PROLOG IN lO FIGURES

Alain Colmerauer

Centre Mondial d' In-formatique
22 avenue Matignon, 75008 Paris
and
Faculte des Sciences de Luminy
case 901, 13288 Marseille Cedex 9

Abstract; Prolog is presented in a rigourous way, through 10 easily understandable figures. Its theoretical model is completly rewrought. After introducing infinite trees and inequalities, this paper puts forth the minimal set of concepts necessary to give Prolog an autonomous existence, independent of lengthy considerations about first order logic and inference rules. Mystery is sacrificed in favor of clarity.

Artificial Intelligence interacts with many fields including psychology, linguistics, history, geology, biology, medical science ... These sciences are complex, and special tools are needed to represent and process the knowledge they deal with. Furthermore, these tools should not introduce new problems, inherent to computer science. Traditionally, the science of knowledge has been mathematical logic. Therefore it was reasonable to turn to logic for help in developing a tool for Artificial Intelligence: that was how Prolog was born.

Prolog, developed in 1972 by A.Colmerauer and P.Roussel, was at first a theorem prover, based on A.Robinson's resolution principle (1965) with strong restrictions to narrow the search space. Credit is given to R.Kowalski and M.van Emden for having pointed out these restrictions as equivalent to the use of clauses having at least one positive literal (Horn clauses), and for having proposed the first theoretical model of what is computed by Prolog: a minimal Herbrand interpretation.

However, Prolog's close links with Logic proved sometimes to be inhibiting vis-a-vis its implementation. It was necessary to reformulate the theory to take into account implementation constraints: this new theory is unencumbered by distinctions necessary only in logic, and is enriched by concepts indispensable for programming purposes (such as inequalities). We can say that, after a careful implementation, a new theoretical model of Prolog emerged and it is this new model that we present here in 10 commented figures.

The reader interested in further readings on this subject is referred to the following:

On automatic theorem proving and logic:

ROBINSON J.A. (1979). "Logic: Form and Function", Edinburgh University Press and Elsevier North Holland.

On the links between logic and Prolog:

KOWALSKI R.A. (1979). "Logic For Problem Solving", Artificial Intelligence series, (Ed- Nilsson, N.J.), North Holland.

On the genesis of Prolog:

COLMERAUER A., KANOUI H., PASERO R. et ROUSSEL Ph. (1973), "Un systeme de communication homme-machine en frangais", Research Report, Groupe Intelligence Artificielle, Faculte des Sciences de Luminy, Marseille.

ROUSSEL Ph. (1975). "Prolog, Manuel de Reference et d'Utilisation, Groupe Intelligence Artificielle, Faculte des Sciences de Luminy, Marseille.

A Prolog system, based on the ideas developed here, and implemented on several computers (Apple II, Vax/Vms, etc.), is described in three Internal Reports of the Groupe Intelligence Artificielle, Faculte des Sciences de Luminy, Marseilles

COLMERAUER A. (1982). "Prolog II, Reference Manual and Theoretical Model".

VAN CANEGHEM M. (1982). "Prolog II, User's Manual".

KANOUI H. (1982). "Prolog II, Manual of Examples.

1. TREES

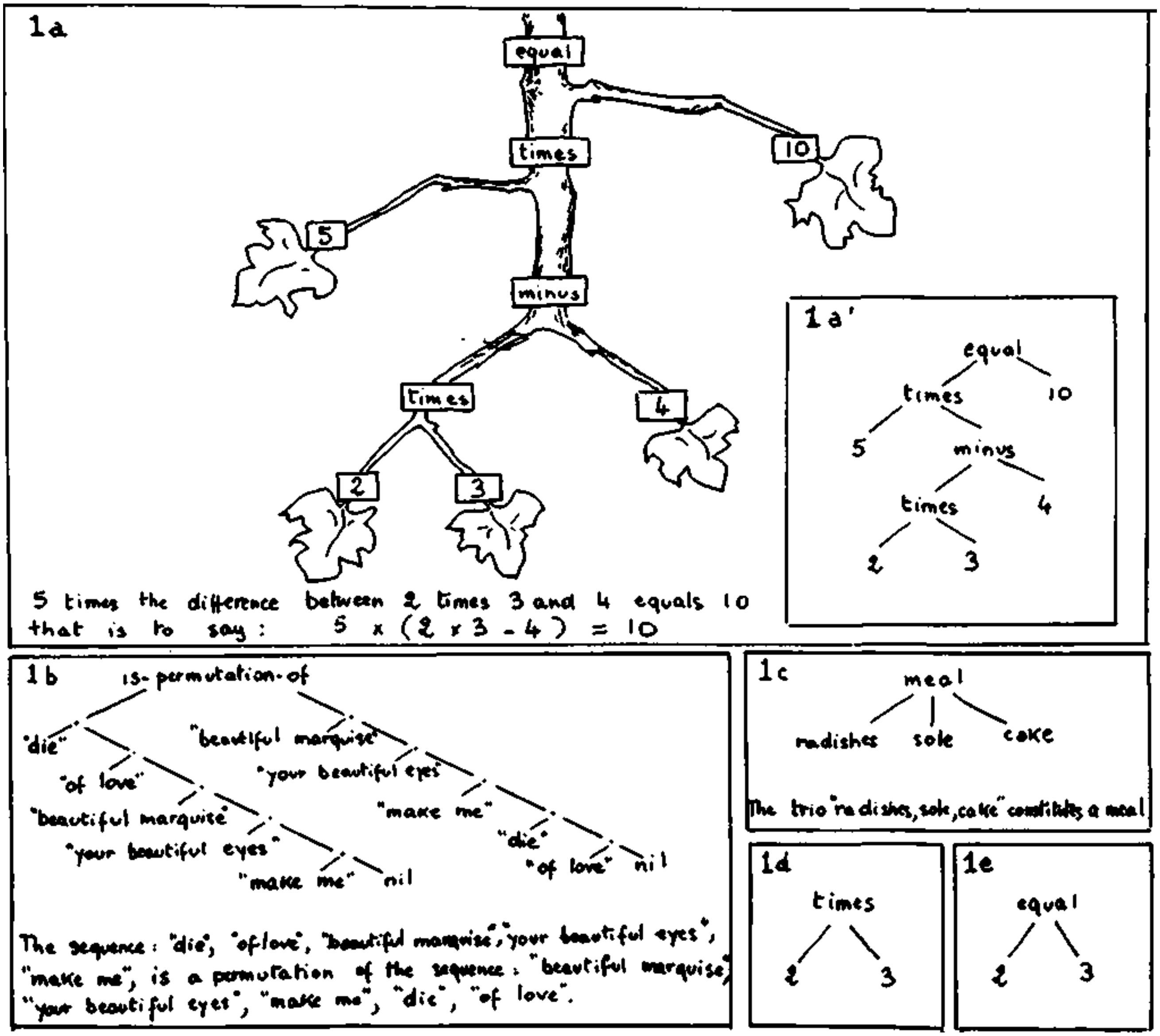
From an abstract point of view, one may say that the knowledge of an intelligent being on a given subject, is the set of facts that he or she can generate on the subject. Therefore, knowledge can be viewed as a set of facts, specified by a set of rules. Each of these facts can be represented by a declarative sentence. In our case we represent a fact by a tree, drawn upside down, as the one shown in Fig la. Each leaf and each node is labeled with an "atom" of information: this atom can be a word, a group of words, a number, or a special character. Only the structure of the tree is relevant. Therefore, Figs la and la' are equivalent. Trees in Figs la, lb and lc are examples of facts in three different fields: arithmetic, (stylistic) permutations, and meal planning. Facts are always trees, but not all trees are facts: obviously the trees in Figs Id and le are not facts in arithmetic, even if tree in Fig Id is a sub-tree of the fact in Fig la.

Trees were purposely chosen as data structures: they are capable of expressing complex information and, at the same time, simple enough to be handled algebraically, and by a computer.

