# Logic as a computer language for children

**Robert Kowalski,** Department of Computing, Imperial College, London, UK

For many years the artificial intelligence community has engaged in a debate about the relative merits of procedural versus declarative representations of knowledge. This debate has spilled over into the educational arena, where computational geometry as realized in Logo is represented as superior to classical nonprocedural geometry.

The commitment of many artificial intelligence researchers to procedural representations of knowledge runs counter to the increasing development in other areas of computing of nonprocedural, declarative programming languages, database query languages and program specification languages.

It is our belief that the apparent conflict between the declarative and the procedural can be reconciled by recognizing that, in the words of Pat Hayes

'computation = controlled deduction (Hayes, 1973)
i.e. 'algorithm = logic + control' (Kowalski, 1979a).

A specific example of this is the procedural interpretation of Horn clauses:

The application of suitably goal-directed proof-procedures to sentences of the form

A if B and C

treats them as procedures, which

reduce problems of the form A
to subproblems of the form B and C.

The procedural interpretation of Horn clauses is the logical foundation of the computer language Prolog, developed and implemented in 1972 by Colmerauer and his colleagues in Marseille. Prolog, at first almost entirely a European phenomenon, has recently gained attention in North America, thanks to McDermott's (1980) Sigart article, and in Japan, where it has been selected as the core programming language for the fifth generation computer systems of the 1990s (JIPDEC, 1981).

Because of the procedural interpretation which is the basis of Prolog, there are two Prolog programming styles: one declarative, the other procedural. Many Prolog programmers, especially those with a strong AI background, employ a procedural style. Others, especially those with a software engineering bias, emphasize the declarative. At Imperial College we are firmly committed to the declarative style. We recognize that Prolog is an efficient but logically restricted realization of logic programming ideal, and we use the extralogic features as rarely as possible. Even then we try to encapsulate them in the definition of logically defensible extensions of the language or its proof procedures (Kowalski, 1981).

We believe that in the first iteration programs should be written as declarative program specifications. Inefficient specifications can be transformed (preserving the logical intention of the specifications) into more efficient logic programs. My colleagues, Keith Clark *et al.* (1977, 1978, 1982), John Darlington and Chris Hogger (1978a, 1978b, 1981) have made significant contributions to this methodology.

As a complement to our activities, beginning in September 1980 we have been teaching logic as a computer language to children, starting at the age of 10. Richard Ennals (1982b) has been responsible for preparing and testing the teaching materials. Besides the computer science objectives of teaching children database query methodology, program specification and ultimately programming itself, the project aims both to apply logic to other subjects taught in school, as well as to teach logic as a subject in its own right. In order to do this we have concentrated almost entirely on the declarative reading of Prolog programs.

In the remainder of this paper I shall first sketch the outline of the one and a half years of the course that has been taught so far. Then I shall address the relationship between computer logic and classical school mathematics. Finally I shall compare the treatment of computational geometry in logic with its treatment in a procedurally oriented programming language such as Logo. A more comprehensive introduction to the children's form of Prolog can be found in Clark, Ennals and McCabe (1981).

## ATOMIC SENTENCES

The children are first taught to translate between English sentences and the *atomic sentences* of Prolog. The following are examples of atomic sentences constituting a simple database:

```
John likes Mary
Bob likes Mary
Mary wants bicycle
wheel part-of bicycle
tyre part-of wheel
Bob supplies wheel
Mary supplies brake-set
John supplies bicycle
```

Each *atomic sentence* expresses a binary relationship between two individuals and has the form

```
name of        name of        name of
individual     relation       individual
```

The spaces are significant and serve to separate the names of individuals from the name of the relation. Such relationships, whether they occur as self-standing sentences or not, are also called *atoms*. For the moment only binary relations are allowed and individuals are named only by constants. Other options are introduced later.

The language is a sugared-up syntax for a subset of logic. It is compiled by a micro-Prolog program into micro-Prolog internal syntax (McCabe, 1980–81). It runs on Z80 microcomputers under the CP/M operating system. Facilities are provided to

Add
Delete
List
Load and
Save

sentences or sets of sentences. For example,

```
Add(brake-set part-of bicycle)
```

augments the part-of relation in the database.

```
List part-of
```

lists the current state of the part-of relation:

```
wheel      part-of bicycle
tyre       part-of wheel
brake-set  part-of bicycle
```

Sentences are listed (and, as we shall see later, are used) in the order in which they are stored in the database.

In the early part of the course emphasis is placed on translation between informal English and this simple subset of logic. For example, the English sentence

John and Bob like Mary

is translated into two atomic sentences

John likes Mary
Bob likes Mary.

But the English sentence

John and Bob like themselves

is ambiguous.

It is a major objective of the project to teach the relationship between natural language syntax and its semantics, where as a first approximation the semantics are expressed in symbolic logic. This is thought to be an important object in its own right as a contribution to the more effective use of natural language: to teach the distinction between English sentences which are clear and precise and English sentences which are imprecise or meaningless.

## ATOMIC QUERIES

It is generally believed in the computer industry that significantly more people can be expected to interact with computers as database users rather than as computer programmers. It is with this in mind that we introduce database queries as early as possible.

