from, we skip it and investigate the case when Mr. Prudent wants to drive the other way round.

Let's try to find a starting oasis for the drive in counterclockwise direction. With a little work we get that the candidates are: Balkwell, Duckpond, and Idleseek. Now we obtain: Starting from Balkwell oasis, Mr. Prudent cannot get farther than Allisdry, i.e., he cannot reach the next oasis, Idleseek. Fortunately, **starting from Duckpond oasis, he succeeds in driving around in counterclockwise direction, using up the whole amount of gas in the oases.** Although there is another possible starting oasis, Idleseek, we may skip it. And we conclude that Mr. Prudent was too cautious when he did not trust the agent.

Remark: It is easy to check that there is no other way to drive around.

Exercise

E4.1 Is it true that if Mr. Prudent can complete a circle driving one way round, then he can complete a circle driving the other way round?

4.2 Prolog programs

As one can see, the above solution of the puzzle is rather mechanical. And, obviously, a program is a much more appropriate means of solution than manual calculation. In this section, we are going to present two Prolog programs that implement the same approach as the mathematical solution. We note, however, that most other programming languages provide equally adequate tools for implementing the algorithm.

Program Version 1

The data relevant to the program are recorded as clauses **oasis**, **neighbors**, and **direction**. **oasis** is a two-argument predicate, the first argument is the name of the oasis, while the second one is the amount of gasoline (gallons) the oasis provides initally. Predicate **neighbors** has three arguments: the first and the second arguments are the names of two neighboring oases listed in clockwise direction; the third argument shows the distance (miles) between them. The two opposite directions, *clockwise* and *counterclockwise*, are also recorded.

The algorithm is very simple: first we choose an oasis to start from, then we choose a direction, and try to reach the same oasis driving permanently in the chosen direction. If we fail to complete a circle, then we try driving in the opposite direction, and if we fail again, we choose another oasis to start from. In order to obtain all solutions, we force backtracking even after a successfully completed circle.

The core of the program is definition **reach(FROM, TO, BEFORE, AFTER, DIR)**, which defines recursively when an oasis (*TO*) is reachable, considering the gas supply (*BEFORE, AFTER*), from another one (*FROM*) driving permanently in the same direction (*DIR*). A destination oasis *TO* is reachable from a starting oasis *FROM* if there is an oasis *Z* such that *Z* is reachable from *FROM* and *TO* is reachable from *Z* (second clause). Either of two neighboring oases is reachable from the other if the combined amount of gas the driver has at the starting oasis (*HAVE*), that is the amount remained in the tank (*BEFORE*) plus the amount he can get at that oasis, is not less than the amount needed to reach the other oasis (*NEED*); AFTER contains the amount of gas remained upon arriving at the destination oasis (first clause). Notice that any oasis can be reached starting from itself only if every other oasis is visited en route. Hence the subgoal **reach(X, X, 0, REST, DIR)**, where *X* is bound to a particualr oasis, *DIR* is bound to a particular direction, and 0 is the initial amount of gas in the tank, provides us with a solution with the amount of gas remained at the end of the trip in variable *REST*.

Notice that computation is quite important in this program, and, what causes the real problem, the results may not always be integers. Unfortunately, a number of Prolog dialects do not support floating point arithmetic, which means that programs written in those dialects cannot directly handle fractions. To overcome this difficulty, we used the following trick: Having investigated the data and the operations to be performed on them, we realized that decimal fractions will not have more than two digits. On this basis, we multiplied the numbers by 100, performed the required operations, and, before printing the result, we produced the integer and fractional parts of the result, and arranged a suitable display format.

```
%             The Case of a Circle in a Desert
%                        Version 1


dynamic(circle/0).

start:-
   oasis(X,_),
   direction(DIR),
   reach(X, X, 0, REST, DIR), assert(circle),
   nl, write("Starting from "), write(X), write(" oasis, "),
   nl, write("it is possible to drive around in "),
   write(DIR), write(" direction."), nl,
   write("   Gas remained: "), out(REST), nl, fail.
start:- circle, retractall(circle).
start:-
   nl, write("It is impossible to drive around."), nl, nl.

reach(FROM, TO, BEFORE, AFTER, DIR):-
   ( DIR = clockwise,  neighbors(FROM, TO, DISTANCE);
     DIR = counterclockwise, neighbors(TO, FROM, DISTANCE) ), !,
   DISTANCE1 is 100*DISTANCE,
   oasis(FROM, GAS),  GAS1 is 100*GAS,
   NEED is DISTANCE1 div 25,
   HAVE  is GAS1 + BEFORE,
   AFTER is HAVE - NEED,
   HAVE >= NEED.

reach(FROM, TO, BEFORE, AFTER, DIR):-
   reach(FROM, Z, BEFORE, AFTER_Z, DIR),
   reach(Z,  TO,  AFTER_Z,  AFTER, DIR).

out(N):-
   N >= 0,
   P is N div 100,
   Q is N mod 100,
   write(P), write("."), write(Q).

oasis(allisdry, 11).
oasis(balkwell, 14).
oasis(catcreek, 11).
oasis(duckpond, 32).
oasis(eventide,  1).
oasis(farwater, 28).
oasis(gulpable, 20).
oasis(hopelake,  2).
oasis(idleseek, 25).

neighbors(allisdry, balkwell, 270).
neighbors(balkwell, catcreek, 355).
neighbors(catcreek, duckpond, 277).
neighbors(duckpond, eventide, 100).
neighbors(eventide, farwater, 726).
neighbors(farwater, gulpable, 690).
neighbors(gulpable, hopelake, 300).
neighbors(hopelake, idleseek, 240).
neighbors(idleseek, allisdry, 642).

direction(clockwise).
direction(counterclockwise).
```

----------- **output** -----------

```
? start.

Starting from duckpond oasis,
it is possible to drive around in counterclockwise direction.
   Gas remained: 0.0

Starting from farwater oasis,
it is possible to drive around in clockwise direction.
   Gas remained: 0.0

Yes
```

Built-in predicates used in program Version 1

write, nl, =, >=, -, +, *, div, mod, is, !, ;, fail, assert, retractall.

Program Version 2

In this version we extend the previous program so that it should have an extra control step at the beginning and that it should produce a user-friendly trace of the trials. Before even trying to move in any direction, it is wise to check if the total amount of gas is enough for driving a whole circle. This preliminary check is performed by predicate **pre_check**, which calls predicate **accumulate** to compute the combined amount of gas and the length of a circle.

In order to obtain a user-friendly trace of the trials, we have extended the end of the first clause in definition **reach** as well as the clauses in definitions **start**, **out**, and **out1** which produce some output. Predicates **out** and **out1** now produce tabulated messages which may include negative fractions as well. Although the result justifies the effort, we should notice that the size of the program fragment to produce pretty input/output is not at all negligible (see also the Pascal program in Section 1).

```
%            The Case of a Circle in a Desert
%                          Version 2

dynamic(circle/0).
dynamic(quantity/1).

start:-
   pre_check,
   oasis(X,_), nl, nl,
   write("Start from "), write(X), write(" oasis in"), nl,
   direction(DIR),
   write(DIR), write(" direction:"), nl,
   reach(X, X, 0, REST, DIR),  assert(circle),
   nl, write("Starting from "), write(X), write(" oasis, "),
   nl, write("it is possible to drive around in "),
   write(DIR), write(" direction."), nl,
   write("   Gas remained: "), out(REST), nl, nl, fail.
start:- circle, retractall(circle).
start:-
   nl, write("It is impossible to drive around."), nl, nl.

pre_check:-
   accumulate(gas, GAS), accumulate(miles, MILES),
   CAN_DRIVE is 25*GAS, !, CAN_DRIVE >= MILES.

accumulate(WHAT, TOTAL):-
   assert(quantity(0)),
   ( WHAT == gas, oasis(_, QTY);
     WHAT == miles, neighbors(_, _, QTY) ),
   retractfirst(quantity(Q)), Q1 is Q + QTY,
   assert(quantity(Q1)), fail.
accumulate(WHAT, TOTAL):-
   ( WHAT == gas; WHAT == miles ),
   retract(quantity(TOTAL)).
accumulate(WHAT, TOTAL):-
   nl, write("Program error--"), write(WHAT), nl, abort.

reach(FROM, TO, BEFORE, AFTER, DIR):-
  ( DIR = clockwise,  neighbors(FROM, TO, DISTANCE);
   DIR = counterclockwise, neighbors(TO, FROM, DISTANCE) ), !,
   DISTANCE1 is 100*DISTANCE,
   oasis(FROM, GAS),  GAS1 is 100*GAS,
   NEED is DISTANCE1 div 25,
   HAVE  is GAS1 + BEFORE,
   AFTER is HAVE - NEED,
   tab(4), write(FROM), write(" -> "), write(TO),
   write("  Have: "), out(HAVE), write("  Need: "), out(NEED),
   write("  Remains: "), out(AFTER), nl,
   ( HAVE < NEED, !,
     tab(59), write("Wrong way, go back."), nl, fail; true ).
```

```
reach(FROM, TO, BEFORE, AFTER, DIR):-
   reach(FROM, Z, BEFORE, AFTER_Z, DIR),
   reach(Z,  TO,  AFTER_Z,  AFTER, DIR).

out(N):- N >= 0, tab(1), out1(N), !.
out(N):- N1 is -N, write("-"), out1(N1).

out1(N):-
   P is N div 100,
   Q is N mod 100,
   ( P < 10, !, tab(1); true ),
   write(P), write("."), write(Q),
   ( Q < 10, !, write(0); true ).

retractfirst(CLAUSE):- retract(CLAUSE), !.

oasis(allisdry, 11).
oasis(balkwell, 14).
oasis(catcreek, 11).
oasis(duckpond, 32).
oasis(eventide,  1).
oasis(farwater, 28).
oasis(gulpable, 20).
oasis(hopelake,  2).
oasis(idleseek, 25).

neighbors(allisdry, balkwell, 270).
neighbors(balkwell, catcreek, 355).
neighbors(catcreek, duckpond, 277).
neighbors(duckpond, eventide, 100).
neighbors(eventide, farwater, 726).
neighbors(farwater, gulpable, 690).
neighbors(gulpable, hopelake, 300).
neighbors(hopelake, idleseek, 240).
neighbors(idleseek, allisdry, 642).

direction(clockwise).
direction(counterclockwise).
```

----------- **output** -----------

```
? start.

Start from allisdry oasis in
clockwise direction:
   allisdry -> balkwell  Have:  11.00  Need:  10.80  Remains:   0.20
   balkwell -> catcreek  Have:  14.20  Need:  14.20  Remains:   0.00
   catcreek -> duckpond  Have:  11.00  Need:  11.80  Remains: - 0.80
                                                          Wrong way, go back.
counterclockwise direction:
   allisdry -> idleseek  Have:  11.00  Need:  25.68  Remains: -14.68
                                                          Wrong way, go back.
```

```
Start from balkwell oasis in
clockwise direction:
    balkwell -> catcreek  Have:  14.00  Need:  14.20  Remains: - 0.20
                                                          Wrong way, go back.
counterclockwise direction:
    balkwell -> allisdry  Have:  14.00  Need:  10.80  Remains:   3.20
    allisdry -> idleseek  Have:  14.20  Need:  25.68  Remains: -11.48
                                                          Wrong way, go back.


Start from catcreek oasis in
clockwise direction:
    catcreek -> duckpond  Have:  11.00  Need:  11.80  Remains: - 0.80
                                                          Wrong way, go back.
counterclockwise direction:
   catcreek -> balkwell  Have:  11.00  Need:  14.20  Remains: - 3.20
                                                         Wrong way, go back.


Start from duckpond oasis in
clockwise direction:
    duckpond -> eventide  Have:  32.00  Need:   4.00  Remains:  28.00
    eventide -> farwater  Have:  29.00  Need:  29.40  Remains: - 0.40
                                                          Wrong way, go back.
counterclockwise direction:
    duckpond -> catcreek  Have:  32.00  Need:  11.80  Remains:  20.92
    catcreek -> balkwell  Have:  31.92  Need:  14.20  Remains:  17.72
    balkwell -> allisdry  Have:  31.72  Need:  10.80  Remains:  20.92
    allisdry -> idleseek  Have:  31.92  Need:  25.68  Remains:   6.24
    idleseek -> hopelake  Have:  31.24  Need:   9.60  Remains:  21.64
    hopelake -> gulpable  Have:  23.64  Need:  12.00  Remains:  11.64
    gulpable -> farwater  Have:  31.64  Need:  27.60  Remains:   4.40
    farwater -> eventide  Have:  32.40  Need:  29.40  Remains:   3.00
    eventide -> duckpond  Have:   4.00  Need:   4.00  Remains:   0.00

Starting from duckpond oasis,
it is possible to drive around in counterclockwise direction.
   Gas remained:   0.00


Start from eventide oasis in
clockwise direction:
    eventide -> farwater  Have:   1.00  Need:  29.40  Remains: -28.40
                                                          Wrong way, go back.
counterclockwise direction:
    eventide -> duckpond  Have:   1.00  Need:   4.00  Remains: - 3.00
                                                          Wrong way, go back.
```

```
Start from farwater oasis in
clockwise direction:
    farwater -> gulpable  Have:  28.00  Need:  27.60  Remains:   0.40
    gulpable -> hopelake  Have:  20.40  Need:  12.00  Remains:   8.40
    hopelake -> idleseek  Have:  10.40  Need:   9.60  Remains:   0.80
    idleseek -> allisdry  Have:  25.80  Need:  25.68  Remains:   0.12
    allisdry -> balkwell  Have:  11.12  Need:  10.80  Remains:   0.32
    balkwell -> catcreek  Have:  14.32  Need:  14.20  Remains:   0.12
    catcreek -> duckpond  Have:  11.12  Need:  11.80  Remains:   0.40
    duckpond -> eventide  Have:  32.40  Need:   4.00  Remains:  28.40
    eventide -> farwater  Have:  29.40  Need:  29.40  Remains:   0.00

Starting from farwater oasis,
it is possible to drive around in clockwise direction.
   Gas remained:  0.00

counterclockwise direction:
    farwater -> eventide  Have:  28.00  Need:  29.40  Remains: - 1.40
                                                      Wrong way, go back.


Start from gulpable oasis in
clockwise direction:
    gulpable -> hopelake  Have:  20.00  Need:  12.00  Remains:   8.00
    hopelake -> idleseek  Have:  10.00  Need:   9.60  Remains:   0.40
    idleseek -> allisdry  Have:  25.40  Need:  25.68  Remains: - 0.28
                                                      Wrong way, go back.
counterclockwise direction:
    gulpable -> farwater  Have:  20.00  Need:  27.60  Remains: - 7.60
                                                      Wrong way, go back.


Start from hopelake oasis in
clockwise direction:
    hopelake -> idleseek  Have:   2.00  Need:   9.60  Remains: - 7.60
                                                      Wrong way, go back.
counterclockwise direction:
    hopelake -> gulpable  Have:   2.00  Need:  12.00  Remains: -10.00
                                                      Wrong way, go back.


Start from idleseek oasis in
clockwise direction:
    idleseek -> allisdry  Have:  25.00  Need:  25.68  Remains: - 0.68
                                                      Wrong way, go back.
counterclockwise direction:
    idleseek -> hopelake  Have:  25.00  Need:   9.60  Remains:  15.40
    hopelake -> gulpable  Have:  17.40  Need:  12.00  Remains:   5.40
    gulpable -> farwater  Have:  25.40  Need:  27.60  Remains: - 2.20
                                                      Wrong way, go back.

Yes
```

Notice finally that groups

```
( HAVE < NEED, !,
  tab(59), write("Wrong way, go back."), nl, fail;
  true )
```

or

```
( P < 10, !, tab(1); true )
```

are of the pattern

```
( CONDITION, !, THEN ;  true )
```

which, declaratively, means that the group is true if *CONDITION* as well as *THEN* is true, or if *CONDITION* is false. Procedurally, the group translates into the following: "If *CONDITION* is true then evaluate *THEN* and, if the evaluation yields true, go on to the next predicate following the group, or, if the evaluation yields false, backtrack starting at the predicate immediately preceding the group; otherwise do nothing but go on to the next predicate right after the group." The corresponding Pascal control structure is

if *CONDITION* **then** *THEN* ;

For some more details about the subject, see Section 8.