2. TERMS

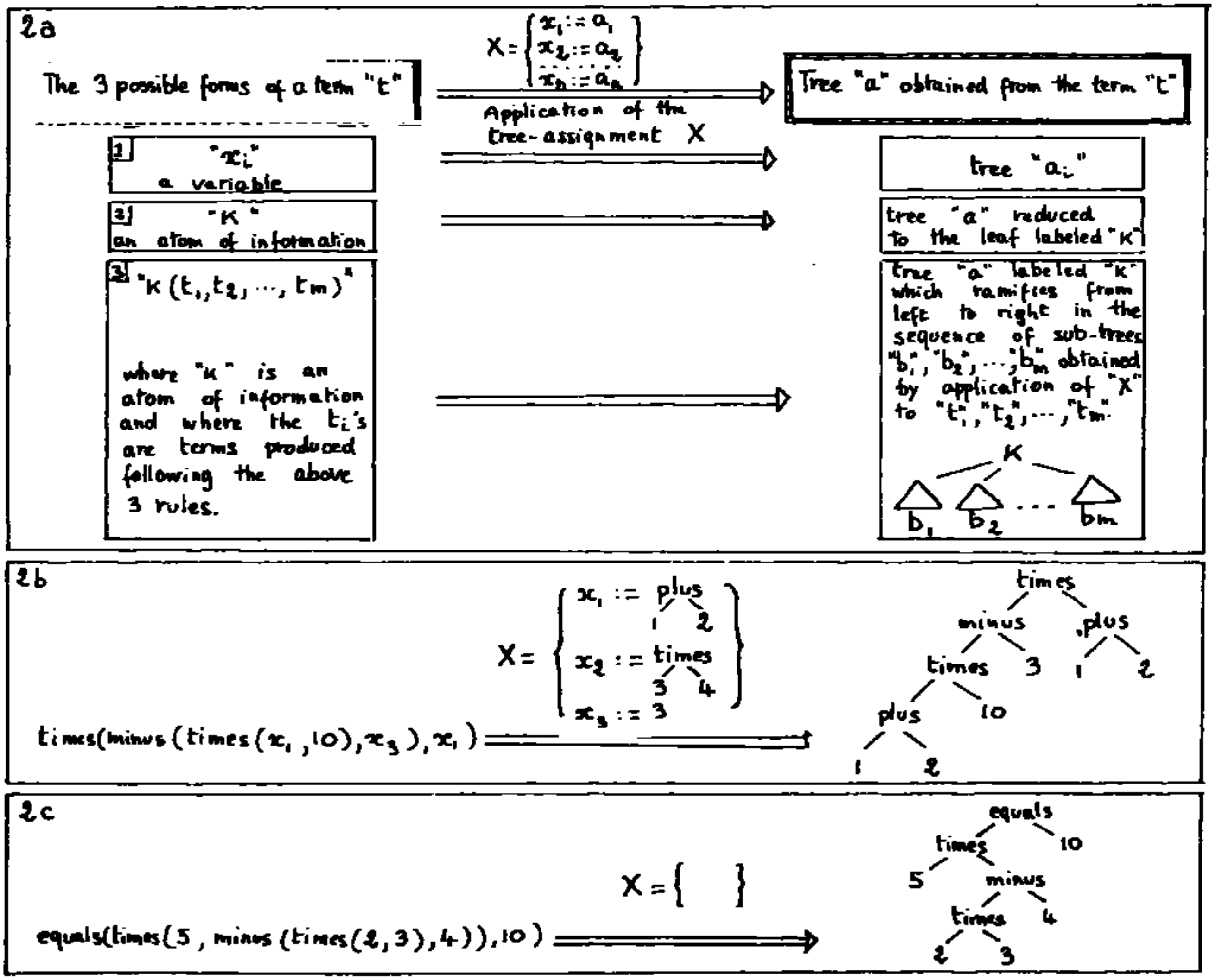
Formulas are used to represent tree patterns. These formulas called "terms", consist of atoms of information, variables, parentheses and commas. Recall that an atom of information is either a group of words, a number, or a special character. In the left column of Fig 2a the syntactic structure of a term is defined; this is a recursive definition where complex terms are defined from simpler terms; the simplest terms are variables or atoms of information. Examples of terms can be found in the left part of Figs 2b and 2c.

Je remercie Jacques Cohen de m'avoir aide a rediger cet article en anglais.


1a
equal
times
10
5
minus
times
4
2
3
5 times the difference between 2 times 3 and 4 equals 10
that is to say : 5 x (2 x 3 - 4) = 10
1a'
equal
times
10
5
minus
times
4
2
3
1b
is-permutation-of
"die"
"of love"
"beautiful marquise"
"your beautiful eyes"
"make me"
nil
"beautiful marquise"
"your beautiful eyes"
"make me"
"die"
"of love"
nil
The sequence : "die", "of love", "beautiful marquise", "your beautiful eyes", "make me", is a permutation of the sequence : "beautiful marquise", "your beautiful eyes", "make me", "die", "of love".
1c
meal
radishes
sole
cake
The trio "radishes, sole, cake" constitutes a meal
1d
times
2
3
1e
equal
2
3

Variables occurring in terms represent unknown trees. Therefore, the tree expressed by a term will depend upon the trees assigned to the variables. Such an assignment "X", called a "tree-assignment", is just a set of pairs "xi:=ai", "ai" being the tree assigned to the variable "x_1". The right column of Fig 2a gives the tree "a" represented by the term "t" after the application of tree-assignment "X". It is assumed that if "t" contains no variable, an empty tree-assignment can be applied.

Figs 2b and 2c depict two examples of tree-assignments. Example 2b shows that it is possible to find in the assignment "X", variables which do not occur in the term, but the contrary is not possible. In example 2c, the term contains no variable; this means that the corresponding tree does not depend on the assignment. The last example shows a systematic way of coding a finite tree by a term without variables.


2a
X = { x1 := a1, x2 := a2, ..., xn := an }
The 3 possible forms of a term "t"
Application of the tree-assignment X
Tree "a" obtained from the term "t"
1 "xi" a variable
tree "ai"
2 "K" an atom of information
tree "a" reduced to the leaf labeled "K"
3 "K(t1,t2,...,tm)"
where "K" is an atom of information and where the ti's are terms produced following the above 3 rules.
tree "a" labeled "K" which ramifies from left to right in the sequence of sub-trees "b1","b2",...,"bm" obtained by application of "X" to "t1","t2",...,"tm"
K
b1 b2 ... bm
2b
X = { x1 := plus(1, 2), x2 := times(3, 4), x3 := 3 }
times(minus(times(x1,10),x3),x1)
times minus plus times 3 1 2 plus 10 1 2
2c
X = { }
equals(times(5, minus(times(2,3),4)),10)
equals times 10 5 minus times 4 2 3

3. CONSTRAINTS

Prolog is a language which "computes" on trees "aj" represented by variables "xi". This computation is done by accumulating constraints that final trees must satisfy. These constraints limit the values variables can take, that is the. tree-assignment of variables "xi" by trees "ai". As shown in Fig 3a, a constraint "C" consists of a set of elementary constraints, each of them to be satisfied. An elementary constraint is either a pair of terms "<SJ,SJ'>" which will represent equal trees, or a pair of terms "<tk,t|<')" which will represent unequal trees. Fig 3a illustrates the general condition under which a tree-assignment "X" satisfies a constraint "C". "X" is also said to be a solution of "C". Fig 3b shows an example of a constraint "CI" satisfiable by the tree-assignment "XI". In Fig 3c there are three constraints which cannot be satisfied by any tree-assi gnment.