The simplest queries involve only a single atom and are of the Yes–No variety:

```
Does (John likes John)
    No
Does (John likes Mary)
    Yes
Does (Mary likes John)
    No
```

The system as it is currently implemented uses the closed world assumption (Clark, 1978). If information cannot be derived from the database then the system assumes it to be false. This blanket assumption is, of course, extremely dangerous. For example, given the current state of the database:

```
Does (pedal part-of bicycle)
    No.
```

It is typical of computing systems today to act as though they 'know it all'. Perhaps it is just as well to encourage a healthy suspicion of computers as early as possible.

More complicated than Yes–No queries are queries with variables standing for unknown individuals. For example,

```
Which (x John likes x)
        Mary
Which (x x likes Mary)
        John
        Bob.
```

Each such query has the form

```
Which (output-pattern conditions)
```

where the *output pattern* can be a single variable, i.e. one of

```
x, y, z, X, Y, Z, x1, x2, etc.
```

or a *list* of names of individuals (including variables) enclosed in parentheses and separated by spaces, e.g.

```
(x y)
(the answer is x) (happy x)
```

Each *condition* is either a simple atom (with variables standing for unknown individuals) or as we shall see later a conjunction of atoms. Thus we can ask

```
Which ((x y) x part-of y)
        (wheel bicycle)
        (tyre wheel)
        (brake-set bicycle)
Which ((the answer is x) x likes Mary)
        (the answer is John)
        (the answer is Bob)
Which ((happy x) Mary likes x)
        No answer.
```

We shall see later that lists themselves can be used as names of individuals.

## COMPOUND QUERIES

Several atoms joined by 'and' and separated by spaces can be used as the conditions of a query:

```
Does (John likes Mary and Mary likes John)
                No
Which (x Mary wants x and Bob supplies x)
                No answer
Which (x Mary wants x and Bob supplies y and y part-of x)
                bicycle.
```

Compound queries provide a rich source of examples for analysing the meanings of English sentences. They are adequate, in particular, for expressing the meaning of the English questions:

Who supplies something Mary wants?
Who supplies a part of a part of a bicycle?

## GENERAL RULES

Instead of asking the same form of compound query over and over again, e.g.

```
Which (x x likes Mary and Mary likes x)
Which (x x likes John and John likes x)
Which ((x y) x likes y and y likes x)
```

it is possible to define a new relation by means of a general rule and query it instead.

```
Add (x friends-with y if x likes y and y likes x)
Which (x      x friends-with Mary)
Which (x      x friends-with John)
Which ((x y) x friends-with y)
```

A *general rule* has the form

conclusion if conditions

where 'conclusion' is an atom and 'conditions' is a conjunction of atoms (connected by 'and' and separated by spaces). Atomic sentences can be regarded as general rules which have no conditions. This means that variables can be used in atomic sentences, e.g.

```
Add (Bob likes x)
```

i.e. Bob likes everything.

Whereas *variables in queries* stand for unknown individuals, *variables in general rules* stand for any individual. All occurrences of the same variable in the same sentence stand for the same arbitrary individual. However, there is absolutely no connection between variables in different sentences even though they may have the same name. Thus the two sentences

```
x is female if x is-mother-of y
x is male   if x is-father-of y
```

do not in any way imply that an individual might be both male and female.

The use of general rules promotes a database user to a programmer — or more accurately, because we have been using only the declarative semantics of queries and general rules, to a program specifier.

## RECURSION

General rules can be used to augment relations already in the database, e.g.

> Add (Mary likes x if x likes Mary).

Here the general rule is recursive. Our experience has been that such recursion, stripped of its operational semantics, causes no conceptual difficulties for 10–11-year-old children.

Not every recursion is quite so easy. Consider for example, the 'part-of' relation. At this stage, this describes only the relationship between a part and its immediate subparts. There is nothing to imply, for example, that

```
if tyre       is part of wheel
and wheel is part of bicycle
then tyre   is part of bicycle.
```

Such an inference, however, is a special case of the recursive general rule

```
x part-of z  if    x part-of y
                   and y part-of z.
```

It can help to understand such recursions if we picture them graphically (Deliyanni and Kowalski, 1979).

## SEMANTIC NETWORKS

Suppose that we have three individuals, x, y, z, pictorially:

$$\begin{tikzcd}
\overset{x}{\bullet} & \overset{y}{\bullet} & \overset{z}{\bullet}
\end{tikzcd}$$

and that they are connected by part-of relationships

$$\begin{tikzcd}
\overset{x}{\bullet} \arrow[r, bend left, "\text{part-of}"] & \overset{y}{\bullet} \arrow[r, bend left, "\text{part-of}"] & \overset{z}{\bullet}
\end{tikzcd}$$

then we can add a new part-of relationship between x and z.

$$\begin{tikzcd}
\overset{x}{\bullet} \arrow[r, bend left, "\text{part-of}"] \arrow[rr, dashed, bend right, "\text{Part-of}"'] & \overset{y}{\bullet} \arrow[r, bend left, "\text{part-of}"] & \overset{z}{\bullet}
\end{tikzcd}$$

Such pictorial representations can certainly help to formulate and understand general rules. However, they are no substitute for the linear syntax. They bear a similar relationship to the logic of binary relations that Venn diagrams bear to the logic of one-place predicates, i.e. sets.