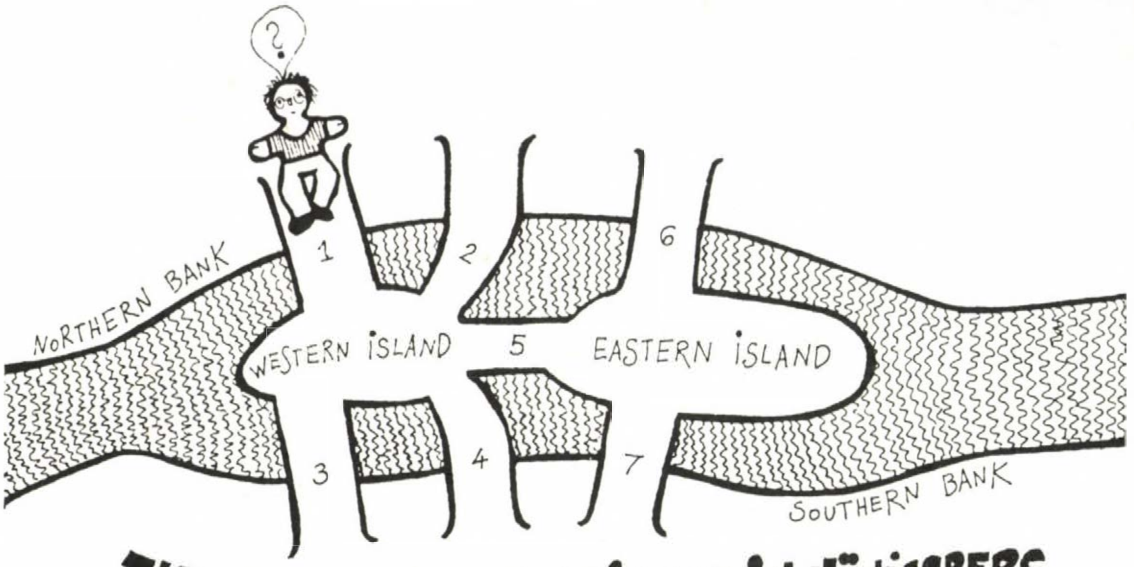
Built-in predicates used in program Version 2

write, nl, tab, ==, =, >=, <, -/2, +, *, -/1, div, mod, is, !, ;, fail, true, assert, retract, retractall, abort.

Notice that -/1 denotes the unary minus, the minus sign, while -/2 denotes the binary operator *subtract*.

Exercise

E4.2 Rewrite the programs in Pascal.



THE CASE OF THE BRIDGES IN KÖNIGSBERG

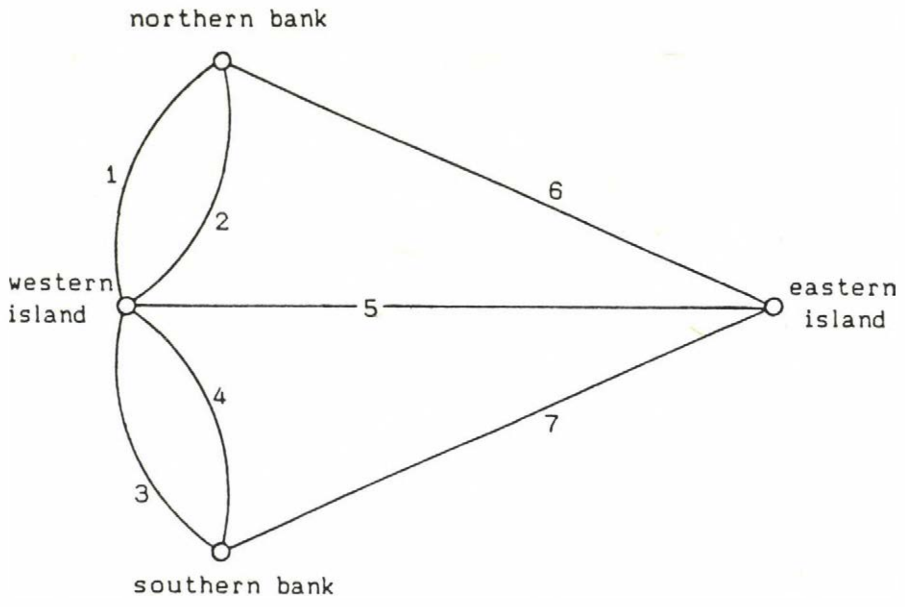
5

In the 18th century, there were seven bridges over the river Pregel in the city of Koenigsberg (or Kaliningrad, as is known nowadays). The figure above shows the relevant part of a map of the city. The citizens living in Koenigsberg used to take walks along the banks and in the islands crossing one bridge or another. And then, among the dames, noblemen, and noblewomen parading along on holidays, the question arose: Is it possible to walk around crossing every bridge exactly once?

Answer the question.

5.1 Solution

When one actually takes a walk, he may walk around on the banks or islands before or after crossing a bridge. Such mainland or island walks are of no importance, however, when we want to solve the puzzle. Neither are of any importance the walks from one bridge to another on the same bank or island. Hence, we can represent a bank or an island as a node and a bridge as an edge between two nodes, having the following graph representation of the puzzle:



And the question of the citizens of Koenigsberg is as follows: (Q) Is it possible to walk along the edges of the graph in such a way that every edge is used exactly once?

We say a graph is connected if there is a route via its edges between any two nodes in it. Consider a node of a connected graph for which the answer to question (Q) is yes, and let *R* denote a route covering every edge exactly once. If the node is neither the start node nor the end node of route *R*, then whenever we reach the node via an edge, we should leave it via another. Thus, the number of the edges connected to the node is even. If the node is the start (or end) node of route *R*, then we reach it at the beginning (or at the end) of the route and we may reach it several times later (or before). In the latter case, whenever we reach it via an edge, we leave it as well via another, which means an even number of edges connected to the node. An extra initial (or final) edge is also connected to this node (this is the only edge connected to it if the node is not reached en route). Therefore, the start node and the end node may have an odd number of edges connected to them. (Moreover, they have an even number of edges connected to them if and only if they are identical.)

From the above it follows that **for the existence of a route covering every edge exactly once, it is necessary the graph have *at most two* nodes with an odd number of edges connected to them.**

Consider now the graph representation of the puzzle and check if the above condition holds. Since that graph has three nodes with three edges connected to them and one node with five edges connected to it, the condition does not hold. Consequently, **it was impossible to walk around in Koenigsberg crossing every bridge exactly once.**

Exercises

E5.1 Show that the above condition in boldface is a necessary condition for nonconnected graphs as well.

E5.2 Prove that the above condition is a necessary and sufficient condition for connected graphs.

5.2 Prolog program

This famous old puzzle helped a lot in developing a new branch of mathematics, namely graph theory. Leonhard Euler, a prominent mathematician in the 18th century, introduced new concepts, and, on that new level of abstraction, he proved that it was impossible to walk around in Koenigsberg crossing every bridge exactly once. If then, when this problem arose, computers had been existed, a practical computer programmer would probably have suggested a solution of an entirely different kind: he would have programmed a trivial algorithm of trying the different paths, which, manually, is a very hard job even for seven bridges.

Now, at the end of the twentieth century, we have the tools to write a program which implements that algorithm. The program must give the same result, of course. Still, the program is not one without interest. It is very easy to alter the data in the program, which enables us to solve various problems of the kind with negligible extra effort. We will illustrate this possibility at the end of the section.

A closer look into the program

The seven bridges connect two banks and two islands, which are the possible starting point of our walk. We take a possible starting point and try to walk around by calling predicate **walk**. As soon as it is proved that we cannot cross every bridge exactly once, the algorithm backtracts and we test the next possible starting point. The algorithm stops if it has

```
%          The Case of the Bridges in Koenigsberg


start:-
   environment, nl, nl,
   write("Try to take a walk crossing the bridges as required."),
   nl, land(LAND1),
   nl, write("Start from "), write(LAND1),
   write(" and go ..."), nl,
   walk(LAND1, _, [1,2,3,4,5,6,7], [], STOPS, [], ROUTE),
   out(STOPS, ROUTE).
start:-
   nl,write("It is impossible to walk around "),
   write("crossing every bridge exactly once."),nl.

environment:- set_state(evaluation_limit, 50000).

walk(FROM, TO, [OVER], ST, STOPS, RT, ROUTE):-
   cross_bridge(FROM, TO, OVER, [OVER], REST),
   append(ST, [TO], STOPS), append(RT, [OVER], ROUTE), !.
walk(FROM, TO, BRIDGES, ST, STOPS, RT, ROUTE):-
   cross_bridge(FROM, STOP, OVER, BRIDGES, REST),
   append(ST, [STOP], ST1), append(RT, [OVER], RT1),
   walk(STOP, TO, REST, ST1, STOPS, RT1, ROUTE).

cross_bridge(LAND1, LAND2, OVER, BRIDGES, REST):-
   member(OVER, BRIDGES),
   ( bridge(OVER, connects(LAND1, LAND2));
     bridge(OVER, connects(LAND2, LAND1)) ),
   delete(OVER, BRIDGES, REST).

append([], L2, L2).
append([X|L1], L2, [X|L]):- append(L1, L2, L).

delete(H, [H|T], T).
delete(X, [H|T], [H|NEWLIST]):- delete(X, T, NEWLIST).

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

out([X], [Y]):-
   tab(30), write("over to "), write(X),
   write(" via "), write(Y), write("."), nl, nl, !.
out([H1|T1], [H2|T2]):-
   tab(30), write("over to "), write(H1),
   write(" via bridge "), write(H2), write(", then"), nl,
   out(T1, T2).
out([], []).
```

```
land(northern_bank).
land(western_island).
land(southern_bank).
land(eastern_island).

bridge(1, connects(northern_bank,  western_island)).
bridge(2, connects(northern_bank,  western_island)).
bridge(3, connects(southern_bank,  western_island)).
bridge(4, connects(southern_bank,  western_island)).
bridge(5, connects(western_island, eastern_island)).
bridge(6, connects(northern_bank,  eastern_island)).
bridge(7, connects(southern_bank,  eastern_island)).
```

----------- **output** -----------

```
? start.

Try to take a walk crossing the bridges as required.

Start from northern_bank and go ...

Start from western_island and go ...

Start from southern_bank and go ...

Start from eastern_island and go ...

It is impossible to walk around crossing every bridge exactly once.

Yes
```

investigated all possible starting points and if it could not find any way to cross every bridge exactly once. It also stops as soon as a particular way for crossing every bridge exactly once is found, in which case it calls predicate **out**, which lists that route.

The most important part of the program is predicate **walk(FROM, TO, BRIDGES, ST, STOPS, RT, ROUTE)** where *FROM* is the piece of land we walk from, *TO* is the piece of land we walk to, and *BRIDGES* is the list of bridges to be crossed. The next four arguments store information about the route we have made: *STOPS* collects the banks and islands visited in turn, while *ROUTE* collects the bridges crossed in turn. Arguments *ST* and *RT* are auxiliary variables for the collection of stops and bridges crossed. Our subgoal of "walk from *LAND1* to anywhere such that every bridge be crossed exatcly once" translates into the clause

```
walk(LAND1, _, [1,2,3,4,5,6,7], [], STOPS, [], ROUTE)
```

The recursive definition **walk** has two clauses. The first handles the case when exactly one bridge left (*[OVER]* is a list of exactly one element). If it is possible to cross that last bridge, we complete the output lists and our walk ends in success. If we have more than one bridge to cross (second clause), we take one of them and cross it arriving at a bank or island; we append those items to the lists *STOP* and *ROUTE*, and the recursive process goes on.

The actual crossing of bridges is performed by predicate **cross_bridge**. This predicate takes the next bridge to be crossed (predicate **member**) and examines if that bridge connects the place we are standing at with another bank or island. If it does, then that bridge is taken out of the list of the bridges not yet crossed (predicate **delete**).

Another problem

Suppose now that, after a heavy rain, Koenigsberg were flooded and bridge No. 7 were distroyed. The question of the citizens can now be the same: "Can they walk around crossing each one of the bridges remained exactly once?" Having written the Prolog program, we can answer the question very easily. All we have to do is change the call to predicate **walk** for the following:

```
walk(LAND1, _, [1,2,3,4,5,6], [], STOPS, [], ROUTE)
```

In the case of 6 bridges, the program list a possible way for crossing every bridge exactly once (see the figure below). If we ask the goal

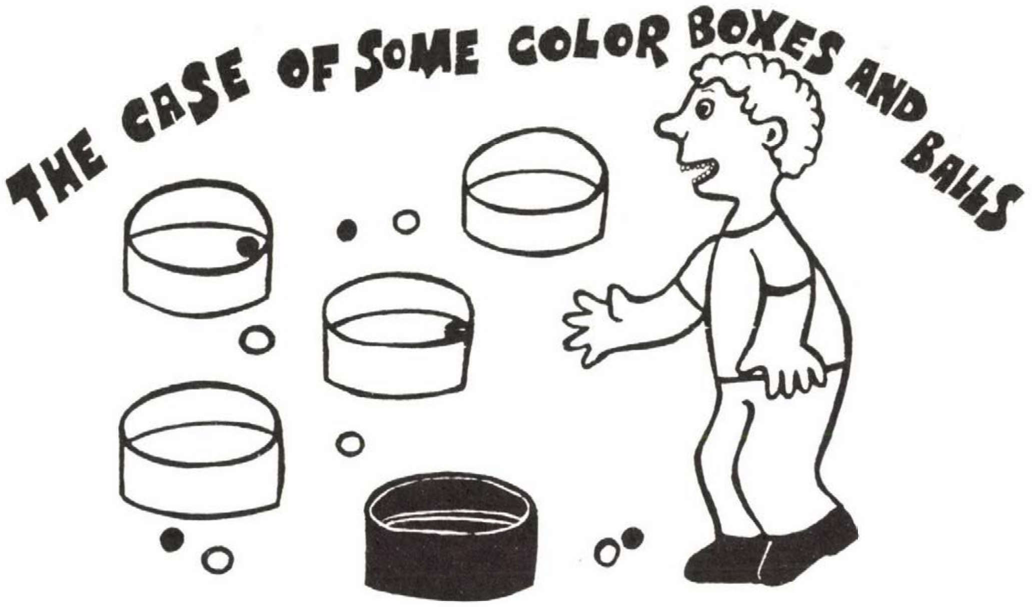
```
? start, fail.
```

the program lists every possible way for crossing the six bridges, each bridge exactly once. We do not wish to present that long output in the report.

```
? start.

Try to take a walk crossing the bridges as required.

Start from northern_bank and go ...
                                 over to western_island via bridge 1, then
                                 over to northern_bank via bridge 2, then
                                 over to eastern_island via bridge 6, then
                                 over to western_island via bridge 5, then
                                 over to southern_bank via bridge 3, then
                                 over to western_island via 4.

Yes
```

Built-in predicates used in the program

write, nl, tab, set_state, ;. !.



6

We have five boxes, a red, a blue, a white, a black, and a green one, and ten balls, 2 red, 2 blue, 2 white, 2 black, and 2 green ones. We should put the balls into the boxes such that:

(1) Into each box, we should put two balls the color of neither of which is the same as that of the box.

(2) There should be no blue ball in the red box.

(3) There should be a box of neutral color with a red and a green ball inside (the neutral colors are: black and white).

(4) The black box should contain balls of cold color (the cold colors are: green and blue).

(5) There should be a box with a white and a blue ball inside.

There should be a black ball in the blue box.

Can we do it?

6.1 Solution

The box of neutral color of constraint (3) cannot be black, since, according to (4), the black box should contain balls of cold color and red is not a cold color. Therefore, **the white box contains a red and a green ball.**

Constraint (4) says that the black box should contain two green balls, or two blue balls, or one green ball and one blue ball. Moreover, there must be a green ball in the white box; and, according to constraint (5), a blue ball, together with a white one, should be in some other box. Consequently, **the black box contains a blue and a green ball.**

What color can the box of constraint (5) be? It can be neither white nor black; according to constraint (2), it cannot be red; and, according to constraint (6), it cannot be blue either. Thus, **the green box contains a white and a blue ball.**

There are two boxes, a red and a blue one, and four balls, a white, a red and two black ones, left. **The red box contanins a black and a white ball,** because it can contain neither a red ball, according to constraint (1), nor two black ones, according to constraint (6).