During the execution of a Prolog program, the basic operation consists of verifying whether a constraint is "satisfiable" or not (by at least one tree-assignment). This is done by "reducing" it, as seen in Fig 3d: the purpose of "reducing" is to simplify the constraint in order to make all its solutions explicit. This involves exhibiting variables distinct from each other as left members of equalities. To do so, we use a specific property of trees: the unique decomposition of a tree into immediate subtrees. This property permits us to replace

(pair(equal(y,10),y)=pair(x,times(5,z))}

by

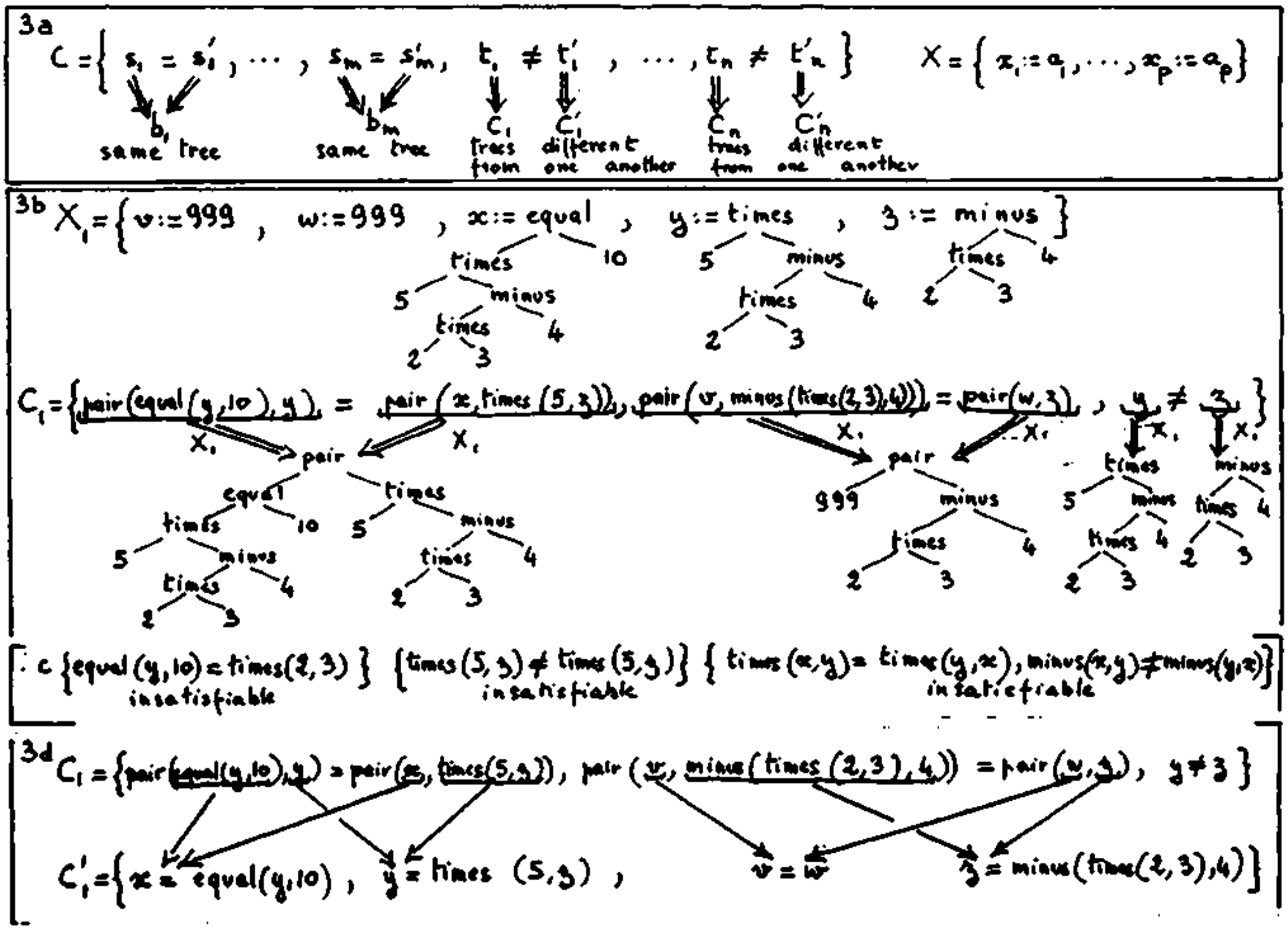
{equal(y,10)=x, y=times(5,z)}.

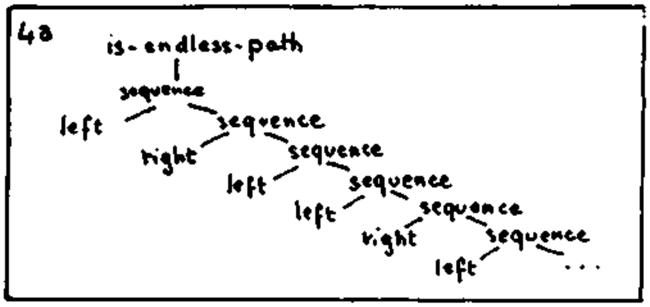
Note that if this property would hold for numbers, we would wrongly conclude that the two constraints "{x+3=2+y)" and "(x+3=2+y)" are equivalent! If we succeed in producing equalities where left members are distinct variables and where there are no inequalities, thpn the constraint is satisfiable. Its solutions are directly obtained by assigning arbitrary trees to variables not appearing as left members.

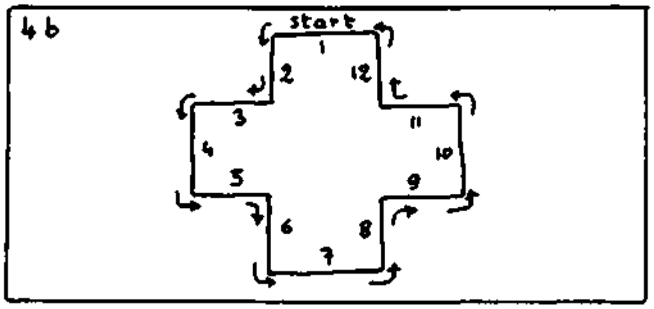
If inequalities *are* left, let "n" be their number.

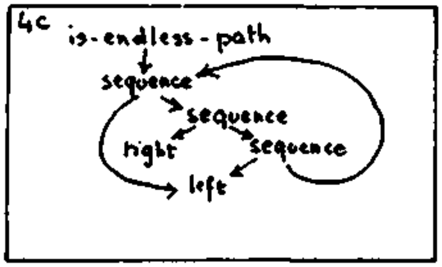
Another basic property allows us to split the initial problem into "n" independent and simpler sub-problems: a constraint of the form "$Cu\{t_1 \neq t_1', \ldots, t_n \neq t_n'\}$" is satis-fiable if and only if each of the constraints "$Cu\{t_1 \neq t_1'\}$", ... , "Cu{t_n*t_n')" is also satisfiable. Again, this is not true in the domain of natural integers "0,1,2,...", because it would be possible to show that the constraint "{x+y=2,x≠0,x≠1,x≠2}" has at least one solution since the constraints *"(x+y=l,i:*0}"*, "{x+y=1,x≠1}" and "{x+y=1,x≠2}" have at least one' In order to verify that the constraint "$Cu\{t_i \neq t_i'\}$" is satisfiable (knowing that "C" already is) we must check that the constraints "C" and "$Cu\{t_i = t_i'\}$" are not equivalent. If the constraint "Cutt^tj'}" is not satisfiable, we can even remove the inequality "$t_i \neq t_i'$", as in example 3d.

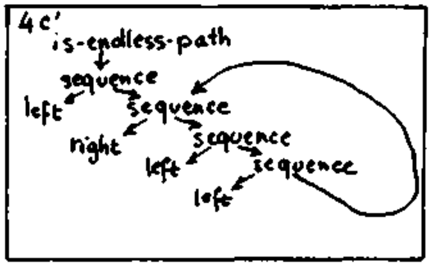
In the same way as we simplify equalities, it is possible to simplify inequalities. This allows us to present any satisfiable constraint in a "reduced form": this reduced form shows that the constraint is satisfiable by making all its solutions explicit. The general form of a reduced constraint containing inequalities is beyond the scope of this paper (see Colmerauer 1982).