## LOOPS

With recursive definitions we now have the possibility that the underlying Prolog interpreter can go into a nonterminating loop. This happens already with the current state of the database and the query

Which (x x part-of bicycle).

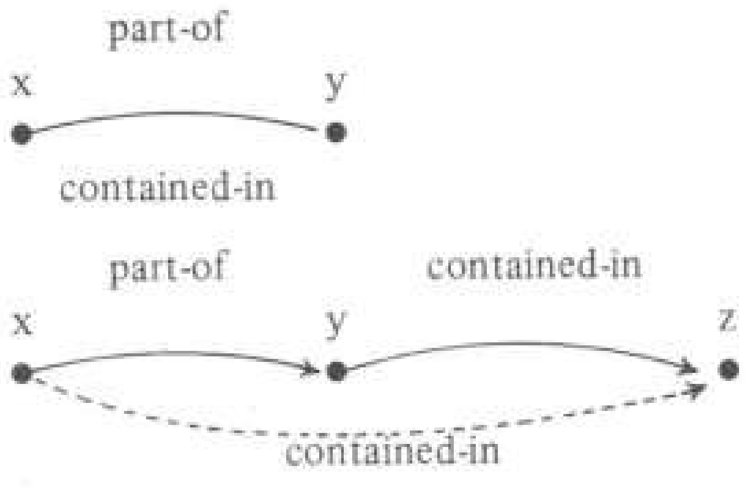
After successfully printing the first two answers

wheel
brake-set

the system goes into an infinite loop without printing the third answer

tyre.

In order to understand why this happens it is necessary to understand something about the procedural interpretation and its Prolog realization. But this defeats the object of restricting our understanding of logic to the declarative interpretation for as long as possible. For the moment we can escape this difficulty by noting that the loop can be avoided in this and many similar cases by using different relations to distinguish between one-step part-of connections and multi-step connections. Pictorially



We can delete the old recursive rule for the part-of relation and replace it by the new rules:

```
x contained-in y    if x part-of y
x contained-in z    if x part-of y
                    and y contained in z
```

## LISTS AS NAMES OF INDIVIDUALS

The range of applications can be greatly extended by allowing lists as names of individuals. The items in a *list* are separated by spaces and can be constants, variables or other lists. Here are some examples:

John born-on (1 May 1984)
(John Mary) parents-of (George Jane Alice)

It is also possible to have partially specified lists. We use

(x|y) to name the list $\overset{x}{.}\,\overbrace{.\ .\ .\ .\ .\ .\ .\ .}^{y}$
which starts with x followed by the list y.

in the same way that cons(x,y) is used in LISP. We use

(  ) to name the empty list

like nil in LISP

One of the most useful relations over lists is the 'belongs-to' relation:

```
x belongs-to (x|y)
x belongs-to (z|y) if x belongs-to y.
```

It can be queried as any other relation. It can be used to test that an item belongs to a list as if it were written in LISP, e.g.

```
Does (Bob belongs-to (George Jane Alice)).
      No.
```

But it can also be used to generate items as if it were defined explicitly by a relational database, e.g.

```
Which (x    x belongs-to (George Jane Alice)
        and x belongs-to (Bob Mary Jane))
             Jane
```

## N-ARY RELATIONS

For many applications it is natural to use one-place predicates and non-binary relations to construct atomic conditions and conclusions. We use the notation

P l

where P is the name of an N-ary relation and l is a list of N arguments. Thus we can write

```
Female (Mary)
Male    (John)
Eternal-triangle (x y z) if x likes z
                  and y likes z
                  and x different-from y
```

To define the relationship 'x different-from y' it is convenient to use the notion of negation by failure. But first we must introduce the procedural interpretation of the language we have introduced so far.

## THE PROCEDURAL INTERPRETATION

There eventually comes a point, as in the case of certain non-terminating loops and negation by failure when something needs to be said about the problem-solving behaviour of the computer. It is possible to do so without entering into excessive detail. Only three points need to be made:

(1) The computer uses general rules of the form

A if B and C

*backwards* to reduce problems of the form A to subproblems of the form B and C. It uses atomic sentences A to solve problems directly without introducing further subproblems.

(2) When several sentences (atomic or compound) can be used to solve the same problem, they are tried one at a time in the order in which they are written.

(3) When several problems

B and C

need to be solved, they are solved one at a time in the order in which they are written.

The first principle above defines the abstract procedural interpretation. It is compatible with both sequential and parallel execution of alternative procedures and of several subgoals. For the sake of efficiency the Prolog interpreter uses sequential execution determined by textual order for both.

The procedural interpretation reconciles the controversy over declarative versus procedural representations of knowledge. It shows that *special-purpose procedures* can be obtained by applying *general-purpose* problem-solving strategies to *domain-specific* knowledge expressed in symbolic logic.