And, finally, **the blue box contains a red and a black ball**--that is the remaining box and those are the remaining balls. It is easy to check, moreover, that no constraint is violated.

From the above argument it also follows that this is the only way to satisfy all constraints.

6.2 Prolog program

It is a classical puzzle again: we have to put color balls into color boxes so that a couple of requirements be fulfilled. When we solve such a problem, we try to satisfy the most restrictive requirement in order to minimize the number of trials required. This is our basic strategy, which is applicable to any problem of the kind. It is, however, too general, which implies that we have to "invent" a new particular technique for almost each problem.

The Prolog program implements a naive and more general approach: the rules and facts in the program can easily be modified to solve most puzzles of this kind. The boxes and the balls are stored in two lists;, the conditions of the puzzle are represented by rules, each condition has a

```
%         The Case of Some Color Boxes and Balls

start:-
   boxes(BOXES), balls(BALLS),
   put_balls(BOXES, _ , BALLS, _ , [], RESULT),
   out(RESULT).
start:-
   nl, write("The balls cannot be put into the boxes "), nl,
   write("such that all conditions be fulfilled."), nl, nl.

put_balls(BoxesIn, BoxesOut, BallsIn, BallsOut, RESULT0, RESULT):-
   take(BOX, BoxesIn, BoxesOut),
   box(BOX, X, Y, BallsIn, BallsOut),
   append(RESULT0, [BOX,X,Y], RESULT1),
   put_balls(BoxesOut, _ , BallsOut, _ , RESULT1, RESULT).
put_balls([], _ , _ , _ , RESULT, RESULT).

box(blue, black, Y, BallsIn, BallsOut):-
   take(black, BallsIn, BallsTemp),
   take(Y, BallsTemp, BallsOut), Y =/= blue.
box(red, X, Y, BallsIn, BallsOut):-
   take(X, BallsIn, BallsTemp),  X =/= red, X =/= blue,
   take(Y, BallsTemp, BallsOut), Y =/= red, Y =/= blue.
box(black, X, Y, BallsIn, BallsOut):-
   take(X, BallsIn, BallsTemp),  X =/= black, cold_color(X),
   take(Y, BallsTemp, BallsOut), Y =/= black, cold_color(Y).
box(BOX, red, green, BallsIn, BallsOut):-
   BOX =/= red, BOX =/= green, neutral_color(BOX),
   take(red, BallsIn, BallsTemp),
   take(green, BallsTemp, BallsOut).
box(BOX, white, blue, BallsIn, BallsOut):-
   BOX =/= white, BOX =/= blue,
   take(white, BallsIn, BallsTemp),
   take(blue, BallsTemp, BallsOut).

out(RESULT):-
   append([BOX, BALL1, BALL2], REST, RESULT),
   nl, write("There is a "), write(BALL1), write(" and a "),
   write(BALL2), write(" ball in the "),
   write(BOX), write(" box."),
   out(REST).
out([]):- nl, nl.

take(H, [H|T], T).
take(X, [H|T], [H|NEWLIST]):-
   take(X, T, NEWLIST).

append([], L2, L2).
append([X|L1], L2, [X|L3]):- append(L1, L2, L3).
```

```
boxes([white, red, green, blue, black]).

balls([white,white, red,red, green,green,
                                 blue,blue, black,black]).

neutral_color(white).
neutral_color(black).

cold_color(green).
cold_color(blue).
```

----------- **output** -----------

```
? start.

There is a red and a green ball in the white box.
There is a white and a black ball in the red box.
There is a white and a blue ball in the green box.
There is a black and a red ball in the blue box.
There is a green and a blue ball in the black box.

Yes
```

corresponding clause **box(BOX, BALL1, BALL2, BallsIn, BallsOut)** except the first one, which is incorporated into the **box** clauses in order to improve efficiency.

The problem solving algorithm is very simple: Take a box and put two balls into it such that all requirements be satisfied. If you can do that, take the box and the two balls out of the lists, and repeat the procedure by taking the next box in the list. If you fail to find balls as required, you should have filled in a previous box in some other way, therefore you should now put back boxes and balls inside them onto the lists until you find another way for filling in a box (backtracking). If you do not find any new way of filling in any box, then you have examined all possible cases and you can conlcude that it is impossible to fulfill all requirements. On the other hand, if both lists (the list of boxes and the list of balls) are empty, you have filled in the boxes so that all requirement are satisfied.

The above algorithm is implemented by the recursive definition **put_balls(BoxesIn, BoxesOut, BallsIn, BallsOut, RESULTO, RESULT)**, where *BoxesIn* and *BoxesOut* denote, respectively, the list of empty boxes and the list of balls not yet put into boxes at the beginning of a step of the algorithm, while *BoxesOut* and *BallsOut* denote, respectively, the list of empty boxes and the list of balls not yet put into boxes at the end of a step of the algorithm; *RESULTO* is an auxiliary parameter: it is the empty list at the beginning of the algorithm, then it accumulates the successfully matched boxes and balls; and *RESULT* will contain the corresponding boxes and balls at the end of the successful recursive process.

The boxes and balls are taken out of the lists by predicate **take(WHAT, OUT_OF, REMAINS)**, and the selected items are checked by predicate **box.** If the test is unsuccessful, the algorithm backtracks. Otherwise predicate **append** appends the resulting sublist *[BOX, X, Y]* at the end of list *RESULTO*, and the process starts again recursively.

The recursive process stops as soon as list *BoxesIn* (and also list *BallsIn*) becomes empty, in which case predicate **out(RESULT)** displays a solution to the puzzle, or it stops when all possible cases are tested and no solution is found. (Notice the usage of predicate **append** in definition **out.**)

Built-in predicates used in the program

nl, write, =/=.

Exercises

E6.1 Notice that the order of subgoals in program clauses is chosen so that the algorithm do not perform unnecessary operations. This is not the case with the list of boxes and balls (they are listed in reverse alphabetical order), though the actual order of list elements significantly affects the performance of the program. Sort the elements of the lists to speed up the program.

E6.2 What is the difference between the definition

```
app([X:L1], L2, [X:L3]) :- app(L1, L2, L3).
app([], L2, L2).
```

and the definition of **append** given in the program? When should we use one or the other? (Be careful with the order of clauses in recursive definitions: a stopping clause at the end may be a "royal way" to infinite recursion.)

E6.3 What is the difference between predicates **take** and **append**?
Notice that we do not actually need predicate **append** in the above program, we can use predicate **take** instead. In that case, however, we obtain the resulting list in reverse order. In order to reverse the list, we can use predicate **reverse:**

```
reverse(L1, L3) :- reverse(L1, [], L3).
reverse([X:L1], L2, L3) :- reverse(L1, [X:L2], L3).
```

which is, forutnately, more efficient (linear) than the usual definition using **append** (which is quadratic):

```
rev([], []).
rev([X|L1], L3) :- rev(L1, L2), append(L2, [X], L3).
```

Rewrite the program using predicate **take** instead of **append**.

7

Once upon a time there were two integer numbers either of which was greater than one and less than a hudred. And there were two friends, Monsieur P and Monsieur S, too, either of whom was rather on the close-mouthed side. Monsieur S happend to learn the sum of the two numbers, and Monsieur P happend to learn the product of the numbers. One evening Monsieur P phoned his friend.

(1) "I don't know the two numbers," said Monsieur P.

(2) "I know you can't know them," said Monsieur S.

(3) "Now I've got them," said Monsieur P after a short while.

(4) "So ... Now I've got them, too," said Monsieur S and hung up.

That's how Messieurs P and S got the two numbers.

Find the two numbers.

7.1 Solution

Let x and y denote the two integers between 1 and 100. Obviously, if $x=a$, $y=b$ is a solution, then $x=b$, $y=a$ is also a solution. Therefore, it is sufficient to find a solution with $x \leq y$. Sentence (1) actually states that the product xy cannot be factorized uniquely. Therefore, as is easy to see, from sentence (1) it follows that

(a) both integers cannot be primes at the same time and
(b) neither of them can be a prime greater than 50.

Sentence (2) actually states that the sum $x+y$ cannot be the sum of two terms a and b such that the product ab is uniquely factorizable. Consequently, from sentence (2) it follows that

(c) $x+y$ cannot be the sum of two terms for which (a) and (b) hold.

Moreover, sentence (2) implies that

(d) $x+y < 55$, for otherwise $x+y = 53+n$ where $n \geq 2$, in which case xy might be the product $53*n$, which has a unique factorization: $x=53$, $y=n$ (c.f. (b));
(e) $x+y$ is an odd integer, since it is at least 5 and, as is easy to check, each even numbers between 3 and 55 is the sum of two prime numbers (c.f. (a) and Exercise E7.1); and
(f) $x+y-2$ cannot be a prime (c.f. (a)).

Sentence (4) actually states that there has remained a unique way to divide the sum $x+y$ into two terms. Therefore, from sentence (4) it follows that

(g) $x+y \leq 31$, for otherwise $x+y = 31+2k = 29+2(k+1)$ where k is a positive integer (c.f. also (e)), in which case sentences (1)-(3) allow x and y to be either $x=2k$, $y=31$ or $x=2(k+1)$, $y=29$.

Constraints (e), (f), and (g) imply that the possible sums $x+y$ are 11, 17, 23, 27, and 29.

Then, forming the possible sums, we have:
If $x+y=11$,
then sentences (1)-(3) allow, e.g., $x=3$, $y=8$ or $x=4$, $y=7$.
If $x+y=23$,
then sentences (1)-(3) allow, e.g., $x=7$, $y=16$ or $x=4$, $y=19$.
If $x+y=27$,
then sentences (1)-(3) allow, e.g., $x=8$, $y=19$ or $x=4$, $y=23$.
If $x+y=29$,
then sentences (1)-(3) allow, e.g., $x=13$, $y=16$ or $x=6$, $y=23$.

In all but the last cases, it is easy to see that constraint (e) excludes any other factorization of xy (e.g., $3*8=4*6=2*12$ and the sums of the factors are even in the last two cases), thus, Monsieur P can, in fact, know x and y. In the last case, $6*23=3*46=2*69$; $46+3-2$ is a prime and

69+2 > 55. Therefore, Monsieur P can know *x* and *y* in the last case, too. Monsieur S, however, cannot uniquely find the two numbers in any of the above cases.

On the other hand, if $x+y=17$, then sentences (1)-(3) allow $x=4$, $y=13$ only. The rest of the seemingly possible sums are excluded because xy has at least two (not necessarily different) odd prime factors. Thus, if xy is not divisible by 4 and if x is not equal to 2, then sentences (1) and (2) are valid for both pairs x, y and $x'=2$, $y'=xy/2$ (notice that $xy/2 < 36$ and odd), and, consequently, Monsieur P cannot say sentence (3). As for the remaining cases, pair $x=2$, $y=15$ is indistinguishable from pair $x=5$, $y=6$, pair $x=5$, $y=12$ is indistinguishable from pair $x=3$, $y=20$, and pair $x=8$, $y=9$ is indistinguishable from pair $x=3$, $y=24$ for Monsieur P. Therefore, $x=4$, $y=13$ is only case when all sentences can and do hold.

Consequently, **the two integers were 4 and 13, and 52 was given to Monsieur P, while 17 was given to Monsieur S.**

Exercise

E7.1 For a small even number greater than two it is easy to find two primes whose sum is that number. (We used this fact for even numbers less than 55 in the solution.) The general statement is known as *Goldbach's conjecture*: Every even number greater than two is the sum of two primes; moreover, every odd number greater than five is the sum of three primes. Try to prove or refute the conjecture.

7.2 Prolog program

When we read the puzzle at the first time, all we can do is realize: it is not at all easy. And when studying its solution above, we feel to be confirmed. Althoug we can translate the sentences of the conversation into arithmetical conditions in a few minutes, the actual application of our ideas requires a great deal of tedious investigation. Fortunately, we can program the arithmetical conditions as well as those parts of the solutions in the case of which we have no helpful idea. Moreover, the Prolog program obtained is rather transparent, short, and contains a fairly efficient algorithm for generating prime numbes as well.

A closer look into the program

To form the arithmetical conditions, we need the prime numbers less than 100. Therefore, at the beginning of the program, we

```
%              The Case of Messieurs S and P


dynamic(prime/1).
dynamic(sum/1).
dynamic(prod/5).
dynamic(exist/0).

start:-
   environment,
   generate_primes(1, 100),
   nl, write("At the beginning of the conversation, "), nl,
   write("the possible sums are:  5 <= SUM <= 197 "), nl,
   generate_sum,
   nl, write("After the 2nd sentence of the conversation,"),
   nl, write("the possible sums are: "), nl,
   select, nl, out,
   nl, write("After the 3rd sentence of the conversation,"),
   nl, write("the possible products are: "), nl, nl,
   products,   out1, nl, nl,
   write("After the 4th sentence of the conversation, "), nl,
   write("the possibe pairs of numbers are: "), nl, nl,
   unique,
   retractall(prime(_)),
   retractall(sum(_)),
   retractall(prod(_, _, _, _, _)).

environment:- set_state(evaluation_limit, 50000).

generate_primes(FROM, TO):-
   integer(FROM), integer(TO),
   0 < FROM, FROM < TO, TO > 4,
   primes(FROM, 4, 0, 9, TO).

generate_sum:-
   number(5, 197, N),
   assert(sum(N)), fail.
generate_sum.

number(FROM, TO, FROM).
number(FROM, TO, N):-
   M is FROM+1, M <= TO,
   number(M, TO, N).

select:-
   sum(N),  N > 54,
   retractfirst(sum(N)), fail.
select:-
   prime(P), prime(Q), P < Q,
   N is P+Q, N < 55,
   retractfirst(sum(N)), fail.
```

```
select:-
   prime(P),  Q is P*P,  Q < 100,
   N is P+Q, N < 55,
   retractfirst(sum(N)), fail.
select:-
   prime(R), prime(Q),
   P is R*Q,  P < 100,  Q2 is Q*Q,  Q2 > 100,
   N is P+Q, N < 55,
   retractfirst(sum(N)), fail.
select.

out:-
   sum(N),
   write(N), tab(2), fail.
out:- nl, nl.

products:-
   sum(SUM),
   HALF is SUM div 2,  number(2, HALF, N),
   M is SUM-N,  PROD is N*M,
   product(SUM, N, M, PROD), fail.
products.

product(SUM, N, M, PROD):-
   prod(_, _, _, PROD, MULT),   MULT1 is MULT+1,
   retractfirst(prod(_, _, _, PROD, MULT)),
   assert(prod(_, _, _, PROD, MULT1)), !.
product(SUM, N, M, PROD):-
   assert(prod(SUM, N, M, PROD, 1)).

out1:-
   prod(_, _, _, PROD, MULT),
   ( MULT == 1,  out2(PROD);
     MULT > 1,  retractfirst(prod(_, _, _, PROD, MULT)) ),
   fail.
out1:- nl.

out2(PROD):-
   ( PROD < 100, !, tab(1); true), write(PROD), tab(1).

unique:-
   prod(SUM, _, _, PROD1, 1),  prod(SUM, _, _, PROD2, 1),
   PROD1 =/= PROD2,
   retract(prod(SUM, _, _, _, _)), fail.
unique:-
   prod(SUM, N, M, PROD, 1), assert(exist),
   tab(15), write(N), write(" and "), write(M), tab(3),
   write("sum: "), write(SUM), tab(3), write("product: "),
   write(PROD), nl, nl, fail.
unique:- exist, retractall(exist).
unique:- tab(15), write("none"), nl, nl.

retractfirst(CLAUSE):- retract(CLAUSE), !.
```

```
primes(INTEG, INC, MAX, SQUARE, LIMIT):-
   INTEG1 is INTEG + INC, INTEG1 < LIMIT,
   set_test_pars(INTEG1, MAX, SQUARE, MAX1, SQUARE1, EQ),
   add_prime(INTEG1, MAX1, EQ),
   INC1 is 6 - INC,
   primes(INTEG1, INC1, MAX1, SQUARE1, LIMIT).
primes(INTEG, INC, MAX, SQUARE, LIMIT):-
   asserta(prime(3)),
   asserta(prime(2)).

set_test_pars(INTEG1, MAX, SQUARE, MAX, SQUARE, not_eq):-
   INTEG1 < SQUARE, !.
set_test_pars(INTEG1, MAX, SQUARE, MAX1, SQUARE1, EQ):-
   prime(MAX1), MAX1 > MAX, SQUARE1 is MAX1 * MAX1,
   ( INTEG1 == SQUARE, !, EQ = equal; EQ = not_eq ), !.
set_test_pars(INTEG1, MAX, SQUARE, MAX, SQUARE, EQ):-
   ( INTEG1 == SQUARE, !, EQ = equal; EQ = not_eq ), !.

add_prime(NUMBER, MAX, equal).
add_prime(NUMBER, 0, not_eq):- assert(prime(NUMBER)), !.
add_prime(NUMBER, MAX, not_eq):-
   prime(P), P <= MAX, REM is NUMBER mod P,
   ( REM =/= 0, P == MAX, assert(prime(NUMBER));
     REM == 0, true ).
```

----------- **output** -----------

```
? start.

At the beginning of the conversation,
the possible sums are:  5 <= SUM <= 197

After the 2nd sentence of the conversation,
the possible sums are:

11  17  23  27  29  35  37  41  47  53


After the 3rd sentence of the conversation,
the possible products are:

 18  24  28  52  76 112 130  50  92 110 140 152 162 170 176 182  54 100 138 154
168 190 198 204 208  96 124 174 216 234 250 276 294 304 306 160 186 232 252 270
336 340 114 148 238 288 310 348 364 378 390 400 408 414 418 172 246 280 370 442
480 496 510 522 532 540 550 552 240 282 360 430 492 520 570 592 612 630 646 660
672 682 690 696 700 702


After the 4th sentence of the conversation,
the possibe pairs of numbers are:

              4 and 13   sum: 17   product: 52

Yes
```

call predicate **generate_primes(1, 100)**, which generates the prime numbers between 1 and 100 and stores them as dynamic clauses **prime(P)**. The actual algorithm of prime number generation will be discussed later.