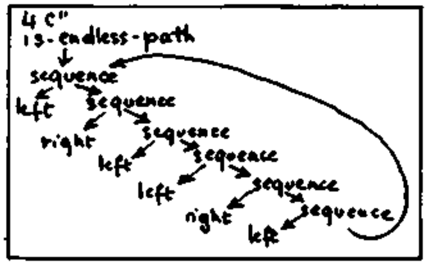

3a
C = { s1 = s'1 , ... , sm = s'm , t1 ≠ t'1 , ... , tn ≠ t'n }
X = { x1 := a1 , ... , xp := ap }
b1 same tree
bm same tree
C1 C'1 trees different from one another
Cn C'n trees different from one another
3b
X1 = { v := 999 , w := 999 , x := equal , y := times , z := minus }
C1 = { pair(equal(y,10),y) = pair(x,times(5,z)), pair(v,minus(times(2,3),4)) = pair(w,z) , y ≠ z }
X1
pair
equal times minus 999 5 10 4 2 3
c {equal(y,10) = times(2,3)}
insatisfiable
{times(5,z) ≠ times(5,z)}
insatisfiable
{times(x,y) = times(y,x), minus(x,y) ≠ minus(y,x)}
insatisfiable
3d
C1 = { pair(equal(y,10),y) = pair(x,times(5,z)), pair(v,minus(times(2,3),4)) = pair(w,z), y ≠ z }
C'1 = { x = equal(y,10) , y = times (5,z) , v = w z = minus(times(2,3),4) }

4a


4b


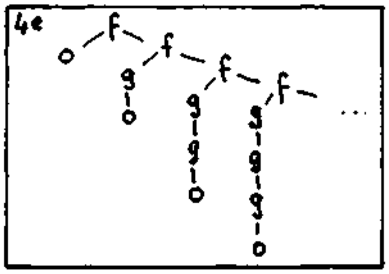
4c


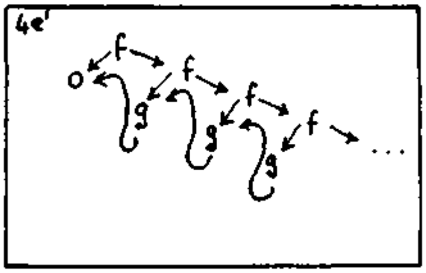
4c'


4c''


4d

$$\left\{\begin{array}{l} x = \text{is-endless-path}(y_1) \\ y_1 = \text{sequence}(\text{left}, y_2) \\ y_2 = \text{sequence}(\text{right}, y_3) \\ \cdots\cdots\cdots\cdots \\ y_{11} = \text{sequence}(\text{right}, y_{12}) \\ y_{12} = \text{sequence}(\text{left}, y_1) \end{array}\right\}$$

4e


4e'


4d' $\{ x = \text{is-endless-path}(y), \quad y = \text{sequence}(\text{left}, \text{sequence}(\text{right}, \text{sequence}(\text{left}, y))) \}$

4. INFINITE TREES

As surprising as it may be, it is also possible to handle in-finite trees. Such a tree is shown in Fig 4a: it represents an endless path along the cross-like -figure shown in Fig 4b. It is possible to present this tree by the diagram with a loop in 4c, obtained by merging all the nodes from which isomorphic subtrees arise, that is, -from which equal subtrees arise. If we omit to merge a -few nodes, we obtain the different diagrams in 4c? and 4c'' which still represent the same tree. That Fig 4c is a finite diagram means that the initial tree in 4a contains a finite set of configurations or, more precisely, that the set of its subtrees is finite: this is the definition of a "rational" tree. Of course, all finite trees *are* rational. Although finite trees can be defined by simple terms without variables, infinite rational trees can only be defined by the constraints they must satisfy. Taking into account successively all sides "1,2,...,12" of the cross-like figure in 4b, we construct the constraint 4d which is satisfied only in case of the assignment of "x" by the tree in Fig 4a. From the diagram shown in Fig 4c, we can construct a simpler constraint 4d', having the same property.

For the curious reader we provide in Fig 4e an example of a non-rational infinite tree. After merging all possible nodes this tree yields the infinite diagram in Fig 4e'. Note that it would be necessary to have a constraint, made from an infinity of elementary constraints, to completely describe this type of tree.

5. A PROLOG PROGRAM: LET'S EAT WELL

We now interrupt our theoretical development to present an example of a Prolog program. The program computes the composition of "light" meals and consists of the three parts shown in Figs 5a, 5b and 5c.

In Fig 5a we describe, by 11 rules, a possible set of meals, regardless of their dietetic qualities. The first rule states that:
   - if "a" is an appetizer and,
   - if "m" is a main course and,
   - if "d" is a dessert,
   then the triplet "a,m,d" is a meal.
The next two rules state that:
   - if "m" is a fish, "m" is a main course and,
   - if "m" is a meat, "m" is a main course.
The remaining eight rules classify a few courses. In Fig 5a', the computer answers two questions based on the knowledge described in Fig 5a. The

first question is:

    what are the values of "m",
    that make "m" a main course?

There are several possible answers and each answer is given as a reduced constraint on "m". The second question is:

    what are the triplets "a,m,d"
    which constitute a meal?

The corresponding answers are also presented in Fig 5a'.

In Fig 5b we introduce a minimal knowledge of arithmetic of positive integers: the addition "x+y=z" with "z<10", denoted by "small-sum(x,y,z)". The fact that "x+y=z" implies "(x+1)+y=(z+1)" is used to define the notion "small-sum" from the notion "small-successor", utilizing two rules. The notion "small-successor" is defined by eight rules so that "small-successor(x,y)" corresponds to the equality "y=x+1" with "y<10". Fig 5b' presents the computer's answers to a few questions about arithmetic. Observe that, according to the formulated questions, the same small Prolog program of Fig 5b also computes the sum, the difference, or decomposes a number in all possible sums of two numbers.

Fig 5c defines a light meal (based on Figs 5a and 5b) by assigning a certain amount of caloric units to each course, and restricting to meals which add up to a number of units smaller than 10. The main rule of Fig 5c states that:

- if the triplet "a,m,d" is a meal and,
- if the number of units of "a" is "x" and,
- if the number of units of "m" is "y" and,
- if "x+y=u" and "u<10" and,
- if the number of units of "d" is "z" and,
- if "z+u=v" and "v<10",

then the triplet "a,m,d" is a light meal.

In Fig 5c', the computer lists the seven allowed meals!

5a

```
meal(a,m,d) ->
   appetizer(a)
   main(m)
   dessert(d);

main(m) -> fish(m);
main(m) -> meat(m);

appetizer(radishes) ->;
appetizer(pate) ->;

fish(sole) ->;
fish(tuna) ->;

meat(porc) ->;
meat(beef) ->;

dessert(cake) ->;
dessert(fruit) ->;
```

5b

```
little-sum(1,x,y) ->
   little-successor(x,y);
little-sum(x',y,z') ->
   little-successor(x,x')
   little-sum(x,y,z)
   little-successor(z,z');

little-successor(1,2) ->;
little-successor(2,3) ->;
little-successor(3,4) ->;
little-successor(4,5) ->;
little-successor(5,6) ->;
little-successor(6,7) ->;
little-successor(7,8) ->;
little-successor(8,9) ->;
```

5c

```
light-meal(a,m,d) ->
   meal(a,m,d)
   units(a,x)
   units(m,y)
   little-sum(x,y,u)
   units(d,z)
   little-sum(z,u,v);

units(beef,3) ->;
units(fruit,1) ->;
units(cake,5) ->;
units(pate,6) ->;
units(porc,7) ->;
units(radishes,1) ->;
units(sole,2) ->;
units(tuna,4) ->;
```