Thus the recursive definition of the belongs-to relation given earlier, for example, has in addition to its basic declarative interpretation the following procedural interpretation: in the special case of showing that an element belongs to a list,

```
to show x belongs to a list (y|z),
    show x is identical to y
  or show x belongs to the list z.
```

## NEGATION BY FAILURE

Negative conditions can be used in both queries and general rules, e.g.

```
Which (x x belongs-to (George Jane Alice)
        and Not (x belongs-to (Bob Mary Jane)))
Unhappy (x) if x likes y and Not (y likes x)
```

In general a condition

```
        Not (p)
```

is judged to hold if the unnegated Yes–No query

```
        Does (p)
```

fails to hold, i.e. returns the answer 'No'.

Keith Clark (1978) has shown that negation by failure is equivalent to ordinary negation under the closed world assumption.

Unfortunately, the implementation of negation by failure makes the success or failure of a negative condition dangerously sensitive to the context in which it is executed. For example, given the definition of the belongs-to relation, the query

```
        Which  (x x belongs-to (George Jane Alice)
               and Not (x belongs-to (Bob Mary Jane)))
```

correctly gives the answers

```
        George
        Alice.
```

But if the order of the two conditions is reversed, it incorrectly responds that there is no answer. This is because the query

```
        Does (x belongs-to (Bob Mary Jane))
```

succeeds and therefore its negation fails.

This undesirable state of affairs can be corrected by altering the syntax of negation so that variables which should be uninstantiated when the condition is executed are explicitly listed as an additional argument of the negation operator. When the negative condition is executed, all such listed variables should be uninstantiated and all unlisted variables in the condition should be instantiated to terms containing no variables. Otherwise a control error occurs.

A similar solution has been taken in micro-Prolog for universally quantified conditions, which can be implemented by negation by failure.

**FOR-ALL**

Sentences such as

```
        John likes x if (x supplies y) For-All
                        (y John wants y)
```

i.e. John likes anyone who supplies everything he wants, are given the obvious procedural interpretation:

```
to show John likes x
     find all y such that John wants y
     and show, for each such y, x supplies y.
```

This is equivalent to the interpretation it receives when it is rewritten in terms of negation by failure.

```
John likes x if Not (John wants y and Not (x supplies y))
```

```
i.e. to show John likes x
       show there is no y such that
                      John wants y
       and x does not supply y.
```

**IS-ALL**

The last of the extensions of Horn clause logic which we have introduced in our computer logic lessons is the feature 'Is-All', which constructs a list of all answers to a query. This list can then be manipulated: for example, by counting its elements or summing its elements if they are numbers. For example:

```
Which  (x    y Is-All (z John supplies z)
             and y has-cardinality x)
```

expresses the English question

How many items does John supply?

Because Is-All returns a list possibly containing duplicates the definition of 'has-cardinality' needs to take this into account:

```
( ) has-cardinality 0
(x|y) has-cardinality z if x belongs-to y and y has-cardinality z
(x|y) has-cardinality z1
            if Not (x belongs-to y)
            and y has-cardinality z2
            and SUM (z2 1 z1)
```

where SUM (x y z) holds if and only if $x + y = z$.

'Is-All', like 'Not' and 'For-All', is based on the closed world assumption that the database contains all the information which needs to be known about the relations under consideration. Because of the closed world assumption, these three extensions of Horn clause logic can be implemented very efficiently. Given appropriate restrictions on the contexts in which they are used, they coincide in meaning with their classical counterparts and can be interpreted wholly declaratively. A 'set-of' feature similar to 'Is-All' has also been implemented in Dec-10 Prolog by David Warren (1981).

## RELATIONSHIP WITH TRADITIONAL MATHEMATICS

Our emphasis throughout on the declarative reading of logic programs is in the spirit which emphasizes program specifications in modern software engineering and declarative query languages in modern database systems. It is in harmony, moreover, with both modern and traditional approaches to mathematics.

In contrast to the procedural approach to computing, which conflicts with the declarative style of school mathematics, the logic programming approach contributes to mathematics as it is actually taught in school. The SUM relation, for example, which can be used to add two numbers e.g.

```
Which (x SUM (2 2 x))
```

can also be used to subtract, e.g.

```
Which (x SUM (x 2 4))
Which (x SUM (2 x 4))
```

in the same way that primary school children are asked to fill in the boxes, standing for unknowns, in arithmetic relationships:

$$2 + 2 = \square$$
$$\square + 2 = 4$$
$$2 + \square = 4.$$

Compound queries, such as

```
Which ((x y ) SUM (x y 4) and SUM (0 x y))
```

can be used to illustrate the problem of solving several equations in several unknowns. Moreover, the specification of the problem, as represented by the query, can be clearly separated from the method of solution. It needs to be noted, however, that the built-in predicate SUM currently implemented in micro-Prolog will not work for this example. This is because, for the sake of efficiency, SUM will not run with more than one variable. This problem can be dealt with by defining a new SUM predicate in logic.