To start the actual puzzle solving algorithm, we have also to have the set of possible sums. Since the integers given to Messieurs P and S are distinct, the smallest possible sum is 5, while the biggest possible sum is 197. And, obviously, every integer between 5 and 197 is a possible sum. Those integers are generated and stored as dynamic clauses **sum(N)** by predicates **generate_sum** and **number(FROM, TO, N)**. That set of possible sums is then investigated and reduced according to the aricthmetical conditions.

Having read the first two sentences of the conversation, we know that the prime factorization of the product on Monsieur P is not unique and that this fact can be deduced from the sum on Monsieur S. We hope that knowing this, we can exclude the majority of the possible sums, that is, we can delete almost every clauses **sum(N)**. But what does the above paraphasis mean in terms of arithmetic? While answering this question, we present the corresponding clauses of definition **select**, which directly show the underlying considerations.

The sum cannot be too big, more precisely, it cannot exceed 54. Since if the sum *SUM* were at least 55, then it could be decomposed into *53+2+X* where *X* is a nonnegative integer, which would lead to a unique factorization of *53*(X+2)*, thus contradicting the second sentence of the conversation.

```
select :-
   sum(N), N > 54,
   retractfirst(sum(N)), fail.
```

The sum cannot be the sum of two primes.

```
select :-
   prime(P), prime(Q), P < Q,
   N is P+Q, N < 55,
   retractfirst(sum(N)), fail.
```

The sum cannot be three times a prime if the prime squared is less than 100.

```
select :-
   prime(P), Q is P*P, Q < 100,
   N is P+Q, N < 55,
   retractfirst(sum(N)), fail.
```

The sum cannot be *r+q+q*, where *r* and *q* are primes, if $r*q < 100$ and if, simultaneously, $q^2 > 100$.

```
select :-
    prime(R), prime(Q),
    P is R*Q, P < 100, Q2 is Q*Q, Q2 > 100,
    N is P+Q, N < 55,
    retractfirst(sum(N)), fail.
```

Notice predicate **retractfirst(CLAUSE)**, which deletes the first matching clause only. We use this predicate to speed up the program.

Once we have the reduced set of possible sums, we should examine the third sentence of the conversation. It means that the factorization of the product has become unique by now. In order to utilize this information, we generate the possible porducts and store them as dynamic clauses **prod(SUM, N, M, PROD, MULT)** (predicates **products** and **product**). We take each possible sum *SUM* in turn, decompose it into distinct terms *SUM=N+M* in every different way, and form the product *PROD=N*M*. To save storage, we do not record every product generated, the multiple occurrences of a value of *PROD* are recorded as the multiplicity (*MULT*) of that value. If *MULT* > 1, then only the last two arguments of clauses **prod(_, _, _, PROD, MULT)** are meaningful. Fortunately, we do not need the rest; every possible product with multiplicity grater than one is actually impossible and is deleted (predicate **out1**).

Finally, we have to consider the last sentence of the convesation, which says the decomposition of the sum into terms has also become unique by now. On this basis, predicate **unique** deletes all the non-uniquely decomposable still possible sums, using the clauses **prod** that remained, and then lists the solution to the puzzle. If there is not exactly one solution listed, it is our task to correct either the program or the puzzle.

A more efficient version of *select*

The transparency of definition **select** is excellent. Unfortunately, however, it is not efficient enough. The efficiency can be improved with a little bit of manual precomputation, taking into account the resolution strategy of Prolog, too. The new definition is **select1**.

The first and the last clauses of **select1** are the same as those of definition **select**.

As for the second clause, it is known that P, $Q \geq 2$ and $P + Q < 55$. The first inequality implies that $P < 53$, while the second one is equivalent to $Q < 55 - P$.

In the third clause, $Q = P^2$, therefore $P + P^2$ $P + Q < 55$ is equivalent to $P < 7$.

In the fourth clause, $P = R * Q$ where $Q^2 > 100$ and $P < 100$. $Q^2 > 100$ is equivalent to $Q > 10$. Moreover, $P + Q = Q(R+1) < 55$ and $R \geq 2$. All these together imply that $R \leq 6$ (or, equivalently, $R < 6$ since R is a prime) and $Q < 55/3$ (or, equivalently, $Q < 18$ since Q is a prime).

```
select1:-
   sum(N),  N > 54,
   retractfirst(sum(N)), fail.
select1:-
   prime(P), P < 53, B is 55 - P,
   prime(Q), Q < B,
   P < Q, N is P+Q,
   retractfirst(sum(N)), fail.
select1:-
   prime(P),  P < 7,
   N is P+P*P,
   retractfirst(sum(N)), fail.
select1:-
   prime(R), R < 6,
   prime(Q), 10 < Q, Q < 18,
   P is R*Q,  P < 100,
   N is P+Q, N < 55,
   retractfirst(sum(N)), fail.
select1.
```

Notice that neither of the above two definitions for **select** utilize that the primes are listed increasingly. A further step in improving efficiency would be the utilization of that fact in such a way that if, for instance, $P < 10$ is required, then the rest of the primes are abandoned, not even tried out, once a prime greater than 10 (actually, 11) is encountered. A program version obtained in this way may be more efficient, the actual effect depedns on the cost of the more complex control structures and on the amount of data left out of the search space. Unfortunately, the transparency of such a version, **select2**, would be even worse than that of **select1**, thus illustrating a general rule: such alterations for improving efficiency should be hadled with care. In many cases, a trade-off should be found between transparency and efficiency.

Prime number generation

Definitions **generate_primes**, **primes**, **set_test_pars**, and **add_prime** select the prime numbers out of the positive integers and stores them, in turn, as facts **prime(P)** in the program. The algorithm [33] essentially checks the divisibility of successive integers. 2 and 3 are primes known *a priori*, and the tested integers are obtained by incrementing alternatively by 2 and 4, thus avoiding integers divisible by

2 or 3 *ab initio*. Divisibility must be tested for prime divisors less than or equal to the square root of the integers only. To preserve the monotonicity of the sequence of primes generated, 3 and 2 are inserted at the top of definition **prime** at the end of the algorithm.

Built-in predicates used in the program

nl, write, tab, =, ==, =/=, <, >, <=, +, -, *, div, mod, integer, is, fail, true, !, ;, retract, retractall, assert, asserta, set_state.

Notice here the different equality predicates. = checks if its arguments are unifiable, and unification is performed if possible. ==, on the other hand, checks identity rather than unifiability: it yields true if and only if its arguments are identical; unification is never performed.

Exercises

E7.2 Rewrite definition **select** such that the sorted list of prime numbers be utilized. Which version is the most efficient?

E7.3 Using the prime number generator fragment of the program, write a program that lists every prime number less than a given limit.
(Hint: Do you have to store all primes to be listed?)

7.3 Pascal program

If we want to solve this shockingly tricky puzzle, we should find a suitable strategy. In doig so, we should collect what we have. Of course, we have the puzzle. And we have moderate skill in writing Pascal programs, we have a computer to run Pascal programs on, we have a reasonable amount of common sense, and we have anything but an affinity to arithmetic, especially to number theory. That being the case, we can do nothing but write a Pascal program implementing a rather naive problem solving strategy and hope it will give us a result.

The basic idea of the naive strategy we follow is the updating of tables. First, generate a table *Products* containing the products of two different integers between 1 and 100, and generate a table *Sums* containing, in some way, the sums of two different integers between 1 and 100. Those are the tables of the possible products and the possible sums, respectively. Then consider sentence (1), which implies that we have to delete all the entries in table *Products* which occur only once.

Sentence (2) implies that the entries in table Sumswhich are the sums of the factors of impossible (i.e., deleted) products are impossible. Thus, we have to delete all those entries. At this stage, however, there may remain seemingly possible products, the sums of the factors of which are no more possible. Those products are actually impossible, therefore, we have to delete them, too.

Now sentence (3) implies that all the entries that remained in table Productsand occur there more than once are also impossible; consequently, we have to delete them. As above, this deletion might make some further entries in table Sums impossible, and we have to delete those entries. Moreover, if there do appear new impossible entries in table Sums they might make some entries in table Products~51 impossible, and we have to delete those entries; etc. We have to repeat this cycle until there appear no new impossible entries. Then, if there has remained no possible product or sum, the puzzle is inconsistent. Otherwise we should investigate sentence (4).