5a'

```
main(m)?
{m=sole}
{m=tuna}
{m=porc}
{m=beef}

meal(a,m,d)?
{a=radishes, m=sole, d=cake}
{a=radishes, m=sole, d=fruit}
{a=radishes, m=tuna, d=cake}
{a=radishes, m=tuna, d=fruit}
{a=radishes, m=porc, d=cake}
{a=radishes, m=porc, d=fruit}
{a=radishes, m=beef, d=cake}
{a=radishes, m=beef, d=fruit}
{a=pate, m=sole, d=cake}
{a=pate, m=sole, d=fruit}
{a=pate, m=tuna, d=cake}
{a=pate, m=tuna, d=fruit}
{a=pate, m=porc, d=cake}
{a=pate, m=porc, d=fruit}
{a=pate, m=beef, d=cake}
{a=pate, m=beef, d=fruit}
```

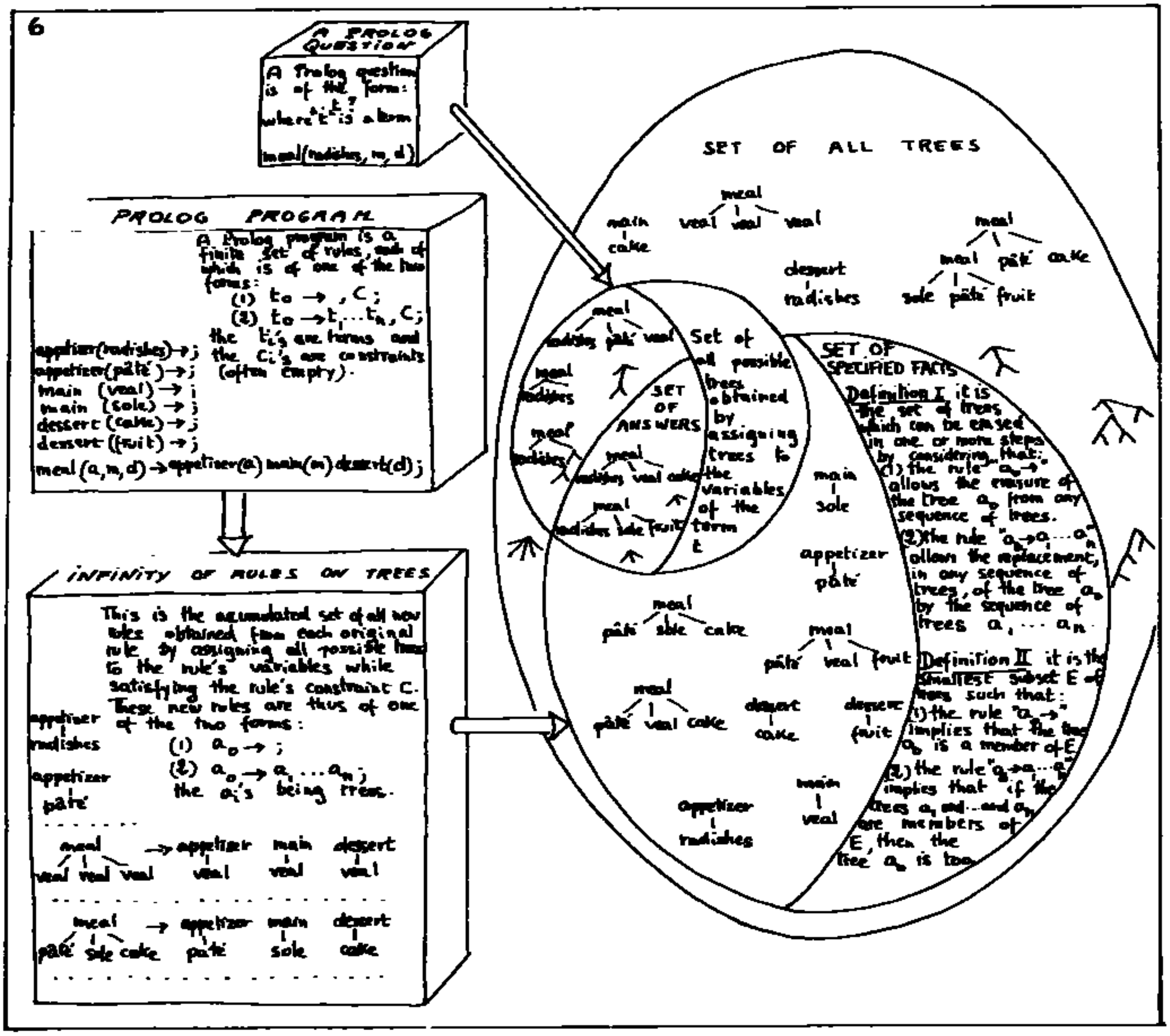
5b'

```
little-sum(4,3,x)?
{x=7}

little-sum(4,x,7)?
{x=3}

little-sum(x,y,5)?
{x=1, y=4}
{x=2, y=3}
{x=3, y=2}
{x=4, y=1}
```

5c'

```
light-meal(a,m,d)?
{a=radishes, m=sole, d=cake}
{a=radishes, m=sole, d=fruit}
{a=radishes, m=tuna, d=fruit}
{a=radishes, m=porc, d=fruit}
{a=radishes, m=beef, d=cake}
{a=radishes, m=beef, d=fruit}
{a=pate, m=sole, d=fruit}
```

6



A Prolog question is of the form:
t?
where "t" is a term
meal(radishes, m, d)

A Prolog program is a finite set of rules, each of which is of one of the two forms:
(1) $t_0 \to , C;$
(2) $t_0 \to t_1 \ldots t_n, C;$
the t_i's are terms and the C_i's are constraints (often empty).

```
appetizer(radishes) → ;
appetizer(pâté) → ;
main (veal) → ;
main (sole) → ;
dessert (cake) → ;
dessert (fruit) → ;
meal (a,m,d) → appetizer(a) main(m) dessert(d);
```

This is the accumulated set of all new rules obtained from each original rule by assigning all possible trees to the rule's variables while satisfying the rule's constraint C. These new rules are thus of one of the two forms:
(1) $a_0 \to ;$
(2) $a_0 \to a_1 \ldots a_n;$
the a_i's being trees.

Set of all possible trees obtained by assigning trees to the variables of the term t

Definition I it is the set of trees which can be erased in one or more steps by considering that:
(1) the rule "$a_0 \to$" allows the erasure of the tree a_0 from any sequence of trees.
(2) the rule "$a_0 \to a_1 \ldots a_n$" allow the replacement, in any sequence of trees, of the tree a_0 by the sequence of trees $a_1 \cdots a_n$.

Definition II it is the smallest subset E of trees such that:
(1) the rule "$a_0 \to$" implies that the tree a_0 is a member of E
(2) the rule "$a_0 \to a_1 \ldots a_n$" implies that if the trees a_1 and ... and a_n are members of E, then the tree a_0 is too.