## THE EUCLIDEAN ALGORITHM

The Euclidean algorithm is a good example of the way in which logic programming complements and reinforces traditional mathematics teaching. It also illustrates the equally important point that classical mathematical techniques can be used to verify the correctness of logic programs much more directly than they can be used for conventional programs with side effects. A more advanced example along these same lines by Clark, McKeeman and Sickel (1982) is the derivation of an entire family of definite integration algorithms from a single specification.

The Euclidean algorithm computes the greatest common divisor of two non-negative integers. In order to understand the algorithm it is necessary first to understand its specification, namely the definition of greatest common divisor. This can be expressed in computer intelligible terms in our extension of the Horn clause subset of logic. We let

```
    GCD (x y z) express that
            z is the greatest common divisor of x and y, and
    "z divides x" express that
            z and x are non-negative integers and z divides x with remainder 0.
```

The GCD z of x and y divides both x and y and is ⩾ all such common divisors. Symbolically:

```
    GCD (x y z) if (and only if)
            z divides x and z divides y
            and (z ⩾ u) ForAll (u u divides x and u divides y)
```

Given an appropriate definition of the 'divides' relation, the specification can actually be run as a very inefficient program.

The Euclidean algorithm is more efficient. It uses the division algorithm, which

> for every pair of positive integers
> x and y such that $x > y$, computes
> integers q and r such that
> $x = q\,y + r$ and $0 \leqslant r < y$.
> q is the *quotient* and r the *remainder*.

The Euclidean algorithm is based on the following

*Theorem D. The greatest common divisor of x and y is equal to the greatest common divisor of y and r (where r is the remainder of the division of x by y as above).*

The algorithm consists in using this equality repeatedly in one direction only

> to reduce the problem of computing
> the GCD of x and y to the subproblems of
> (1) dividing x by y to obtain remainder r and
> (2) finding the GCD of y and r, until r = 0, in which case the GCD is the quotient q.

Applied, for example, to the problem of computing the GCD of 554 and 119 the algorithm generates the following sequence of divisions.

```
    554 = 4 X 119 + 68
    119 = 1 X  68 + 51
     68 = 1 X  51 + 17
     51 = 3 X  17 +  0
```

This shows that the GCD is 17.

Thus the Euclidean algorithm consists of the controlled unidirectional application of the declarative statement of Theorem D above. If we express the GCD and the division algorithm in relational form, then the equality in the statement of the theorem becomes an if-and-only-if and the algorithm reduces to the procedural interpretation of the if-half of the theorem. Let

```
Division (x q r y) express that
        x, y, q, r are non-negative integers,
        x ⩽ y and
        x = q y + r and 0 ⩽ r < y.
```

The theorem can then be expressed formally by the conditional equivalence.

```
[GCD (x y z) if and only if GCD (y r z)]
        if Division (x q y r).
```

The if-half of the equivalence gives the recursive clause of the Euclidean algorithm:

```
EA2.  GCD (x y z) if Division (x q y r)
                  and GCD (y r z)
```

the terminating case of the algorithm is given by the clause

```
EA1.  GCD (x y y) if Division (x q y 0)
```

which is a trivial consequence of the specification of GCD.

Thus the correctness of the Euclidean algorithm reduces to the problem of verifying that EA2 is a logical consequence of the definition of GCD and the theorems of arithmetic. This is a purely mathematical problem. For the sake of completeness we include the simple proof in Appendix A.

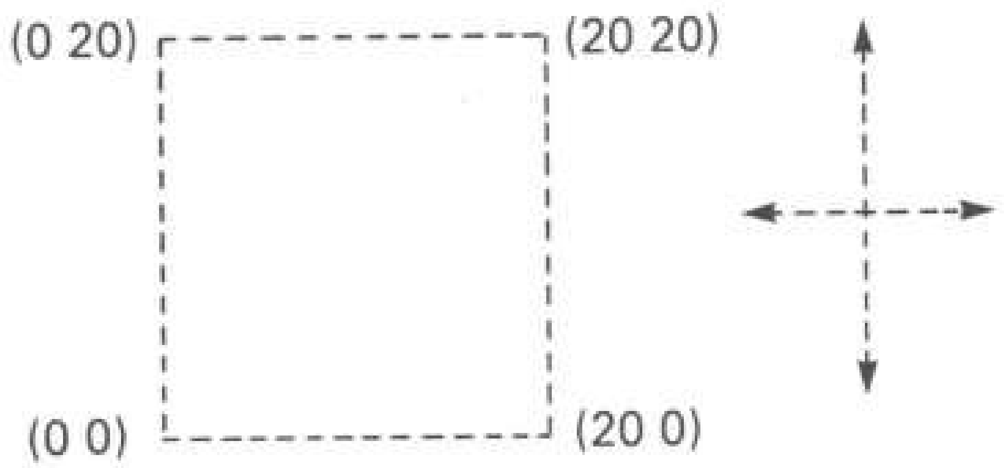
## COMPUTATIONAL GEOMETRY

Computational geometry affords a more direct comparison between the wholly procedural approach of a conventional programming language and the combined declarative and procedural approach of logic.

The purpose of the following discussion is not to advocate a particular method for doing graphics in Prolog, but to show how the procedural and declarative approaches are related. In particular, we do not presume to claim that either of the two approaches we shall discuss provide sophisticated graphics facilities. On the contrary, the facilities have been deliberately simplified in order to facilitate the discussion.