Consider now the possible sums and decompose them, in turn, into two terms that are the factors of possible products. Sentence (4) states that there is exactly one possible sum whose decomposition in the above way is unique. If there are more than one such sums or if there is none, the puzzle is, again, inconsistent. If there is exactly one such sum, its terms are the two numbers.

```
program PandS (output, lst);
  const
    MinNum = 2;  MaxNum = 99; MaxNumMinusOne = 98;
    MinSum = 5;  MaxSum = 197;
    MinProd = 6; MaxProd = 9702;  { = MaxNum * MaxNumMinusOne }
  type
    A_Number = MinNum..MaxNum;
    A_Prod = 0..MaxProd;
    ProdType = -MaxProd..MaxProd;
    A_Sum = MinSum..MaxSum;
    S_Type = (impossible, possible);
    ProductsType = array [A_Number, A_Number] of ProdType;
    SumsType = array [A_Sum] of S_Type;
  var
    Products: ProductsType;
    Sums: SumsType;

  procedure Initialize (var Products: ProductsType;
                        var Sums: SumsType);
    { initially every combination not excluded
      by the initial constraints is possible }
  var
    i, j: A_Number;
    k: A_Sum;
  begin  { Initialize }
    for i := MinNum to MaxNum do
      for j := MinNum to i do  Products[i, j] := 0;
    for i := MinNum to MaxNumMinusOne do
      for j := i+1 to MaxNum do  Products[i, j] := -(i*j);
    for k := MinSum to MaxSum do  Sums[k] := possible
  end;  { Initialize }

procedure P_DoesNotKnowNumbers (var Products: ProductsType);
 { Delete all uniquely factorizable products }
  var
    i, j, ir, jr: A_Number;
    T_Prod: ProdType;
    Negated: Boolean;
  begin  { P_DoesNotKnowNumbers }
    { negate all the products occurring more than once }
    for i := MinNum to MaxNumMinusOne do
      begin
      for j := i+1 to MaxNum do
        begin
          T_Prod := Products[i, j];  Negated := false;
          if (T_Prod < 0) and (-T_Prod < MaxProd) then
                { entries to be checked against this value
                  remain only in this case }
          begin
            if j < MaxNum then  { check the rest of row
              for jr := j+1 to MaxNum do
                if Products[i, jr] = T_Prod then
                begin
                  Products[i, jr] := -T_Prod;
                  Negated := true
                end;
            if i < MaxNumMinusOne then
                      { check the (whole) rows below
              for ir := i+1 to MaxNumMinusOne do
                for jr := ir+1 to MaxNum do
                  if Products[ir, jr] = T_Prod then
                  begin
                    Products[ir, jr] := -T_Prod;
                    Negated := true
                  end
          end;
          if Negated then Products[i, j] := -T_Prod
        end
      end;
    { substitute zeros for the rest of the products }
    for i := MinNum to MaxNumMinusOne do
      for j := i+1 to MaxNum do
        if Products[i, j] < 0 then Products[i, j] := 0
  end;  { P_DoesNotKnowNumbers }
```

```
procedure S_HasKnownThat_P_CannotKnowThem (
                                var Products: ProductsType;
                                var Sums: SumsType);
  { the values that are the sum of i and j such that
    Products[i,j]=0 are not possible sums for S to have }
  var
    i, j: A_Number;

  procedure PossibleSums (var Sums: SumsType);
    { display possible sums }
    var
      k: A_Sum;
    begin  { PossibleSums }
      writeln(lst);
      writeln(lst, 'After the 2nd sentence of the ',
                   'conversation, the possible sums,');
      write(lst, 'which P knows, are: ');
      for k := MinSum to MaxSum do
        if Sums[k] = possible then write(lst, k:3);
      writeln(lst);  writeln(lst)
    end;  { PossibleSums }

  begin  { S_HasKnownThat_P_DoesNotKnowThem }
    for i := MinNum to MaxNumMinusOne do
      for j := i+1 to MaxNum do
        if Products[i, j] = 0 then Sums[i+j] := impossible;
    PossibleSums(Sums);
  end;  { S_HasKnowThat_P_DoesNotKnowThem }

procedure Now_P_HasGotThem (var Products: ProductsType;
                            var Sums: SumsType);
  var
    i, j, ir, jr: A_Number;
    T_Prod: ProdType;
    Deleted: Boolean;

  procedure PossibleProducts (var Products: ProductsType);
   { display possible products }
  var
    i, j: A_Number;
    Count: A_Prod;
  begin  { PossibleProducts }
    writeln(lst,
      'After the 3rd sentence of the conversation, ');
    writeln(lst,
      'the possible products, which S knows, are:');
    Count := 0;
    for i := MinNum to MaxNumMinusOne do
      for j := i+1 to MaxNum do
        if Products[i, j] > 0 then
        begin
          Count := Count + 1;
          if Count mod 20 = 1 then writeln(lst);
          write(lst, Products[i,j]:4)
        end;
      writeln(lst)
    end;  { PossibleProducts }

    begin  { Now_P_HasGotThem }
      { Products[i,j] can be a possible product
        only if i+j is a possible sum }
      for i := MinNum to MaxNumMinusOne do
        for j := i+1 to MaxNum do
  if Products[i, j] > 0 then
          if Sums[i+j] = impossible then Products[i, j] := 0;
      { all Products entries occurring more than once
        are impossible }
      for i := MinNum to MaxNumMinusOne do
        begin
        for j := i+1 to MaxNum do
          begin
            T_Prod := Products[i, j];  Deleted := false;
            if (T_Prod > 0) and (T_Prod < MaxProd) then
               { entries to be checked against this value
                 remain only in this case }
            begin
              if j < MaxNum then  { check the rest of row }
                for jr := j+1 to MaxNum do
                  if Products[i, jr] = T_Prod then
                  begin
                    Products[i, jr] := 0;
                    Deleted := true
                  end;
              if i < MaxNumMinusOne then
                             { check the (whole) rows below }
                for ir := i+1 to MaxNumMinusOne do
                  for jr := ir+1 to MaxNum do
                    if Products[ir, jr] = T_Prod then
                    begin
                      Products[ir, jr] := 0;
                      Deleted := true
                    end
            end;
            if Deleted then Products[i, j] := 0
          end
        end;
      PossibleProducts(Products)
    end;  { Now_P_HasGotThem }

  procedure Then_S_GetsThemToo (var Products: ProductsType;
                                 var Sums: SumsType);
    var
      i, j: A_Number;
      k: A_Sum;
      Count: A_Prod;
      Which: 0..1;
      ci, cj: 0..MaxNum;
  ck: 0..MaxSum;
  StillPossible, Changed, Inconsistent: Boolean;

procedure CombinationsToCheck (var Product: ProductsType;
                               var Sums: SumsType);
  { display the combination remained }
  var
    i, j: A_Number;
    k: A_Sum;
    Count, C: A_Prod;
  begin  { CombinationsToCheck }
    writeln(lst);
    writeln(lst,
     'S has the following combinations to check:');
    for k := MinSum to MaxSum do
      if Sums[k] = possible then
      begin
        Count := 0;
        for i := MinNum to MaxNumMinusOne do
          for j := i+1 to MaxNum do
            if (Products[i, j] > 0) and (i+j = k) then
            begin
              Count := Count + 1;
              Products[i, j] := -Products[i, j]
            end;
        if Count < 1 then writeln('Error in your deduction')
        else
        begin
          writeln(lst);
          write(lst,
            'If Monsieur S has ', k:0, ', then Monsieur P ');
          if Count = 1 then
          begin
            for i := MinNum to MaxNumMinusOne do
              for j := i+1 to MaxNum do
                if Products[i, j] < 0 then
                begin
                  Products[i, j] := -Products[i, j];
                  write(lst, 'has ', Products[i,j]:0);
                  writeln(lst, ' (=', i:0, '*', j:0, ').')
                end;
            writeln(lst)
          end
          else
          begin
            writeln(lst, 'can have one of the following:');
            C := 1;
            for i := MinNum to MaxNumMinusOne do
              for j := i+1 to MaxNum do
               if Products[i,j] < 0 then
               begin
                 Products[i, j] := -Products[i, j];
                 write(lst, Products[i,j]:11, ' (=', i:2,
                                               '*', j:2, ')');
                C := C + 1;
                if C mod 4 = 1 then writeln(lst)
              end;
            writeln(lst)
          end
        end
      end;
    writeln(lst);
    writeln(lst, 'A combination is a solution',
                 ' to the problem if and only if');
    writeln(lst, 'S has only one number to choose from.');
    writeln(lst)
  end;    { CombinationsToCheck }

begin  { Then_S_GetsThemToo }
  { delete sums and products that have become impossible }
  Which := 0; Changed := true;
  while Changed do
  begin
    Which := Which + 1 mod 2;  Changed := false;
    if Which = 1 then
    begin
      for k := MinSum to MaxSum do
        if Sums[k] = possible then
        begin
          StillPossible := false;
          for i := MinNum to MaxNumMinusOne do
            for j := i+1 to MaxNum do
              if (Products[i,j] > 0) and (i+j = k)
              then StillPossible := true;
          if not StillPossible then
          begin
            Sums[k] := impossible; Changed := true
          end
        end
    end
    else  { Which = 0 }
    begin
      for i := MinNum to MaxNumMinusOne do
        for j := i+1 to MaxNum do
          if Products[i, j] > 0 then
          begin
            StillPossible := false;
            for k := MinSum to MaxSum do
              if (Sums[k] = possible) and (i+j = k)
              then StillPossible := true;
            if not StillPossible then
            begin
              Products[i, j] := 0; Changed := true
            end
          end
    end
  end;
```

```
{ consistency check 1
  --inconsistent if there is no possible sum or product }
Inconsistent := true;  ck := MinSum;
while Inconsistent and (ck <= MaxSum) do
  if Sums[ck] = possible then Inconsistent := false
                         else ck := ck + 1;
if not Inconsistent then
begin
  Inconsistent := true;    ci :- MinNum - 1;
  while Inconsistent and (ci < MaxNumMinusOne) do
  begin
    ci := ci + 1;  cj := ci + 1;
    while Inconsistent and (cj <= MaxNum) do
      if Products[ci, cj] > 0 then Inconsistent := false
                              else cj := cj + 1
  end
end;
if Inconsistent then
begin
  writeln;  writeln;
  writeln('      The problem is inconsistent.');
  writeln;  writeln
end
else
begin
{ delete the sums with more than one way of their
  decomposition into terms that are factors of
  possible products }
  CombinationsToCheck(Products, Sums);
  writeln(lst); writeln;
  writeln(lst, 'The solution is given by the list below:');
  writeln('The solution is given by the list below:');
  for k := MinSum to MaxSum do
    if Sums[k] = possible then
    begin
      Count := 0;
      for i := MinNum to MaxNumMinusOne do
        for j := i+1 to MaxNum do
          if (Products[i, j] > 0) and (i+j = k) then
          begin
            Count := Count + 1;
            Products[i, j] := -Products[i, j]
          end;
      if Count < 1 then
        writeln('Error in your deduction--1')
      else
      begin
        if Count = 1 then
        begin
          for i := MinNum to MaxNumMinusOne do
            for j := i+1 to MaxNum do
              if Products[i, j] < 0 then
```

```
                  begin
                    Products[i, j] := -Products[i, j];
                    writeln(lst);
                    write(lst, 'Monsieur S has ', k:0,
                      ' (=', i:0, '+', j:0, ')');
                    write(lst, ' and Monsieur P has ',
                                          Products[i,j]:0);
                    writeln(lst, ' (=', i:0, '*', j:0, ').');
                    writeln;
                    write('Monsieur S has ', k:0, ' (=',
                                      i:0, '+', j:0, ')');
                    write(' and Monsieur P has ',
                                          Products[i,j]:0);
                    writeln(' (=', i:0, '*', j:0, ').')
                  end;
              writeln(lst); writeln
            end
            else
            begin
              for i := MinNum to MaxNumMinusOne do
                for j := i+1 to MaxNum do
                  if Products[i,j] < 0 then
                    Products[i, j] := -Products[i, j];
              Sums[k] := impossible
            end
          end
        end;
      { consistency check 2
        --inconsistent if there is no unique solution }
      Count := 0;
      for k := MinSum to MaxSum do
        if Sums[k] = possible then Count := Count + 1;
      if Count <> 1 then
      begin
        writeln;
        writeln('   All in all, the problem is inconsistent.')
      end
    end
  end;  { Then_S_GetsThemToo }

begin  { PandS }
  writeln(' Initialize ...');    { monitors action }
  Initialize(Products, Sums);
  writeln(' P does not know numbers ...');  { monitors action }
  P_DoesNotKnowNumbers(Products);
  writeln(' S has known that P cannot know them ...');
                                        { monitors action }
  S_HasKnownThat_P_CannotKnowThem(Products, Sums);
  writeln(' Now P has got them ...');  { monitors action }
  Now_P_HasGotThem(Products, Sums);
  writeln(' Then S gets them too ...');   { monitors action }
  Then_S_GetsThemToo(Products, Sums)
end.  { PandS }
```

----------- **screen output** -----------

```
 Initialize ...
 P does not know numbers ...
 S has known that P cannot know them ...
 Now P has got them ...
 Then S gets them too ...

The solution is given by the list below:

Monsieur S has 17 (=4+13) and Monsieur P has 52 (=4*13).
```

----------- **printer output** -----------

```
After the 2nd sentence of the conversation, the possible sums,
which P knows, are:  11 17 23 27 29 35 37 41 47 53

After the 3rd sentence of the conversation,
the possible products, which S knows, are:

 18  50  54  24  96 114  28  52  76  92 100 124 148 172 110 160 240 138 174 186
 246 282 112 140 154 238 280 152 168 216 232 360 162 234 252 288 130 170 190 250
 270 310 370 430 176 198 204 276 348 492 182 208 364 442 520 294 378 390 480 570
 304 336 400 496 592 306 340 408 510 612 414 522 630 418 532 646 540 660 672 550
 682 552 690 696 700 702

S has the following combinations to check:

If Monsieur S has 11, then Monsieur P can have one of the following:
         18 (= 2* 9)         24 (= 3* 8)          28 (= 4* 7)

If Monsieur S has 17, then Monsieur P has 52 (=4*13).


If Monsieur S has 23, then Monsieur P can have one of the following:
         76 (= 4*19)        112 (= 7*16)         130 (=10*13)

If Monsieur S has 27, then Monsieur P can have one of the following:
         50 (= 2*25)         92 (= 4*23)         110 (= 5*22)         140 (= 7*20)
        152 (= 8*19)        162 (= 9*18)         170 (=10*17)         176 (=11*16)
        182 (=13*14)

If Monsieur S has 29, then Monsieur P can have one of the following:
         54 (= 2*27)        100 (= 4*25)         138 (= 6*23)         154 (= 7*22)
        168 (= 8*21)        190 (=10*19)         198 (=11*18)         204 (=12*17)
        208 (=13*16)

If Monsieur S has 35, then Monsieur P can have one of the following:
         96 (= 3*32)        124 (= 4*31)         174 (= 6*29)         216 (= 8*27)
        234 (= 9*26)        250 (=10*25)         276 (=12*23)         294 (=14*21)
        304 (=16*19)        306 (=17*18)

If Monsieur S has 37, then Monsieur P can have one of the following:
        160 (= 5*32)        186 (= 6*31)         232 (= 8*29)         252 (= 9*28)
        270 (=10*27)        336 (=16*21)         340 (=17*20)
```

```
If Monsieur S has 41, then Monsieur P can have one of the following:
        114 (= 3*38)        148 (= 4*37)        238 (= 7*34)        288 (= 9*32)
        310 (=10*31)        348 (=12*29)        364 (=13*28)        378 (=14*27)
        390 (=15*26)        400 (=16*25)        408 (=17*24)        414 (=18*23)
        418 (=19*22)

If Monsieur S has 47, then Monsieur P can have one of the following:
        172 (= 4*43)        246 (= 6*41)        280 (= 7*40)        370 (=10*37)
        442 (=13*34)        480 (=15*32)        496 (=16*31)        510 (=17*30)
        522 (=18*29)        532 (=19*28)        540 (=20*27)        550 (=22*25)
        552 (=23*24)

If Monsieur S has 53, then Monsieur P can have one of the following:
        240 (= 5*48)        282 (= 6*47)        360 (= 8*45)        430 (=10*43)
        492 (=12*41)        520 (=13*40)        570 (=15*38)        592 (=16*37)
        612 (=17*36)        630 (=18*35)        646 (=19*34)        660 (=20*33)
        672 (=21*32)        682 (=22*31)        690 (=23*30)        696 (=24*29)
        700 (=25*28)        702 (=26*27)

A combination is a solution to the problem if and only if
S has only one number to choose from.


The solution is given by the list below:

Monsieur S has 17 (=4+13) and Monsieur P has 52 (=4*13).
```

A closer look into the program

Since the two numbers we are to find are distinct and are between 1 and 100, the least possible sum is 5, the least possible product is 6, the greatest possible sum is 197, and the greatest possible product is 4753. Moreover, the upper triangle part of a two-dimensional array, *Products*, is sufficient to store all possible products: an entry is the product of its indices. The possible sums can be stored in a one-dimensional array, *Sums*: an entry is either *possible* or *impossible*, meaning if its index, as the sum of the integers searched for, is possible or not at a particular stage of the solution.

Initially, every product as well as every sum is possible (**procedure Initialize**)--the products are negated just for technical reasons. These tables are updated according to the solution strategy above. Each major step is performed by an individual procedure.

procedure P_DoesNotKnowNumbers deletes, i.e., rewrites as zeros, all uniquely factorizable entries in *Products*: negates all products occurring more than once (they are the possible products) and then substitutes zeros for the rest.

procedure S_HasKnownThat_P_CannotKnowThem deletes, i.e., rewrites as *impossible*, all entries in *Sums* that have become impossible because of new impossible products. Then it calls **procedure PossibleSums**, which lists the sums possible after the second sentence of the conversation.

procedure Now_P_HasGotThem deletes all entries in *Products* that have become impossible because new impossible sums. Then it deletes all the *Products* entries occurring more than once, since they are impossible at that stage. Furthermore, it calls **procedure PossibleProducts**, which list the products possible after the third sentence of the conversation.

Finally, **procedure S_GetsThemToo** deletes, alternatively, the impossible sums and products that might have appeared. Then it checks the consistency of the puzzle and calls **procedure CombinationsToCheck**, which lists the combinations Monsieur S has to check to figure out the two numbers. In the end, a consistency check is performed again.

Consistency check is an important issue for two reasons. First, the puzzle itself may be inconsistent and therefore cannot be solved. Second, what is more likely, the solution may be incorrect or the program may contain errors, which should be detected and corrected.

Note finally that **procedures PossibleSums**, **PossibleProducts**, and **CombinationsToCheck** provide only a trace of the solution, they all may as well be omitted.

Problem solving strategies reconsidered

Essentialy, this puzzle is solved in three different ways. The first way may be called the solution of a mathematician, which requires a great deal of good ideas in order to avoid the tedious task of manual calculation. Unfortunately, there is not always a way to avoid it completely.

The third way of solution may be called the solution of a programmer. In this case, the entire problem is solved by a program written in a (not neceassarily algorithmic) programming language. The strategy is simple, and there may appear some technical difficulties, such as efficiency (the Pascal program presented is fairly slow), software or hardware limitations (if the numbers to be found were bigger, for example, between 1 and 10,000 then the memory would almost surely become too small--see also The Case of a Lot of Cans of Beer in Section 8).

The second way of solution may be called the solution of a computer scientist. In this case, the ideas and programming techniques are balanced: some almost trivial concepts and background knowledge are utilized in the program, and when they cannot help, simple programming techniques, similar to those used in the third solution, are used. This last approach seems to be the proper one in most cases, since it combines the most powerful features of human reasoning and computer programming.

Exercises

E7.4 As we have a closer look into the Pascal program, we notice that whenever the possible products or sums are updated or checked, the whole arrays *Products* and *Sums* are looked through, although we could have recorded the relevant fragments of the arrays and examined only those fragments. At first sight, this latter approach may seem to be more efficient. Is it really more efficient?

E7.5 Try to increase the upper limit of 100 for the two numbers in the puzzle. Find and program a suitable algorithm for testing the various hypotheses. What is the greatest upper limit?

8

After the maths lecture two students, Brian and David, were walking over to the bar next door. Suddenly, Brian asked his friend, "How big do you think the number of the figures in the sum of the number of the figures in the sum of the number of the figures in 4444^{4444} *is?"*

"Oh, it must be extremely big," said David.

"I don't think so," said Brian after a few steps. "O.K., look. I'll be buying that many cans of beer for you. Right now."

"No, don't kid me! You can't be so rich a guy. ... Though, anyway, why not? But don't blame me then."

"Come on, pal! I can *afford twenty bucks for that."*

"O.K. Then it's a deal," David agreed and swallowed: he could feel a XXXX coming on.

How much did it cost for Brian to by David his favorite brand?

8.1 Solution

Let *A* denote the sum of the digits in 4444^{4444}, let *B* denote the sum of the digits in *B*, and let *C* denote the sum of the digits in *B*. The value of *C* is to be found. Let's find upper limits for *A*, *B*, and *C* first.

Since $4444^{4444} < 10{,}000^{4444} < 10{,}000^{5000} = 10^{20{,}000}$, the number has at most 20,000 digits; therefore, the sum of its digits, *A*, cannot be greater than 9*20,000 < 200,000, which means that *A* has at most 6 digits. Thus, the sum of its digits, *B*, cannot exceed 9*6 = 54, which means that $C \leq 13$ since the sum of the digits in a positive integer less than or equal to 54 is maximal if the integer is 49, and 4+9 = 13.

On the other hand, it is known that the remainder in the division of a number by nine is the same as that in the division, by nine, of the sum of the digits in that number. From this it follows that 4444^{4444}, *A*, *B*, and *C* all yield the same remainder when they are divided by nine. Moreover, that remainder is 7 as follows from the argument below. $4444^{4444} = (4444^{4444} - 7^{4444}) + 7*(7^{3*1481} - 1) + 7$, since, for any positive integer *n*, $(a^n - b^n)$ is divisible by $(a-b)$, furthermore 4444-7=9*493 and 73-1=9*38 are divisible by nine.