6. FORMAL MEANING OF PR0L06 PROGRAMS

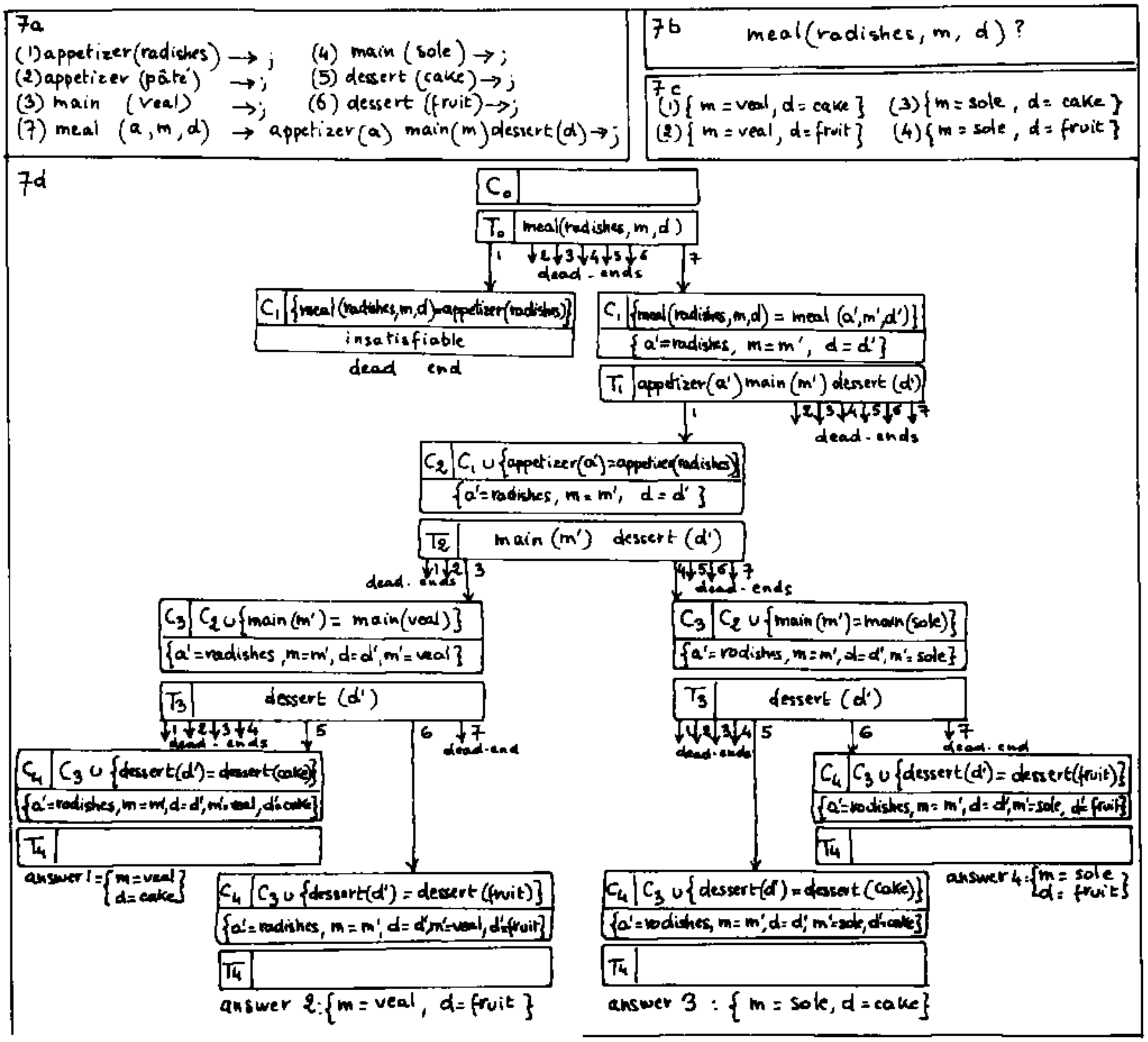
Fig 5 informally described a Prolog program. We now formalize its meaning. For most languages, the meaning of a program is given by the succession of elementary operations which the computer is supposed to perform. This is not true of Prolog which, as presented, is a formalism capable of representing knowledge and to express questions about it, independently of any computer. The computer's simply computes the answers to these questions.

In Fig 6 we derive in two steps the set of facts (a circle) specified by a Prolog program from the original program (a block). This set represents the potential knowledge contained in the program. Each of the facts is a tree, taken from all possible trees. The first step is required since the rules of a Prolog program are, actually, pattern of rules, and since it is first necessary to generate precise rules dealing with trees. The second step can be performed in two ways: either by considering the rules as rewriting rules (definition I), or by considering them as logical implications (definition II).

Fig 6 also caracterizes the set of facts which yield the answer to a Prolog question. A question is a single term "t" which stating:

what *are* the facts of the form "t"?

The set of valid answers is the intersection of the set of specified facts with the sub-set of trees, obtained by assigning all conceivable trees to the variables of term "t^M.



7a
```
(1) appetizer(radishes) -> ;        (4) main (sole) -> ;
(2) appetizer (pâté)    -> ;        (5) dessert (cake) -> ;
(3) main  (veal)        -> ;        (6) dessert (fruit) -> ;
(7) meal  (a, m, d)  -> appetizer(a) main(m) dessert(d) -> ;
```

7b meal(radishes, m, d) ?

7c
(1) { m = veal, d = cake } (3) { m = sole , d = cake }
(2) { m = veal, d = fruit } (4) { m = sole , d = fruit }

7d
C₀
T₀ meal(radishes, m, d)
1
2 3 4 5 6 dead - ends
7
C₁ {meal(radishes,m,d)=appetizer(radishes)}
insatisfiable
dead end
C₁ {meal(radishes,m,d) = meal (a',m',d')}
{a'=radishes, m = m', d = d'}
T₁ appetizer(a') main (m') dessert (d')
1
2 3 4 5 6 7 dead - ends
C₂ C₁ ∪ {appetizer(a')=appetizer(radishes)}
{a'=radishes, m = m', d = d'}
T₂ main (m') dessert (d')
1 2 dead - ends 3
4 5 6 7 dead - ends
C₃ C₂ ∪ {main (m') = main(veal)}
{a'=radishes, m=m', d=d', m'= veal}
C₃ C₂ ∪ {main (m')=main(sole)}
{a'= radishes, m=m', d=d', m'= sole}
T₃ dessert (d')
1 2 3 4 dead - ends 5 6 7 dead-end
T₃ dessert (d')
1 2 3 4 dead-ends 5 6 7 dead-end
C₄ C₃ ∪ {dessert(d')=dessert(cake)}
{a'=radishes, m=m', d=d', m'=veal, d'=cake}
T₄
answer 1 = {m=veal, d=cake}
C₄ C₃ ∪ {dessert(d') = dessert((fruit)}
{a'=radishes, m = m', d = d', m'=veal, d'=fruit}
T₄
answer 2 : {m = veal, d= fruit}
C₄ C₃ ∪ {dessert(d') = dessert(fruit)}
{a'=radishes, m = m', d = d', m'=sole, d'= fruit}
T₄
answer 4 : {m = sole, d = fruit}
C₄ C₃ ∪ {dessert(d') = dessert (cake)}
{a'=radishes, m = m', d = d', m'=sole, d'=cake}
T₄
answer 3 : {m = sole, d = cake}