To start with, we shall show how pictures can be generated, as in a procedural language, by means of picture-plans which are sequences of visible and invisible vectors, represented by lists. When such a list is interpreted as a list of graphics characters and is output to a screen the result is a picture. This approach to generating pictures, described below, is based upon a representation used by the author in micro-Prolog using the graphics facilities of the Zenith Z89 microcomputer.

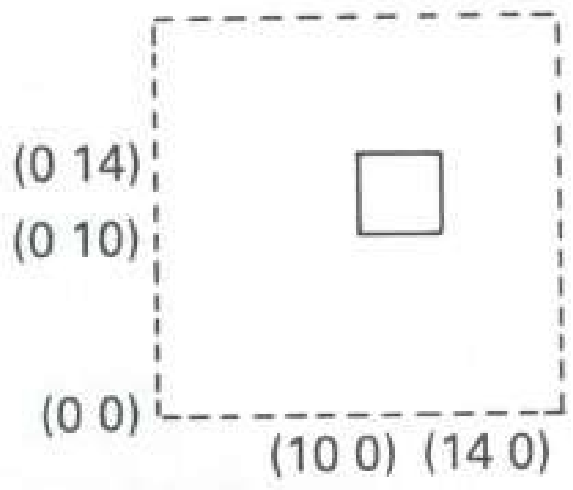
Assume, for simplicity's sake, that the screen is related to a coordinate system as illustrated below:



In Logo (Papert, 1980) if a graphics turtle is positioned at the origin and pointed north then the program segment

```
 1. PENUP                9. FORWARD 4
 2. FORWARD 12          10. RIGHT 90
 3. RIGHT 90            11. FORWARD 4
 4. FORWARD 10          12. RIGHT 90
 5. LEFT 90             13. FORWARD 4
 6. PENDOWN             14. RIGHT 90
 7. FORWARD 2           15. FORWARD 2
 8. RIGHT 90
```

generates the following picture of a square:



A similar program can be written in Prolog using its extralogical printing facilities. We prefer instead to represent the sequence of instructions as a list and print it at the top-most level of a query. This turns the drawing into an object which can be talked about, manipulated if desired, and printed as output. In order to simplify the discussion below, instead of the PENUP, PENDOWN, FORWARD, RIGHT and LEFT instructions, we use the following:

| | |
|---|---|
| V | (visible, i.e. PENDOWN) |
| I | (invisible, i.e. PENUP) |
| (N x) | (North x steps) |
| (E x) | (East x steps) |
| (S x) | (South x steps) |
| (W x) | (West x steps) |

and we specify the initial position of the turtle explicitly. Using the two place predicate

```
x names y
```

to relate a picture-plan x to the corresponding list y of graphics characters, the query

```
Which (x ((0 0) I (N 12) (E 10) V (N 2)
            (E 4) (S 4) (W 4) (N 2)) names x)
```

for example, generates the same picture of the square above. Let us call this square S, since we shall refer to it again later.

Throughout this section we distinguish between *picture-plans* which are sequences of actions and *pictures* which such picture-plans generate. In general, a picture-plan is named by a list whose first item gives the coordinates of the *origin* from which the plan starts, whose second item specifies the *mode* (one of V or I) which indicates whether subsequent actions are Visible or Invisible and whose remaining items are either

(1) a mode or
(2) a *vector* which is a pair giving
    direction (one of N, E, S, W) and
    distance (a positive integer).

Several picture-plans can result in the same picture. For example, both of the lists

```
((0 0) I (E 14) (N 14) V (W 4) (S 4) (E 4) N 4))
((10 10) V (N 4) (E 4) (S 4) (W 4))
```

describe different plans of the same square S. Picture-plans can also be described by means of general laws. For example, the relationship

```
Square-plan (x y z)
```

which holds when z is a picture-plan of a square with south-west corner at point x having side of length y is defined by

```
P:        Square-plan (x y (x V (N y) (E y) (S y) (W y).))
```

The same picture of the square S can now be obtained by asking

```
Which  (x Square-plan ((10 10) 4 y)
          and y names x).
```

To tell whether two plans generate the same picture we can print them and compare the results. This may not be satisfactory, however, if the pictures are complicated and the screen is erased between pictures. Moreover, it will not work for showing that a general program like P above is correct. For this we need a specification which is different from the program. We need moreover, a formal notion of picture which is distinct from that of picture-plan.

A *picture* can be regarded as a set of (visible) line segments and can be named by a list each item of which is a pair of points. For example, the list

```
( ((10 10) (10 14))
  ((10 14) (14 14))
  ((10 10) (14 10))
  ((14 10) (14 14)) )
```

names the *picture* of the square S. Thus a picture is an order-independent set of line segments represented by pairs of end points. The order of the end-points in a line segment does not matter. Notice that we use the same term 'picture' both for the visual pattern produced on a screen by printing the object 'named' by a picture-plan and for its mathematical representation as a set of line segments. Which of these two notions is intended should be clear from the context.