Consequently, *C* is less than or equal to 13 and it yields a remainder of 7 when it is divided by nine. There is only one such positive integer: $C = 7$.

8.2 Prolog program

We have this puzzle in this report to illustrate the fact that there are situations when a computer cannot actually help us. Every computer has some physical limitations: the memory is limited, the numbers are represented in a particular way, which restricts the range of representable numbers, operations are performed at a certain speed, and thus certain programs might run virtually forever, ets. For the programs for previous puzzles, such limitations were not really restricting, the minor difficulties that appeared were easy to overcome (see Section 2). This is not the case with this puzzle. Now, at first sight, one can feel that 4444^{4444} is far too big an integer to be representable in an ordinary computer. But, anyway, we should try and make sure we are right.

To this end we write a program to solve the puzzle--at least theoretically. Predicate **add_up_figures(NUMBER, SUM)** returns , in *SUM*, the sum of the figures in the integer *NUMBER*. Predicate **power(BASE, EXP, POW)**, returns, in *POW*, the value of *BASE* raised to the power of *EXP*. We present two definitions for exponentiation: **power0** implements the naive algorithm of successive multiplication by *BASE* *EXP* times, while **power1** implements a more sophisticated and more efficient algorithm [15] based on the binary representation of the exponent, e.g.,

$$x^{27} = x^{16}*x^{8}*x^{2}*x = (((x^{2})^{2})^{2})^{2}*((x^{2})^{2})^{2}*x^{2}*x$$

Unfortunately, however efficient the latter algorithm is, the only result we can achieve is an even earlier overflow error message. So, we may conclude that we cannot find a way to overcome the difficulties along this track. We had better think and try to solve the puzzle in some other way.

```
%            The Case of a Lot of Cans of Beer


start:-
   nl,
   write("If A is the sum of the figures in 4444 raised to"),
   nl,
   write("the 4444th power and if B is the sum of the figures"),
   nl,
   write("in A and if C is the sum of the figures in B, then"),
   nl, nl,
   power(4444, 4444, POW),
   add_up_figures(POW, A),
   add_up_figures(A, B),
   add_up_figures(B, C),
   tab(15), write("C is "), write(C), write(".").

power(BASE, EXP, POW):- power0(BASE, EXP, POW).

add_up_figures(NUMBER, SUM):-
   integer(NUMBER), NUMBER < 0, !,
   NUM is - NUMBER, add_up(NUM, 0, SUM);
   add_up(NUMBER, 0, SUM).

add_up(0, SUM, SUM).
add_up(NUMB, ACC, SUM):-
   ACC1 is ACC + NUMB mod 10,  NUMB1 is NUMB div 10,
   add_up(NUMB1, ACC1, SUM).
```

```
%           Naive algorithm for exponentiation

power0(0, 0, POW):-
   nl, write("Error: 0 raised to the 0th power is undefined."),
   nl, abort.
power0(BASE, 1, BASE):- integer(BASE), !.
power0(0, EXP, 0):- integer(EXP), !.
power0(BASE, EXP, POW):-
   integer(EXP), EXP < 0,
   nl, write("Error: negative exponent not accepted."), nl,
   abort.
power0(BASE, EXP, POW):-
   integer(BASE), integer(EXP), power00(BASE, EXP, 1, POW).

power00(BASE, 0, POW, POW).
power00(BASE, EXP, ACC, POW):-
   ACC1 is ACC * BASE,  EXP1 is EXP - 1,
   power00(BASE, EXP1, ACC1, POW).

%     A more efficient algorithm for exponentiation

power1(0, 0, POW):-
   nl, write("Error: 0 raised to the 0th power is undefined."),
   nl, abort.
power1(BASE, 1, BASE):- integer(BASE), !.
power1(0, EXP, 0):- integer(EXP), !.
power1(BASE, EXP, POW):-
   integer(EXP), EXP < 0,
   nl, write("Error: negative exponent not accepted."), nl,
   abort.
power1(BASE, EXP, POW):-
   integer(BASE), integer(EXP), power11(BASE, EXP, 1, POW).

power11(BASE, 0, POW, POW).
power11(BASE, EXP, ACC, POW):-
   BIT is EXP mod 2,
   ( BIT == 1, !, ACC1 is ACC * BASE; ACC1 is ACC ),
   BASE1 is BASE * BASE, EXP1 is EXP div 2,
   power11(BASE1, EXP1, ACC1, POW).
```

----------- **output** -----------

```
 start.

If A is the sum of the figures in 4444 raised to
the 4444th power and if B is the sum of the figures
in A and if C is the sum of the figures in B, then


Error: overflow at _738 is 4444*4444
   > _738 is 4444*4444
trace:
```

Notice finally that the group

```
( BIT == 1, !,
  ACC1 is ACC*BASE ;
  ACC1 is ACC )
```

in the second clause of definition **power11** is of the pattern

```
( CONDITION, !, THEN ;  ELSE )
```

which declaratively means that the group is true if *CONDITION* as well as *THEN* is true or if *CONDITION* is false while *ELSE* is true. Procedurally, the group translates into the following: "If *CONDITION* is true, then evaluate *THEN* else evaluate *ELSE*. If, in either case, the evaluation yields true, then go on to the next predicate following the group, or, if the evaluation yields false, backtrack starting at the predicate immediately preceding the group." The coresponding Pascal control structure is

if *CONDITION* **then** *THEN* **else** *ELSE* **;**

Built-in predicates used in the program

nl, write, tab, integer, <, is, -/1, -/2, +, mod, div, *, ==, ;, !, abort.

8.3 Pascal programs

We have also written two Pascal programs to try to solve the puzzle. The basic idea of the first one, **CansOfBeer1**, is the same as that of the Prolog programs. The only significant difference is in that the Pascal program utilizes the floating point representation of numbers and thereby it can handle much bigger numbers. **function SumOfFigures** returns the sum of the figures in its argument; both the argument and the result are integer numbers represented as floating point numbers; *EPS* is an upper limit for the floating point rounding error.

function Power implements the same algorithm for exponentiation as predicate **power1** in the Prolog program. The base as well as the result are integer numbers represented as floating point numbers, while the exponent is a fixed point number.

The actual result of this program is, in accordance with our expectation, the same as that of the Prolog programs: an immediate overflow error message.

```
program CansOfBeer1 (output);
  const  Number = 4444.0;
         Expo   = 4444;
  var    C: real;

  function Power (Base: real; Exponent: integer): real;
    { compute the power of a real number
        using Knuth's algorithm }
    var  Negative: Boolean;
         P: real;
    begin  { Power }
      if Exponent < 0 then
      begin
        Negative := true;
        Exponent := - Exponent
      end
      else  Negative := false;
      if Exponent <> 0 then
      begin
        P := 1;
        repeat
          if Exponent mod 2 = 1 then P := P * Base;
          Base := Base * Base;
          Exponent := Exponent div 2
        until Exponent = 0;
      Power := P;
      if Negative then Power := 1.0/P
      end
      else
        if Base <> 0 then Power := 1
        else
          writeln(
          'Error: 0 raised to the 0th power is undefined.')
    end;  { Power }

  function SumOfFigures (Number: real): real;
    { compute the sum of the figures in an integer number
      represented as a real one }
    const  EPS = 0.01;  { upper limit for the floating point
                          rounding error }
    var Numb1 : real;
        FracNumb1: real;     { the fractional part of Numb1 }
        Sum : real;
    begin  { SumOfFigures }
      if Number < 0 then Number := - Number;
      if Number > 1 - EPS then
      begin
        Sum := 0.0;  Numb1 := Number;
        repeat
          Numb1 := Numb1/10.0;
          Sum := Sum + round(10.0*frac(Numb1))
        until Numb1 < 1-EPS;
      SumOfFigures := Sum
    end
    else  SumOfFigures := 0.0
  end;  { SumOfFigures }

begin  { CansOfBeer1 }
  writeln('If A is the sum of the figures in ',
          round(Number):0, ' raised to the ',
          Expo:0, 'th power');
  writeln('and if B is the sum of the figures in A');
  writeln('and if C is the sum of the figures in B, then');
  write('C is ');
  C := SumOfFigures(SumOfFigures(SumOfFigures(
                                   Power(Number,Expo))));
  if C >= (maxint + 0.5) then write(C)
                         else write(round(C):0);
  writeln(' .')
end.  { CansOfBeer1 }
```

----------- **output** -----------

```
If A is the sum of the figures in 4444 raised to the 4444th power
and if B is the sum of the figures in A
and if C is the sum of the figures in B, then
C is
Run-time error 01, PC=2D43
Program aborted

Searching
  21 lines

Run-time error position found. Press <ESC>
```

Despite the limitations and aborted trials above, it *is* possible to solve the puzzle via a computer program, though the solution is tedious and, in general, not smart. Program **CansOfBeer2** represents a further step toward the solution of the puzzle. The basic idea is the representation of integers as strings (actually arrays) of digits; the operations simulate the process of manual calculation (c.f. [33]). An integer is represented as an array of its digits, indexed from right to left, and the length of the digit string, which is a fixed point number. As an exception, the exponent is represented as a fixed point nonnegative integer (cardinal). **procedure AddUpLong** computes the sum of the digits of an integer, while **procedure Multiply** multiplies two integers and returns the product. It first generates the lines of partial products (the lines are indexed from bottom up), then adds up the lines to obtain the product (see the example below). **procedure PowerLong** implements the second algorithm for exponentiation.

Example

```
  543 * 987     Fact1=[3,4,5]; Fact2=[7,8,9]; LenFac1=LenFac2=3
488700          Line 3,              Offset=2
043440          Line 2,              Offset=1
003801          Line 1,              Offset=0
535941          Res=[1,4,9,5,3,5];   LenRes=6
```

```
program CansOfBeer2 (output);
  const Number =   4444;
        Exponent = 4444;
        MaxFig = 100;     { maximum length of digit strings }
  type Length = 0..MaxFig;
       OneLine = array [1..MaxFig] of 0..9;
                 { digit string representation of cardinals }
       Card = 0..maxint;
  var  A, B, C, Base, Power: OneLine;
       LenA, LenB, LenC, LenBase, LenPow, i: Length;
       Temp: Card;

  procedure Multiply (var Fact1: OneLine; var LenFac1: Length;
                      var Fact2: OneLine; var LenFac2: Length;
                      var Res:   OneLine; var LenRes:  Length);
    { compute  Fact1*Fact2 as a string of digits }
    var Carry, temp: Card;
        Offset, B, i, j, k, TotalLength: Length;
        Lines: array [1..MaxFig] of OneLine;
                                    { lines in multiplication }
            { TotalLength is the maximum length of lines }
    begin { Multiply }
      if ((LenFac1=1) and (Fact1[1]=0)) or
         ((LenFac2=1) and (Fact2[1]=0))
      then   begin  LenRes := 1; Res[1] := 0  end
      else
      if (LenFac1=1) and (Fact1[1]=1) then
      begin
        LenRes := LenFac2;
        for k := 1 to LenFac2 do Res[k] := Fact2[k]
      end
      else
      if (LenFac2=1) and (Fact2[1]=1) then
      begin
        LenRes := LenFac1;
        for k := 1 to LenFac1 do Res[k] := Fact1[k]
      end
      else
```

```
begin
{ generate lines }
TotalLength := LenFac1 + LenFac2 - 1;
for i := LenFac2 downto 1 do
begin
  Carry := 0;
  Offset := i-1;
  for k := 1 to Offset do  Lines [i, k] := 0;
  for j := 1 to LenFac1 do
  begin
    temp := Fact1[j] * Fact2[i] + Carry;
    if temp > 9 then
    begin
      Carry := temp div 10;
      Lines[i, Offset+j] := temp mod 10
    end
    else
    begin
      Carry := 0;
      Lines[i, Offset+j] := temp
    end
  end;
  if i = LenFac2 then
    if temp > 9 then
    begin
      TotalLength := TotalLength + 1;
      Lines[i, TotalLength] := Carry
    end;
  if i < LenFac2 then
  begin
    if temp > 9 then
    begin
      B := Offset + LenFac1 + 1;
      Lines[i, B] := Carry
    end
    else B := Offset + LenFac1;
    for k := B+1 to TotalLength do Lines[i, k] := 0
  end
end;
{ add up lines }
LenRes := TotalLength;  Carry := 0;
for k := 1 to TotalLength do
begin
  temp := Carry;
  for i := 1 to LenFac2 do temp := temp + Lines[i, k];
     { temp is assumed to be a representable integer }
  if temp > 9 then
  begin
    Carry := temp div 10;
    Res[k] := temp mod 10
  end
  else
  begin
    Carry := 0;
    Res[k] := temp
  end
end;
```

```
    while Carry > 0 do
    begin
      LenRes := LenRes + 1;
      Res[LenRes] := Carry mod 10;
      Carry := Carry div 10
    end
    end
  end;  { Multiply }

procedure PowerLong (Base: OneLine; LenBase: Length;
                     Exponent: Card;
                     var Power: OneLine; var LenPow: Length);
  { compute Base**Power as a string of digits }
  var i, LenRes: Length;
      Res: OneLine;
  begin  { PowerLong }
    if (Exponent > 0) and (Exponent <> 1) then
    begin
      LenPow := 1;  Power[1] := 1;
      repeat
        if Exponent mod 2 = 1 then
          if (LenPow = 1) and (Power[1]=1) then
          begin
            LenPow := LenBase;
            for i := 1 to LenBase do Power[i] := Base[i]
          end
          else
          begin
            Multiply(Power, LenPow, Base, LenBase,
                                                Res, LenRes);
            LenPow := LenRes;
            for i := 1 to LenRes do Power[i] := Res[i]
          end;
        Multiply(Base, LenBase, Base, LenBase, Res, LenRes);
        LenBase := LenRes;
        for i := 1 to LenRes do Base[i] := Res[i];
        Exponent := Exponent div 2
      until Exponent = 0
    end
    else
    case Exponent of
      1: begin
           LenPow := LenBase;
           for i := 1 to LenBase do Power[i] := Base[i]
         end;
      0: if (LenBase = 1) and (Base[1]=0) then
         writeln(
          'Error: 0 raised to the 0th power is undefined.')
         else
         begin
           LenPow := 1;  Power[1] := 1;
         end;
      else writeln('Error: negative exponent not accepted.')
    end
  end;  { PowerLong }
```

```
procedure AddUpLong (var Num: OneLine; var LenNum: Length;
                     var Sum: OneLine; var LenSum: Length);
 {compute the sum of the digits in Num as a string of digits}
  var  Carry, temp: Card;
       i, j: Length;
  begin  { AddUpLong }
    Sum[1] := 0;  LenSum := 1;
    for i := 1 to LenNum do
    begin
      temp := Num[i] + Sum[1];
      if temp < 10 then Sum[1] := temp
      else
      begin
        Carry := 1;
        Sum[1] := temp - 10;
        j := 1;
        while Carry > 0 do
        begin
          j := j + 1;
          if LenSum < j then
          begin
            LenSum := j;
            Sum[j] := 0
          end;
          temp := Sum[j] + Carry;
          if temp < 10 then
          begin
            Sum[j] := temp;
            Carry := 0
          end
          else  Sum[j] := temp - 10
        end
      end
    end
  end;  { AddUpLong }

begin  { CansOfBeer2 }
  writeln('If A is the sum of the figures in ', Number:0,
          ' raised to the ', Exponent:0, 'th power');
  writeln('and if B is the sum of the figures in A');
  writeln('and if C is the sum of the figures in B, then');
  write('C is ');
  {  produce Base  }
  LenBase := 0;  Temp := Number;
  while Temp > 0 do
  begin
    LenBase := LenBase + 1;
    Base[LenBase] := Temp mod 10;
    Temp := Temp div 10
  end;
```

```
    { solve puzzle }
    PowerLong(Base, LenBase, Exponent, Power, LenPow);
    AddUpLong(Power, LenPow, A, LenA);
    AddUpLong(A, LenA, B, LenB);
    AddUpLong(B, LenB, C, LenC);
    for i := LenC downto 1 do write(C[i]:0);
    writeln(' .')
  end. { CansOfBeer2 }
```

----------- **output** -----------

```
If A is the sum of the figures in 4444 raised to the 4444th power
and if B is the sum of the figures in A
and if C is the sum of the figures in B, then
C is

Memory allocation error
Cannot load COMMAND, system halted
```

Unfortunately, in Pascal the size of every array needs to be known at the beginning of the program, i.e., dynamic or flexible arrays are not allowed. As a consequence, arrays have to be declared at maxial expected length, thus consuming a huge amount of memory. Moreover, an array cannot have more than 2**maxint*+1 entries (the maximum range of indices is [-*maxint*..*maxint*]). That is why program **CansOfBeer2** is only a step toward the solution rather than a solution itself.