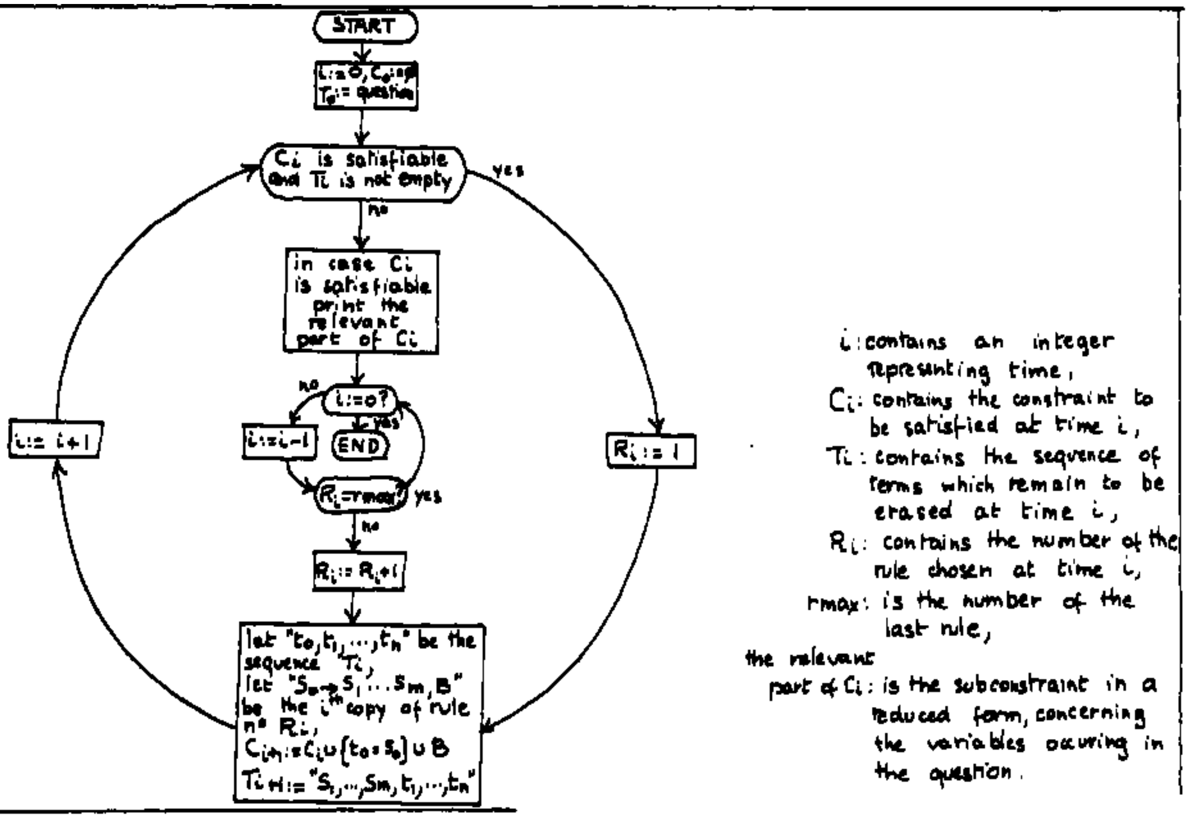
7. THE SEARCH SPACE

Fig 6 illustrated the double definition of the meaning of a Prolog program. Althougt both definitions are conceptually satisfying, they cannot be directly used to compute the answer to a given question.

However, this computation can be performed on the light of definition I by rewriting trees patterns instead of trees, with the use of a finite set of rules patterns instead of an infinite set of rules. A tree pattern is a "term-constraint" pair, the constraint limiting the represented trees; a rule pattern is, in fact, just a Prolog rule.

In Fig 7b the question on program 7a, states: under which constraints does "meal(radishes,m,d)" represent only facts? To compute these constraints the computer inspects the tree-shaped search space of Fig 7d. At each node there is a pair "(C_i,T_i)", "C_i" being a constraint to be satisfied, and "T_i" being a sequence of terms to be erased. At the root of the tree we have the pair "(C_0,T_0)", where "C_0" is empty and the sequence "T_0" is the term which constitutes the Prolog question. If at a given node the constraint "C_i" is not satisfiable, this node becomes a dead end. If not, there will be as many arrows emanating from this node, labeled "(C_i,T_i)", to nodes labeled "(C_{i+1},T_{i+1})", as there are rules. In this particular example there are seven arrows. The constraint "C_{i+1}" is obtained by adding to the constraint "C_i" the constraint "$\{b_0=a_0\}$", where "b_0" is the first element of sequence "T_i" and "a_0" the left member of the rule which one attempts to apply. "T_{i+1}" is obtained by replacing the first element of "T_i" by the right member of the rules which is being applied. Before considering a rule, it is important to rename variables (e.g. by adding primes), so that the rule has no variables in common with already existing variables. The constraints "C1" which are satisfiable are those which can be reduced; in this case, we place their reduced forms just below them. The answers are subparts of the reduced constraints "C1's" appearing in nodes which cannot longer be expanded, since their "T1's" are empty. The four valid answers are found in Fig ?c.

8


START
i:=0, C0:=∅
T0:= question
Ci is satisfiable and Ti is not empty
yes
no
in case Ci is satisfiable print the relevant part of Ci
no
i:=0?
yes
END
i:=i-1
Ri=rmax?
yes
no
Ri:= Ri+1
let "t0,t1,...,tn" be the sequence Ti,
let "S0→S1...Sm,B" be the i^th copy of rule n° Ri,
Ci+1:=Ci∪{t0=S0}∪B
Ti+1:="S1,...,Sm,t1,...,tn"
Ri:=1
i:=i+1
i: contains an integer representing time,
Ci: contains the constraint to be satisfied at time i,
Ti: contains the sequence of terms which remain to be erased at time i,
Ri: contains the number of the rule chosen at time i,
rmax: is the number of the last rule,
the relevant part of Ci: is the subconstraint in a reduced form, concerning the variables occuring in the question.

8. THE PROLOG CLOCK

The best way of explaining in detail how a Prolog program runs on a computer is to idealize this computer by a simple abstract machine. We call our abstract machine "the Prolog clock" because its basic function is to keep track of the time. This machine consists of:

1. a cell "i", containing a non negative integer representing the time;

2. an infinity of cells "$C_0,C_1,C_2,...$" containing the constraints to be satisfied at times 0,1,2,...;

3. an infinity of cells "$T_0,T_1,T_2,...$" containing the sequences of terms which, at times 0,1,2,..., remain to be erased;

4. an infinity of cells "$R_0,R_1,R_2,...$" containing the numbers of the rules which have been chosen at times 0,1,2,...

The rules are numbered "1" to "rmax" and the machine has means of acessing them.

The machine's operation is depicted by two concentric circles in Fig 8: one of them is swept clockwise as time increases by one unit, the other is swept in the opposite direction as time decreases by one unit. It is possible to reverse the time progression by crossing one of the two one-way bridges which link one circle to the other.

The execution *of* a Prolog program consists of answering a question represented by a term. All answers to be computed *are* constraints by which the term represents specified facts. We start with the pair "C_0,T_0)", "C_0" being empty and the sequence of terms "T_0" being reduced to the term that constitutes the question. Each turn around the outward circle increases the current constraint *"C_i"* and transforms the sequence "T_1'. Note that if the rule already contains a constraint "B", this constraint is added to the current set of elementary constraints. The process stops as soon as a non-satisfiable constraint is generated, or the sequence "T_i" becomes empty: in these cases, we backtrack to the "past", to try other rules. On the fly, if "Ci" is satisfiable, an answer is printed.

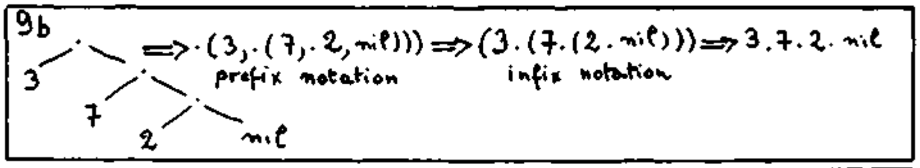
In fact, the above process corresponds to sweeping the tree-shaped search space of Fig 7, from top to bottom and from left to right, the time " i " being the level of the visited node.

The two programs in sections 9 and 10 provide additional examples of more intricate Prolog programs.

9a

1. "beautiful marquise"
2. "your beautiful eyes"
3. "make me"
4. "die"
5. "of love"