The result of a picture-plan is a picture. This relationship

```
x draws y
```

where y is the picture produced by the picture-plan x, can be described straightforwardly by means of Horn clauses with negation by failure. Rather than give the definition here we illustrate it by means of an example, leaving the definition to Appendix B. The application of the relationship to the plan

```
((0 0) I (N 12) (E 10) V (N 2)
        (E 4) (S 4) (W 4) (N 2))
```

can be decomposed into a sequence of three steps

*Step 1* (a) Replace every vector pair of the form (direction distance) by a 3-tuple (base mode end-point) where 'base' is the base-point of the vector, 'mode' describes whether the vector is visible or invisible and 'end-point' is the end-point of the vector,

(b) absorbing the origin into the base of the first 3-tuple, and

```
        (c) absorbing individual modes into subsequent 3-tuples:
            (((0   0) I (0  12)) (( 0 12) I (10 12))
             ((10 12) V (10 14)) ((10 14) V (14 14))
             ((14 14) V (14 10)) ((14 10) V (10 10))
             ((10 10) V (10 12)) )
```

*Step 2* (a) Delete all 3 tuples with mode I and

(b) replace every 3-tuple of the form (base V end-point) by the pair (base end-point):

```
            (((10 12) (10 14)) ((10 14) (14 14))
             ((14 14) (14 10)) ((14 10) (10 10))
             ((10 10) (10 12)))
```

*Step 3* Coalesce colinear, contiguous line segments:

```
        (((10 10) (10 14)) ((10 14) (14 14))
         ((14 14) (14 10)) ((14 10) (10 10)))
```

We can use the concept of picture and the 'draws' relation to generate pictures directly from specifications of the line segments they contain. For example, the query

```
Which (x y draws (((10 10) (10 14))
                  ((10 14) (14 14))
                  ((14 14) (14 10))
                  ((14 10) (10 10)))
           and y names x)
```

generates the picture of the square S.

To prove that the definition P of the 'square-plan' relation correctly defines the notion of square, we need a specification. To simplify matters and to save space, given the restricted nature of the pictures we can draw, it suffices to specify a *square* as a set of four line segments of equal length, connecting four points, every point being the end-point of exactly two segments. More formally, if we let the relation

Square (x y z)

express that z is a square with south-west corner at point x having sides of length y, then the notion of *square* is specified by

```
Square ((x1 y1) y z) if (and only if)
        SUM (x1 y x2)
    and SUM (y1 y y2)
```

```
        and z equals (((x1 y1) (x1 y2))
                      ((x1 y2) (x2 y2))
                      ((x2 y2) (x2 y1))
                      ((x2 y1) (x1 y1))).
```

Here 'x equals y' simply expresses that the lists x and y contain the same line segments, where the order of the end-points of line segments does not matter. Its Horn clause 'definition' is given in Appendix B.

We can use our intuitive understanding of the 'draws' and the 'equals' relations to give an informal correctness proof that

```
    For all x, y, z1 and z2
    if Square-plan (x y z1)
    and z1 draws z2
    then Square (x y z2)

    Absorb ((base mode (direction distance) | list)
            ((base mode end-point) | newlist)
           if Move (base direction distance end-point)
         and Absorb ((end-point mode | list) newlist)
    Absorb ((base mode new-mode          | list) newlist)
           if Absorb ((base new-mode     | list) newlist)
    Absorb ((base mode) ( ))
    Move ((x y) N z (x y1)) if SUM (y   z y1)
    Move ((x y) S  z (x y1)) if SUM (y1 z y)
    Move ((x y) E  z (x1 y)) if SUM (x   z x1)
    Move ((x y) W z (x1 y)) if SUM (x1 z x)

    Visible (((base I end-point) | list) newlist)
            if Visible (list newlist)

    Visible (((base V end-point) | list)
             ((base end-point) | newlist))
          if  Visible (list newlist)

    Visible (( ) ( ))

    Coalesce (lines newlines)
              if    Select      (lines line1 interlines)
              and Select        (interline1 line2 restlines)
              and Combine    (line1 line2 line3)
              and Coalesce    ((line3 | restlines) newlines)

    Coalesce (lines lines)
              if    Not (Select (lines line1 interlines)
              and Select (interlines line2 restlines)
              and Combine (line1 line 2 line3))
```

```
Select ((x|y) x y)
Select ((x|y) u (x|v)) if Select (y u v)

Combine (line1 line2 ((x1 y1) (x1 y3)))
                 if   (line1 line2) equals
                      (((x1 y1) (x1 y2)) ((x1 y2) (x1 y3)))
                 and y1 < y2 and y2 < y3

Combine (line1 line2 ((x1 y1) (x1 y2)))
                 if   (line1 line2) equals
                      (((x1 y1) (x1 y2)) ((x1 y1) (x1 y3)))
                 and y1 < y3 and y3 < y2

Combine (line1 line2 ((x1 y1) (x3 y1)))
                 if   (line1 line2) equals
                      (((x1 y1) (x2 y1)) ((x2 y1) (x3 y1)))
                 and x1 < x2 and x2 < x3

Combine (line1 line2 ((x1 y1) (x2 y1)))
                 if   (line1 line2) equals
                      (((x1 y1) (x2 y1)) ((x1 y1) (x3 y1)))
                 and x1 < x3 and x3 < x2

x equals y if x permutation-of y
( ) permutation-of ( )
(x|y) permutation-of z if x same-seg x'
                          and Select (z x' z')
                          and y permutation-of z'
x same-seg x
(p1 p2) same-seg (p2 p1)
```

*Proof.* Assume Square-plan ((c1 c2) L A ) and A draws B. We need to show

Square ((c1 c2) L B ).