Exercises

E8.1 To overcome the last mentioned difficulties of Pascal programs, rewrite program **CansOfBeer2** using chained lists of records standing for digits instead of arrays of digits. (Notice the similarity to list structures in Prolog.) Can we solve the puzzle in this way? If we can, at what cost?

E8.2 Consider the technique used in program **CansOfBeer2** and try to declare sufficiently large arrays. If the memory proves to be too small, the sizes of the arrays have to be reduced. The shortened arrays, however, may not be able to actually represent the big integers we want to use. In this case we can use several such arrays to represent a big integer, and, almost surely, we have to use secondary storage as well. Going on in this direction, we can create a(n implementation dependent) program to solve the puzzle. Write such a program.

Concluding Remarks

This report contains only a subset of the puzzles we solved via logic programming; the complete set of 25 puzzles is to be published as a book. The 8 puzzles included in the report are selected to represent the diversity of the puzzles we found.

In *Introduction* we posed the question of whether and to what extent logic programming is adequate for solving logic puzzles. Now it is time we answered, and our answer is a definite *yes* for the first part of the question. We by no means want to state, however, that logic programming always provides the only and the best method for solving a logic puzzle. That is why we presented three kinds of solution, a mathematical solution, a solution via a Prolog program, and a solution via a Pascal program, and compared them to one another. And we have found that, in certain cases, one kind of solution is better than the others, and that one may be any one of the three kinds, while, in other cases, there actually is only a slight difference between the solution techniques.

Each Prolog program presented in this report is short, compact, easily understandable and modifiable, and transparent. They are transparent, since the program source texts closely follow the texts of the puzzles. These are inevitable advantages of logic programs. As one can see, on the other hand, the algorithmic approach has its own advantages, especially in terms of efficiency. When programming in an algorithmic language, one finds it quite natural to be aware of inefficiency. When, however, programming in a declarative language, one tends to utilize and emphasize the declarative features of the language in order to write as transparent programs as possible. And he does so despite the fact that the sign "Beware of inefficiency" is none the less adequate for programming in a declarative language such as Prolog. Some point that considerably affect the efficiency of Prolog programs are the order of clauses in definitions, the top-down left-to-right search strategy, the proper use of cuts, etc. These all can be utilized to reduce the actual search space. (For further points concerning efficiency, see, e.g., [7], [28], [30].) We do think that a basic characteristic of a good Prolog program is a suitable ballance between declarative and procedural features.

The programs for solving certain puzzles contain subprograms that are independent of the particular puzzles and solve some general task, such as prime number generation, exponentiation, list concatenation, finding all solutions, etc. For the sake of generality, those subprograms may be more complex than they should necessarily be in that particular environment. In exchange for that complexity, we have gained the portability of the subprograms: the suitable program fragments can be transplanted into other programs almost directly.

The *Case of a Jealous Boyfriend* examplifies the case when it is considerably easier to solve the problem via a Prolog program than via a program written in a "conventional" algorithmic language such as Pascal. *The Case of a Forgotten Phone Number* shows, on the other hand, that the same algorithm may be implemented both in Prolog and Pascal quite naturally. The Pascal program works a bit faster. *The Case of Messieurs P and S* is soved following two different problem solving strategies. In order to solve a problem, we can follow the most naive approach: we rely on the computer algortihm to the greatest extent and do not care about the implementation difficulties. The Pascal program implements such a naive approach. The Prolog program, on the other hand, incorporates the results of some simple mathematical (number theoretical) considerations, and uses a naive algorithm only if there are no simple mathematical ideas to help and thus the naive approach is appropriate. This combined approach has made the Prolog program be much "cleverer," more compact, and rather efficient.

The combined approach for the solution of the puzzle is a piece of the output of a more general consideration: Why do we solve puzzles at all? Why do we solve puzzles via computer programs in particular? When is a problem worth being solved via a program? Once we have decided we write a program, how much should we think before writing it, when should we use naive mechanical algorithms? Once we have to think before writing a program anyway, when is it worth bothering with actually writing it? The answer to the first question is simple: Each puzzle is a new challenge to our mind. And we are eager for testing our capability and finding a solution. That is why we solve puzzles via programs as well. In that case, we test our programming skill, too. The questions remained are much harder. Before solving a problem in any way, we should find estimates for the effort required by the solution via reasoning, and for that required by the solution via a program written in one or another particular programming language (in general, the effort is not the same for different programming languages). Once such estimates are found, we should choose the one with the least effort. Finding the way of solution with minimum effort is the main criterion for deciding on the suitable portion of mathematical or other "external" ideas incorporated into the programs and on the amount and places of naive mechanical algorithms. The solutions of *The Case of a Jealous Boyfriend, The Case of a Circle in a Desert*, and *The Case of Messieurs P and S* are examples for the application of "the principle of minimum effort."

The solutions of *The Case of the Bridges in Koenigsberg* and *The Case of some Color Boxes and Balls* examplify the fact that, provided that it is not too hard to write the program, it is useful to write a program for solving a problem in order to check if the solution via reasoning is correct. The problem solving methods, based solely on simple mechanical algorithms, of the programs for solving the last two puzzles above essentially differ from those used in the "mathematical" solutions. Therefore, the identity of the results supports the correctness of both kinds of solution.

In contrast to the relation between the problem solving method used in the mathematical solutions and in the Prolog programs for the last two puzzles above, in *The Case of Three Gods* the Prolog program follows, step-by-step, the track of human reasoning. The puzzle itself is rather simple, and so is the program. One might as well say there is no need for the program--and he would be right. The porgram, however, will not be considerably more difficult to understand even when the number of constraints and, therefore, the number of relations to be checked are increased, while the human brain can handle only a couple of constraints reliably and effectively.

Computers can help us a great deal in solving a variety of tasks; and logic programming augments the diversity of usable techniques. But there is no computer that could be a substitute for human brain. There are problems that, virtually, cannot be solved via programs because of the hardware or software limitations or the speed of execution. This situation is examplified by *The Case of a Lot of Cans of Beer*. One, however, must not give up even in such a case, he should try to find other ways of solutions. Eventually, he just cannot avoid thinking.

Appendix A. Impelmentation Problems

MROLOG provided us with a minimal set of DEC-10 Prolog built-in predicates [27]. Unfortunately, there are some useful DEC-10 Prolog built-in perdicates not directly supported by MPROLOG. The most important of those is **retractall(P).** Since we need this predicate frequently, we have extended the DEC-10 Prolog supporting module by adding the following clause:

```
retractall(P) :- retract(P), fail;  true.
```

On the other hand, MPROLOG has a couple of useful built-in predicates with no equivalents in DEC-10 Prolog. One of those is **del_statement(P)**, which deletes the first matching clause and fails on backtracking. Another interesting and very useful feature of MPROLOG is the *temporary assertion and retraction* of clauses: the effect is temporary since the assertion or retraction is undone on backtracking. Whenever such features are needed, they are explicitly incorporated into the porgrams:

```
retractfirst(P) :- retract(P), !.
```

For temporary predicates, see Section 3.

Upon reading the Prolog source texts in this report, the reader will find two MPROLOG specific features: the declaration **dynamic** and the environment parameter setting **set_state** and **evaluation_limit.** They are compulsory items of syntax and should be omitted or changed in other Prolog dialects. In order to assert or retract a program clause in MPROLOG, its definition must be declared as **dynamic. evaluation_limit** is an environment parameter with a default value of 10,000. If the default value proves to be too small, the value of the parameter should be set with predicate **set_state**, for instance,

```
set_state(evaluation_limit, 20000).
```

As for the Pascal programs in this report, the nonstandard features of Turbo Pascal are rarely used. The most important nonstandard feature used is the *string type*. The reason is: strings are much more convenient to use than packed arrays of characters. **keypressed** is a Boolean built-in function, which yields *true* if and only if a key is pressed on the keyboard. Built-in functions **assign(lfn, pfn),** reset(lfn), **rewrite(lfn)**, and **close(lfn)** are to handle external files, while compiler directives *{$I-}* and *{$I+}* and standard identifier *IOresult* enable one to control input/output error handling. If one wishes to run the Pascal programs presented in this report in another dialect of Pascal, he can easily, systematically rewrite every Turbo Pascal specific feature used.

Appendix B. Sources

The puzzles discussed in this report originate from various books and problem books. The list below gives the place of origin of each puzzle. The number in brackets after a puzzle refers to the source listed in *References*.

The Case of a Jealous Boyfriend [5]
The Case of a Forgotten Phone Number [6]
The Case of Three Gods [23]
The Case of a Circle in a Desert [6]
The Case of the Bridges in Koenigsberg [1]
The Case of some Color Boxes and Balls [5]
The Case of Messieurs S and P [2]
The Case of a Lot of Cans of Beer [20]

Note that the puzzles in this report are not exactly the same as those in the sources. The original problems are often tailored to suit to the subject.

Appendix C. Further Puzzles

As we noted in *Introduction*, we have collected a number of puzzles and are going to publish a more complete set in another report or in a book; this report is only an extract of the complete work. In this appendix we present the texts of the puzzles solved in the complete version. The numbers in brackets after the titles refer to the places of origin listed in *References*.

1 The Case of a Broken Window [5]

One afternoon four boys, Alex, Birt, Clive, and Dick, played football in the middle of a downtown road. As a result of a big kick, the ball hit a window and broke it. Soon the tenants arrived and began to look into the matter. They asked questions, and the boys gradually told them the whole story:

Alex said, "(1) It's not me who kicked the ball then. (2) It was Dick's idea to play here. (3) Clive's innocent."

Birt said, "(4) I doesn't break no bloody window. (5) Clive did it. (6) Can play football a lot better than Dick."

Clive said, "(7) It ain't my fault. (8) If I'd known it ended in that, I wouldn't've begun to play here with them. (9) It ain't got nothin' to do with Alex."

And Dick said, "(10) Did do no harm to that window. (11) 'Twas Clive. (12) When I came here, they were already playin'."

Of course, the tenants noticed the boys did not always tell the truth; so they kept on asking questions. They found out later that, as far as the above answers were concerned, each boy told them exactly one lie.

Which boy broke the window?

2 The Case of a Fooled Trainer [6]

Al, Bill, Charlie, Dan, and Ed are members of a swimming club. Once, while their trainer was off, they held a sort of a competition among themselves. When the trainer came back and *asked about the result of the competition, they gave him answers as follows.*

Al: "(1) Dan was placed second and (2) I was placed third."

Bill: "(3) I was the best and (4) Charlie was the next."

Charlie: "(5) I was the third while (6) Bill was the last.

Dan: "(7) I was placed second and (8) Ed was placed fourth."

Ed: "(9) I managed to beat just one guy. (10) Al won."

Seeing the trainer's confused face, they admitted, "You're right. We've tried to kid you: one of the two statements of each of us is true, while the other is false. OK, and there is none of us tied for any place; that's for sure. But that's enough. It's your turn now."

Then the trainer began to think and tried to find out the result of the competition. Let's help him.

3 The Case of a Greyhound Race [5]

When I visited Huckleberry City last summer, I was strongly advised not to miss the Huckleberry Greyhound Race. Although I was not familiar with the dogs, I wanted to make some bets--just for fun, of course. So I bought tips: they were really cheap. The first tipster told me, "Believe me, Sir, Arctic Beam will win; it's the best hound I've ever seen. Biddable'll be placed second, Castle Warden third, Diamond fourth, End of Era fifth, and Foot Patrol'll be placed sixth. That's the best and the cheapest tip, believe me, Sir." The second tipster said something else: 1st: End of Era, 2nd: Diamond, 3rd: Biddable, 4th: Castle Warden, 5th: Foot Patrol, and 6th: Arctic Beam; and that tip was a bit more expensive. I was happy with the tips of the professionals--right up till the end of the race, when it turned out that either of them had guessed exactly three places only, and I lost all my bet. 'Damn it,' I thought, 'I should've given tips instead, it's much more profitable.'

Do you already know the result of the Huckleberry Greyhound Race?

4 The Case of a Horse Race [26]

Last Sunday afternoon two friends, Mark and Ron, went to the racetrack to watch the King Cup, the most spectacular event in the season. As soon as they arrived, they went to the paddock to have a look at the horses. Then they made bets on the first five places in the first race with a bookie: Mark thought Asmid, the black stallion, would win, British Hero would be placed second, Carnival would be placed third, Donnal Deux would be placed fourth, and Estralita would be placed fifth; while Ron guessed Donnal Deux would be placed first, Asmid would be the runner-up, Estralita would be the third, Carnival would be the fourth, and British Hero would be placed fifth.

The result of the first race showed that neither of them won:
(1) There was no horse at its actual place in Mark's guessed result;
(2) he could not even guess the actual order in any pairs of horses one after the other.
Ron's guess was much closer to reality:
(3) he guessed the actual places of two horses; and
(4) he guessed the actual order in two pairs of horses one after the other.

What was the result, as far as the first five places were concerned, of the first race at the King Cup?

5 The Case of 100 Tricky Statements [19]

The following 100 statements are written on a sheet of paper:

1 Exactly one statement is false on this piece of paper.
2 Exatcly two statements are false on this piece of paper.

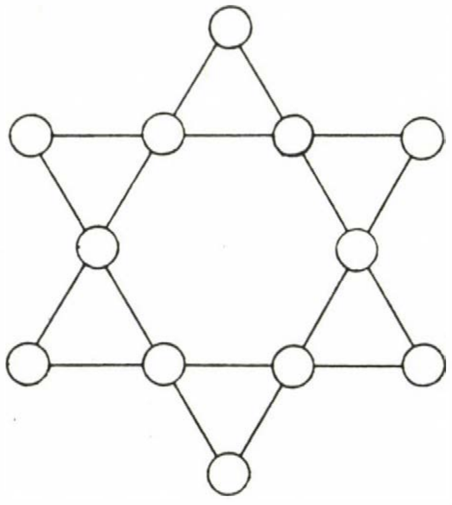
. . .

99 Exactly 99 statements are false on this piece of paper.
100 Exactly 100 statements are false on this piece of paper.

Which of these statements are true and which are not?

6 The Case of a Magic Star [29]

Put the first dozen positive integers into the circles in the figure below such that the sum of the four numbers along each segment as well as that of the six numbers at the vertices be equal to 26. How can you do that?



7 The Case of an Unwiped Blackboard [3]

When the pupils entered their classroom one morning, they found a perfect mess inside. The most disgusting thing of all was the writing on the blackboard: it was nothing but a scribble. "Don't wipe it. It must be the message of an E.T.," said Jerry, a would-be sci-fi writer, who managed to find some pattern on the blackboard.

The whole class began to study the lines, curves, letters, and numbers. But all they managed to figure out was an arithmetic puzzle written most likely in Latin. Unfortunately, there was no one among them who could read Latin. They found, however, a lot of words in the text of the puzzle which were very similar to certain English words. Thus, in spite of the strange patter used, they "deciphered" the puzzle, which read as follows.

In the multiplication below, the letters stand for decimal digits--distinct letters for distinct digits. A dot stands for any decimal digit. How much is *ABC* ?

```
  ABC * ABC
...H
 C.BH
  .EFC
...FFC
```

They tried to solve the problem. After a few minutes Jerry said, "OK, I got it. But what on earth could that mean?"

What was the "extraterrestrial message?"