9b



9c

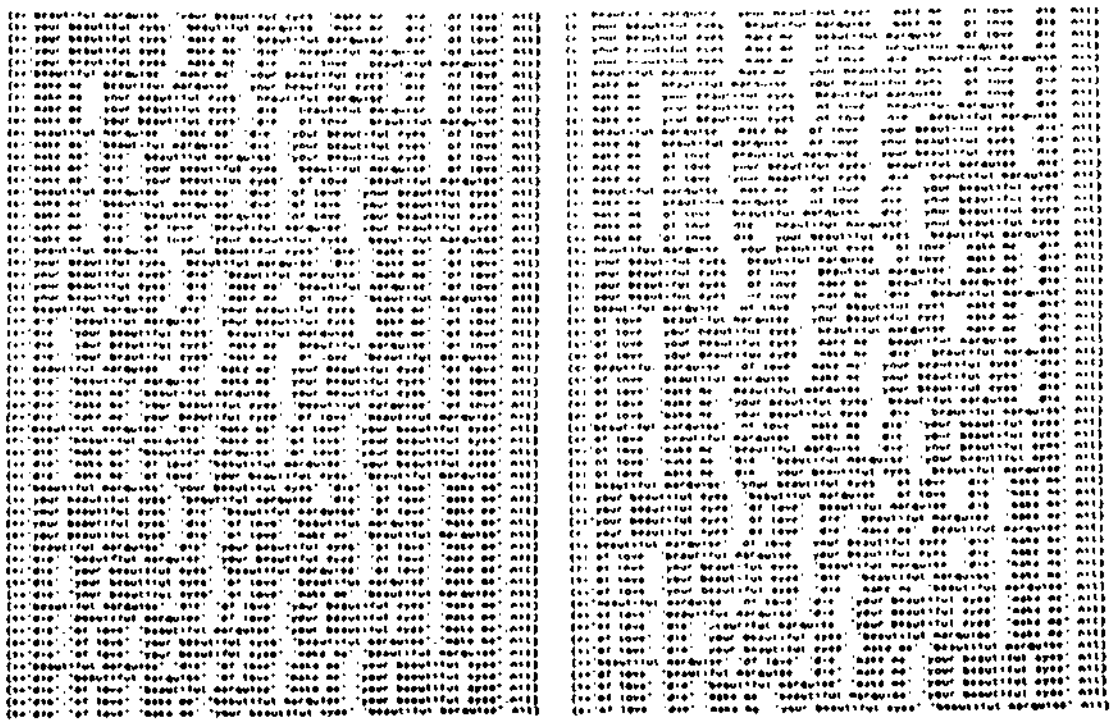
```
permutation(nil,nil) ->;
permutation(e.x,z) ->
    permutation(x,y)
    insertion(e,y,z);

insertion(e,x,e.x) ->;
insertion(e,f.x,f.y) ->
    insertion(e,x,y);
```

9d

```
permutation(1.2.3.nil,x)?              permutation(3.a.1.b.nil,2.4.c.d.nil)?
{x=1.2.3.nil}                          {a=2, b=4, c=3, d=1}
{x=2.1.3.nil}                          {a=2, b=4, c=1, d=3}
{x=2.3.1.nil}                          {a=4, b=2, c=3, d=1}
{x=1.3.2.nil}                          {a=4, b=2, c=1, d=3}
{x=3.1.2.nil}
{x=3.2.1.nil}
```

9e

```
permutation("beautiful marquise"."your beautiful eyes"."make me"."die"."of love".nil,x)?
```



9. STYLISTIC PERMUTATIONS

In Moliere's play, "Le Bourgeois Gentilhomme", a bourgeois who wants to act as a lord (gentilhomme) compliments a noble woman (marquise):

"Beautiful marquise, your beautiful
eyes make me die of love."

Let us construct all of the compliment's possible variations, as the bourgeois tries to do in the play. The sentence is first decomposed into five parts, which *are* given in Fig 9a, each part made up from one, or a few unseparable words. Starting from an initial sequence with these five components we produce all variants by generating all the permutations of the sequence.

We first have to choose a way of coding a sequence by a tree- Since it is necessary to have a notation for the empty sequence, the sequence "3,7/2" is represented by the tree in Fig 9b. Also, since each node is labeled with a single character, a dot, readablity is improved by using infix notation: "u.v", instead of prefix notation: ''(u,v)". To further simplify, we omit parentheses whenever left-right association is implied.

To assert that sequence "y" is a permutation of sequence "x" we write "permutation(x,y)". The first rule of Fig 9c states that the sequence of length zero, that is the empty sequence, has only one permutation, itself. The second rule specifies that, in order to permute a non-empty sequence, that is a sequence of length "n+1", we remove its first element "e" and obtain a sequence "x" of length "n"; we then compute any permutation "y" of this sequence "x" and insert the element "e" in any position of this sequence and produce the desired sequence "z". To insert an element "e" in a sequence "x" and obtain a sequence "y", we introduce the term "insertion(e,x,y)". We either insert "e" be-fore "x" (third rule of Fig 9c), or we insert "e" in the sequence which has its -first element removed (-fourth rule of Fig 9c). These four rules of Fig 9c constitute the entire permutation program.

Fig 9d presents the computer's answers to two questions:

what *are* all permutations "x" o-f the sequence "1,2,3"?

and

what *are* the values of the variables "a","b","c" and "d" so that "2,4,c,d" is a permutation of "3,a,l,b"?

Finally in Fig 9e, we ask the question producing the 120 stylistic variants that the "bourgeois gentilhomme" might have said'

10. SEND MORE MONEY

The purpose of this example is to solve a classical cryptarithmetic puzzle: assign 8 different digits to the 8 letters "S,E,N,D,M,0,R,Y", such that the sum "SEND+M0RE=M0NEY" becomes valid. To do so, we introduce in Fig 10a? the four carry-overs "rl", "r2", "r3" and "r4" which can be null and which have to be added to each column of the sum.

The program consists of the three parts shown in Figs 10b, 10c and lOd. In Fig lOd, the table of sums up to 20 is programmed: any elementary school student knows this table by heart but the machine has to compute it over and over again since it only knows how to add "1" to a number. We use "plus(x,y,z)" to mean "x+y=z". Each number, is represented by two digits, with a dot between them (we use infix notation as in Fig 9). Fig 10c presents the definition of a sequence without repetition (note that the last rule of Fig 10c contains a non-empty constraint). In Fig 10b, it is stated that to compute a solution it is necessary to assign distinct values to the letters "S,E,N,D,M,0,R,Y", and that, in each column of the sum, a property called "admissible", has to be satisfied between the carry-over, the three letters of the column and the preceding carry-over. Of course, this property "admissible" is defined using the property "plus" and the property "plus-one". Since the numbers "SEND", "MORE", and "MONEY" should not begin with the digit 0, an inequality constraint is added to the first rule of Fig 10b. In Fig 10e, we challenge the computer to provide us with the three mystery numbers.

10 a

```
   SEND
 + MORE
 -----
  MONEY
```

10 a'

```
 r1r2r3r4
   SEND
 + MORE
 -----
  MONEY
```

10 b

```
solution(S.E.N.D,M.O.R.E,M.O.N.E.Y) ->
   without-repetition(S.E.N.D.M.O.R.Y.nil)
   admissible(r1,0,0,M,0)
   admissible(r2,S,M,O,r1)
   admissible(r3,E,O,N,r2)
   admissible(r4,N,R,E,r3)
   admissible( 0,D,E,Y,r4),
   {S=0, M≠0};

admissible(0,u1,u2,u3,r) ->
   plus(0.u1,0.u2,r.u3);
admissible(1,u1,u2,u3,r) ->
   plus(0.u1,0.u2,x)
   plus-one(x,r.u3);
```

10 c

```
without-repetition(nil) ->;
without-repetition(u.l) ->
   out-of(u,l)
   without-repetition(l);

out-of(u,nil) ->;
out-of(u,v.l) ->
   out-of(u,l),
   {u=v};
```

10 d

```
plus(0.0,x,x) ->
   less-than-twenty(x);
plus(x',y,z') ->
   plus-one(x,x')
   plus(x,y,z)
   plus-one(z,z');

less-than-twenty(0.0) ->;
less-than-twenty(y) ->
   plus-one(x,y);

plus-one(0.0,0.1) ->;
plus-one(0.1,0.2) ->;
plus-one(0.2,0.3) ->;
plus-one(0.3,0.4) ->;
plus-one(0.4,0.5) ->;
plus-one(0.5,0.6) ->;
plus-one(0.6,0.7) ->;
plus-one(0.7,0.8) ->;
plus-one(0.8,0.9) ->;
plus-one(0.9,1.0) ->;
plus-one(1.0,1.1) ->;
plus-one(1.1,1.2) ->;
plus-one(1.2,1.3) ->;
plus-one(1.3,1.4) ->;
plus-one(1.4,1.5) ->;
plus-one(1.5,1.6) ->;
plus-one(1.6,1.7) ->;
plus-one(1.7,1.8) ->;
plus-one(1.8,1.9) ->;
```

10 e

```
                                                    9567
                                                  + 1085
solution(x,y,z)?                                  -----
{x=9.5.6.7, y=1.0.8.5, z=1.0.6.5.2}                10652
```