We must assume that sentence P is the *only* sentence defining the Square-plan relation. This implies that

A = ((c1 c2) V (N L) (E L) (S L) (W L).)

Since A draws B,

```
    B equals (((c1 c2)       (c1 c2+L))
              ((c1 c2+L)     (c1+L c2+L))
              ((c1+L c2+L)   (c1+L c2))
              ((c1+L c2)     (c1 c2)))
```

which directly implies

Square ((c1 c2) L B).

In this example both the Square-plan program and the Square specification can be expressed in Horn clause logic. In a conventional programming language such as Logo only the analogue of the Square-plan program can be represented. No attention is given to the specification, disregarding concerns of program correctness; and no attention is given to classical geometry, disregarding actual mathematical practice.

## ACKNOWLEDGEMENT

This paper is based on an invited lecture at the 1982 European Conference on Artificial Intelligence at Orsay, France. The author is grateful to the Science Research Council and the Nuffield Foundation for supporting the Schools Computer Logic Project.

## APPENDIX A – PROOF OF THEOREM EA2

We assume the normal properties of division as well as the specification of GCD and present the proof in conventional informal mathematical style and notation.

Assume

(1) $A = QB+R$, where $0 \leqslant R < B$ and
(2) C is GCD of B and R

we want to show

C is GCD of A and B.

Because of (2) and the definition of GCD,

C divides both B and R.

But then C divides the right-hand side of equation (1) and therefore it divides left-hand side as well – namely

C divides A.

But then C is a common divisor of both A and B.

It remains to show that C is $\geqslant$ any D which divides both A and B. So suppose, to the contrary, that some $D > C$ divides both A and B. Then D divides the left-hand side of

$$A - QB = R$$

and therefore it divides the right-hand side as well, i.e.

D divides R and B

and since it is more than C, C cannot be GCD of R and B, which contradicts our original assumption.

## APPENDIX B

Here to aid readability lower-case identifiers are used for variables. Upper-case letters are used for constants.

```
x draws y if Absorb (x x1) and Visible (x1 x2)
          and Coalesce (x2 y).

Absorb( (base mode (direction distance) | list)
      ((base mode end-point) | newlist)
      if Move (base direction distance end-point)
   and Absorb ((end-point mode | list) newlist)

Absorb( (base mode new-mode      | list) newlist)
      if Absorb ( (base new-mode | list) newlist)

Absorb( (base mode) ( ))

Move ( (x y) N z (x y1)) if SUM (y  z  y1)
Move ( (x y) S  z (x y1)) if SUM (y1 z y)
Move ( (x y) E z (x1 y)) if SUM (x   z x1)
Move ( (x y) W z (x1 y)) if SUM (x1 z x)

Visible ( ((base I end-point) | list) newlist)
      If Visible (list newlist)

Visible ( ((base V end-point) | list)
       ((base end-point) |  newlist) )
      if Visible (list newlist)

Visible ( ( )   ( ) )

Coalesce (lines newlines)
        if Select       (lines line1 interlines)
      and Select       (interline1 line2 restlines)
      and Combine     (line1 line2 line3)
      and Coalesce    ( (line3 | restlines) newlines)

Coalesce (lines lines)
        if Not(Select  (lines line1 interlines)
      and Select       (interlines line2 restlines)
      and Combine     (line1 line2 line3) )

Select   ( (x|y) x y)
Select   ( (x|y) u (x|v)) if Select (y u v)

Combine  (line1 line2 ((x1 y1) (x1 y3)) )
            if (line1 line2) equals
               (((x1 y1) (x1 y2)) ((x1 y2) (x1 y3)))
          and y1 < y2 and y2 < y3
Combine (line1 line2 ((x1 y1) (x1 y2)))
            if (line1 line2) equals
               (((x1 y1) (x1 y2)) ((x1 y1) (x1 y3)))
           and y1 < y3 and y3 < y2

Combine (line1 line2 ((x1 y1) (x3 y1)))
            if (line1 line2) equals
               (((x1 y1) (x2 y1)) ((x2 y1) (x3 y1)))
          and x1 < x2 and x2 < x3

Combine (line1 line2 **x1 y1) (x2 y1)))
           if (line1 line2) equals
              (((x1 y1) (x2 y1)) ((x1 y1) (x3 y1)))
         and x1 < x3 and x3 < x2

x equals y if x permutation-of y
( ) permutation of ( )
(x|y) permutation of z  if x same-seg x'
                        and Select (z x' z')
                        and y permutation-of z'

x same-seg x
(p1 p2) same seg (p2 p1)
```