8 The Case of a Royal Parade [10], [14], [20]

At the end of the 18th century, there was to be a royal parade at the court of Czarina Yekatyerina II in St. Petersburg. In order to produce a spectacular event as a part of the parade, thirty-six officers of six different ranks were taken from six different regiments, one of each rank in each regiment. Those thirty-six officers were to be arranged in a solid sqare formation of six by six such that each row and each column contains one and only one officer of each rank and one and only one officer from each regiment. Because the regiments were selected so that each of them had a unique colorful uniform, the square formation was going to be a worthy one for the royal sight. But was it possible to arrange the thirty-six officers in that formation?

9 The Case of a Swimming Championship [5]

It was spring, the time of the Local Swimming Championship. Every teenaged boy in the small town ought to take part in such an event--a cup in the cupboard or a medal on the wall was always a most effective, self-explanatory intoroduction to the beloved girl. And, of course, all the girsl in the town went to watch the swimmers. All but poor Pru, who had caught flu and had to stay in bed. Fortunately, her friend, Sue, ran over to her place after the semifinals and told her the names of the boys who would compete in the final.

"What do you guess the result will be?" Sue asked.

"Well, I don't know. Alf may be placed first, Bob second, Cliff third, Daryl fourth, Eric fifth, and, perhaps, Fred sixth and George seventh," said Pru. However she wished, she dare not have mentioned her loved one, Bob, at the first place.

"A bit of superstition, isn't it? I'll come back after the final and tell you the result," said Sue and off she went.

When she returned, she told Pru, "I've changed my mind: I've decided not to tell you the result directly. I'm gonna give you some hints instead.

(1) "Your guess isn't correct: there's no one at his place.

(2) "Not even at a neighoring place of his actual place.

(3) "And not even at a second neighboring place of his actual place.

(4) "Yea, and I should tell you that if you wanna alter your guessed result in order to get the real result, you have to move more guys forward than backward.

(5) "Well, and there was no tie. That's egough, I guess. Now I'm leaving you; I'm in a hurry as always. See you later," said Sue and left.

Is the above hint really enough for Pru to figure out the result of the championship?

10 The Case of Five Lottery Numbers*

"Do you know the numbers drawn?" a mathematician asked his friend, who was a mathematician, too, after a lottery draw.

(1) "They are very funny. Really funny. There is one among them which divides the sum of any two of the numbers drawn."

"And what's that number?"

(2) "I won't tell you that. If I did, you'd find out every winning number."

* A problem of the Daniel Arany Highschool Competition in Mathematics, Hungary, 1986

(3) "Tell me then, at least, if that number is even or odd." On hearing the answer, the first guy jumped up, "Boy! Won a first division!"

What were the winning numbers?

(According to the rules of the Hungarian lottery, five numbers are drawn out of ninety, more precisely, out of the numbers 1, 2, ..., 90.)

11 The Case of Nine Different Bottles [11]

A store sells three kinds of alcoholic drinks: wine, beer, and brandy; French, German, and Hungarian made drinks of each kind. In order to represent the diversity of alcoholic drinks on stock, the window-dresser took nine bottles of the three kinds of drinks, one of each made in each kind, and was going to arrange them into a square formation of three rows and three columns such that each row and each coulmn contains exactly one bottle of drink of each made and exactly one bottle of drink of each kind. How many different ways could he find to do that?

12 The Case of Some Walking Girls [2]

Penthouse Pansion has changed recently. So have the girls. It is now one of the most expensive girls college in the state. Mademoiselle Spinstaire, the schoolmaster, is willing to admit girls of the most prominent families only. The afternoon walks on Wednesdays are guided by Mademoiselle Spinstraire herself--and by her inevitable pink umbrella, of course. The girls, in their pretty uniforms and with their yellow straw hats on, walk side by side, three girls a row. Madmoiselle Spinstaire is rigorous; she does know how to spoil even the afternoon walks of the girls.

"I do not like those little gangs of yours," she said last Wednesday. "As for the forthcoming weeks, rearrange your rows such that no one walk with either of her present rowmates on the same row until one walks with everybody else on the same row."

The girls had found the task too difficult, so Madmoiselle Spinstaire herself had to rearrange the girls. For the form of nine girls, she soon found distinct *arrangements for four consecutive weeks, and she could prove that there was no more distinct arrangements.*

For the form of fifteen girls, she has proved that there could be distinct arrangements for at most seven consecutive weeks, but she has not yet found the actual arrangements. And now it's 1 p.m. Wednesday. What a shame!

Let's check the reasoning of Madmoiselle Spinstaire and generate the arrangements for both forms.

13 The Case of Some Zeros [12]

Can we put down the zeros at the end of 1000! on a single page if we can write 32 lines a page and 60 letters or figures a line?

(If n *is positive integer, then* n! *is, by definition, equal to the product of* 1*2*...*(n-1)*n.*)*

14 The Case of the Ancestors' Ancestors [11]

Are my grandfathers' great-grandfathers the same persons as my great-grandfathers' grandfathers? (Suppose there was no marriage between relatives in the last five generations.)

15 The Case of Three Boys [22]

Two mathematicians, who have not seen one another for quite a while, is talking during the tea break of a congress.

"Well, and how old are your children?"

"You know what? Remember the old days at the Uni, don't you? Well, I'm giving you a puzzle instead of an answer again. OK?

(1) "Right. I've got three kids.

(2) "Multiply the numbers of years they have lived so far--don't bother with the fractions--and you'll get 36.

(3) "And now add up those numbers and you'll have ... Look! You'll have the sum of the windows of that orange house opposite."

After a short while the other said,

(4) "Give me some more hint. It's not enough to find out the ages of your kids."

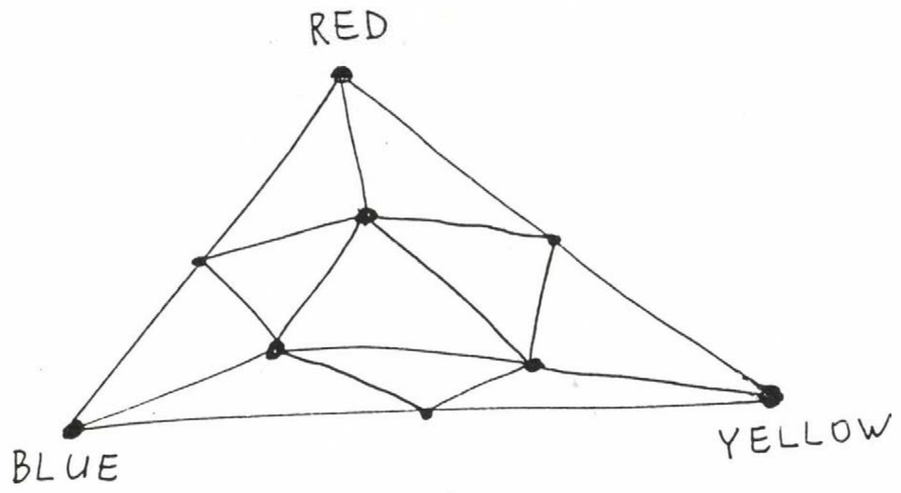
(5) "Oh, sorry. You're right. I should've told you that the youngest one doesn't like dark chocolate."

"Thanks. That's more than enough. You have a ... "

Finish the last sentence, that is, figure out how old the children are.

16 The Case of Triangle Coloring [18]

As soon as Steve, the 12-year-old and the naughtiest son of Mr. and Mrs. Tryangel, arrived home from school, he dashed into his room and, after a few seconds, appeared again with a handful of color crayons. He took a black one and scribbled the figure below on the wallpaper. Having finished his work, he called, "Gill, come on in here. I'll show you somethin', a colorful little triangle."



When Gill, his sister, entered the room, Steve gave her some of his crayons and said, "I'll give you three crayons, a blue, a red, and a yellow one. And I'll tell you how to color the triangles. Right? Put a blue, a red, and a yellow spot at the corners of that big triangle. Now put a spot at the middle of each side of that big triangle, there, there, and there, so that each spot is of the same color as one of the endspot of that side. No, you don't know nothin'! That spot must be either blue or yellow 'cause there is a blue and a yellow spot at the ends of that side. Got it? And now you have that small triangle in the middle with no spots at the corners. Now I'll turn away and you'll color the corners as *you like. Hey, don't start yet. Listen. I'll come back when you finish, and if I can find a triangle with a blue, a red, and a yellow spot at the corners, then you'll give me the lolly you got last night. Promise? You still have it, don't you? And if I can't find such a triangle, then I give my lolly to you. OK, just hurry up, Dad's comin' soon."*

Who got the lollipop?

17 The Case of Two Noisy Ghosts [17]

It was last spring when old MacDonald died leaving all his possessions and debts for his nephew, Hamish. That is how my friend inherited a superb Highland castle. Unfortunately, it was not only the castle he inherited: the castle was haunted by two noisy ghosts, as turned out in the very first night after he had moved in. Since then, from midnight till dawn each night, he has been haunted by ghostly noises: a mysterious singing and a vulgar laughter. Those were the first two ghosts in Hamish' life, so he did not really know what to do. After a while he started to observe them, and realized certain regularity in their behavior.

- Whenever he plays the organ and there is no laughter, the singing ghost changes her activity to the opposite in the next minute (that is, if she was singing, she stops singing, or if she was silent, she starts to sing). Otherwise, the singing ghost does, in each minute, what she did in the preceding one.

- Whenever the window is closed, the laughing ghost does what the singing ghost did in the preceding minute (that is, she laughs if the other sang, or she is silent if the singer was silent).

- Whenever the window is open, however, the laughing ghost does the opposite of what the other did in the preceding minute.

And now, with the results of his remarkable observation, Hamish has come to me and wants to know by what manipulations he can get rid of the ghost. What should I tell him?

18 The Case of Two Numbers [26]

I have found two numbers, a three-digit one and a two-digit one. If you divide the three-digit one by the two-digit one, the quotient will be a number equal to the sum of the digits of the divisor and the remainder will be a two-digit number consisting of the digits of the divisor in reverse order. If you multiply the remainder by the quotient and then increment that result by the divisor, the three-digit number you will get will consist of the digits of the dividend in reverse order. Guess the numbers I have found.

References

[1] **Andrasfai, B.,** *An Introduction into Graph Theory.* Tankonyvkiado, Budapest, 1973 (in Hungarian).

[2] **Arsac, J.,** *Jeux et casse-tête a programmer. (Games and Puzzles for Programmers.)* Dunod, Paris, 1985 (in French).

[3] **Bakos, T., P. Lorincz, & G. Tusnady** (eds.), *Highschool Competitions in Mathematics -- 1970.* Tankonyvkiado, Budapest, 1970 (in Hungarian).

[4] **Battani, G. & H. Meloni,** *Interpreteur du Language de Programmation Prolog.* Groupe de I.A., UER Luminy, Université d'Aix-Marseille, 1973.

[5] **Bizam, G. & J. Herczeg,** *Games and Logic in 85 Problems.* Muszaki Konyvkiado, Budapest, 1972 (in Hungarian).

[6] **Bizam, G. & J. Herczeg,** *Colorful Logic.* Muszaki Konyvkiado, Budapest, 1975 (in Hungarian).

[7] **Bratko, I.,** *Prolog Programming for Artificial Intelligence.* Addison-Wesley, Reading, Mass., 1986.

[8] **Clocksin, W. F. & C. S. Mellish,** *Programming in Prolog.* Springer-Verlag, New York, 1984.

[9] **Coelho, H., J. C. Cotta, & L. M. Pereira,** *How to Solve it with Prolog.* Laboratorio Nacional de Engenharia Civil, Lisbon, Portugal, 1980.

[10] **Denes, J. & A. D. Keedwell,** *Latin Squares and Their Applications.* Akademiai Kiado, Budapest, 1974.

[11] **Fried, E., Mrs. Lanczi, & J. Suranyi,** *"Who is an Expert of What?" Problems of the Mathematical Competitions Organized by the Hungarian Television in 1964 and 1966.* Tankonyvkiado, Budapest, 1968 (in Hungarian).

[12] **Hajos, G., G. Neukomm, & J. Suranyi** (eds.), *Problems of Competitions in Mathematics. Part 1: 1894-1928.* 3rd edition. Tankonyvkiado, Budapest, 1965 (in Hungarian).

[13] **Jensen, K. & N. Wirth,** *PASCAL User manual and Report.* Springer-Verlag, New York, 1978.

[14] **Karteszi, F.,** *An Introduction into Finite Geometries.* Akademiai Kiado, Budapest, 1972 (in Hungarian).

[15] **Knuth, D. E.,** *The Art of Computer Programming. Vol. 2 Seminumerical Algorithms.* Addison-Wesley, Reading, Mass., 1969.

[16] **Kowalski, R.,** *Predicate Logic as a Programming Language.* Proc. IFIP 74, North Holland, 1973.

[17] *Kozepiskolai Matematikai Lapok (Hungarian Mathematical Journal for Highschool Students),* Vol. 45, 1972 (in Hungarian).

[18] *Kozepiskolai Matematikai Lapok (Hungarian Mathematical Journal for Highschool Students),* Vol. 46, 1973 (in Hungarian).

[19] *Kozepiskolai Matematikai Lapok (Hungarian Mathematical Journal for Highschool Students),* Vol. 50, 1975 (in Hungarian).

[20] *Kozepiskolai Matematikai Lapok (Hungarian Mathematical Journal for Highschool Students),* Vol. 51, 1975 (in Hungarian).

[21] *Kozepiskolai Matematikai Lapok (Hungarian Mathematical Journal for Highschool Students),* Vol. 59, 1979 (in Hungarian).

[22] *Letters of Mathematics -- Form 3.* Admission Preparation Committee of the Faculties of Sciences, Budapest (in Hungarian).

[23] *Letters of Mathematics -- Form 4.* Admission Preparation Committee of the Faculties of Sciences, Budapest (in Hungarian).

[24] **Lloyd, J. W.,** *Foundations of Logic Programming.* Springer-Verlag, New York, 1987.

[25] **Markusz, S.,** *Easily Comprehensible Prolog Programming.* NOVOTRADE RT., Budapest, 1988 (in Hungarian).

[26] **Molnar, E.,** *Collected Problems of Competitions in Mathematics, 1947-1970.* Tankonyvkiado, Budapest, 1974 (in Hungarian).

[27] *MPROLOG Language Reference Manual.* Logicware, SZKI, Budapest, 1985.

[28] **Naish, L.,** *Negation and Control in Prolog.* Lecture Notes in Computer Science Vol. 238. Springer-Verlag, New York, 1986.

[29] **Perelman, Ya. I.,** *Stories and Puzzles in Mathematics.* Gondolat, Budapest, 1979 (in Hungarian).

[30] **Sterling, L. & E. Shapiro**, *The Art of Prolog.* M.I.T. Press, Cambridge, Mass., 1986.

[31] **Van Emden, M.**, *First-Order Predicate Logic as a High-Level Programming Language.* Dept. of A.I., MIP-R-106, Univ. of Edingurgh, 1974.

[32] **Warren, D. H. D.**, *Implementing Prolog.* Res. Report 39, 40. Dept. of A.I., University of Edinburgh, 1977.

[33] **Wirth, N.**, *Programming in Modula-2.* Springer-Verlag, New York, 1982.

A TANULMÁNYOK SOROZATBAN 1987-BEN MEGJELENTEK:

195/1987	Telegdi László: Bináris változók strukturájának vizsgálata
196/1987	Rónyai Lajos: Algebrai algoritmusok
197/1987	Hernádi Ágnes - Bodó Zoltán - Knuth Előd: A tudásábrázolás technikái és gépi eszközei
198/1987	Miguel Fonfria Atan: A data base management system developed for the Cuban minicomputer CID 300/10
199/1987	Bach Iván - Farkas Ernő - Naszódi Mátyás: A magyar nyelv elemzése számitógéppel
200/1987	Publikációk'86 /Szerkesztette: Petróczy Judit/
201/1987	Eszenszki József - Hévizi László - dr. Kas Iván - dr. Laufer Judit - Palotási András - Szőnyi Tamás - Dr. Vörös Károly: Tanulmányok a számitástechnika nyomdaipari alkalmazásához
202/1987	Problems of Computer Science Proceedings of the joint workshop of Computer and Automation Institute of HAS and Computing Centre of Armenian Academy of Sciences held in Budapest, September 1987. /Edited by: G.B. Marandzjan, B. Uhrin/