RELATIONAL PROGRAMMING
illustrated by a program for
the game of Mastermind

by

M.H. van Emden

Research Report CS-78-48
Department of Computer Science
University of Waterloo

December 1978
(Revised May 1979)

1. Introduction

Many difficulties in programming are caused by the use of imperative languages (those which are based on commands) such as Fortran, Algol, or Pascal. These difficulties can be avoided by using a definitional language such as Lisp or Prolog. Lisp is based on the lambda-calculus and is typically used for the definition of functions. Prolog is based on first-order predicate logic and is typically used for the definition of relations.

This paper aims to show an advantage of specifying a relation instead of a function: the same relation can specify several different functions, depending on which arguments are given. As a result, the Prolog interpreter can use the same relational specification to compute several of the different functions implied in the relation. Several examples of this phenomenon are exhibited and discussed in this paper.

Prolog programs are an essential part of this paper. Between different implementations of Prolog there are minor variations in syntax and in the effect of system-defined predicates. The original Marseille implementation [GDM] is the common ancestor of the systems developed in Budapest [PPL], Edinburgh [CLP], and Waterloo [IOP]. The programs in this paper have been run on the latter system. A more advanced version is IC-Prolog [ICP], which is being developed in Imperial College, London, by K.L. Clark, R.A. Kowalski, and F.G. McCabe. Another recent development is by J.A. Robinson and E. Sibert in the University of Syracuse, where an implementation of logic programming is embedded in Lisp.

Prolog is based on Kowalski's proposal [PLPL] for using logic as a programming language, which , in turn, is based on J.A. Robinson's resolution principle [MOL]. Although not conceptually related, Prolog has similarities to several earlier systems, such as [ABSYS], [PLANNER], and [ABSET].

2. The Principle

The principle of relational programming is explained by means of the following binary relation between natural numbers:

$$R = \{(1,1), (2,4), (3,9), (4,16)\}$$

Depending on which argument is given, the relation specifies a subset of the squaring function, or a subset of the square-root function.

In logic the relation can be specified by the conjunction of the following atomic formulas:

```
R(1,1).
R(2,4).
R(3,9).
R(4,16).
```

The atomic formulas are special cases of clauses, the components of a Prolog program. Not only in this example, but in general also, Prolog programs are regarded as specifying relations.

The Prolog interpreter can be instructed to find a y such that R(3,y) is provable from the specification. It will respond with y = 9, thus computing a value of the squaring function. Or the Prolog interpreter may be instructed to find an x such that R(x,16) is provable from the specification. It will respond with x = 4, thus computing a value of the square-root function.

Both of these computations are, of course, nothing but table look-ups. The remainder of this paper is devoted to less trivial applications, which can, however, be viewed as look-up in virtual tables implicit in specifications

consisting of clauses which have a less restricted form than those of the present example. This point of view is elaborated in [CDI].

3. A simple example of relational programming

In logic programs, as in first-order logic, terms denote objects. The syntax requires that a term is either a constant (in Prolog an identifier or a decimal number), or a variable (in Prolog a constant preceded by an asterisk), or $f(t_1,\ldots,t_m)$ where f is an m-place functor and $t_1,\ldots,t_m$ are terms.

For example,

.(c,.(b,nil))

is a term, where "." is a 2-place functor and b,c,nil are constants. Prolog allows infix notation for 2-place functors so that we can also write

c.(b.nil)

As a further convenience, we are allowed to write

$t_1. \ \ldots \ .t_n$

and to specify whether it means

$t_1.(t_2.(\ldots \ .t_n)\ldots)$

or

$(\ldots(t_1.t_2). \ \ldots \ .t_{n-1}).t_n$

Throughout this paper we assume that the former option is in force.

In this section we discuss a specification of the relation, called append, between three lists which holds if the last list is the result of appending the first two. In this example the objects to be denoted by terms are lists. A nonempty list is a composite object: it consists of a head, which is the first element, and a tail, which is the list of the remaining elements. A non-empty list is denoted by a term of the form $t_1.t_2$ where the term t_1 is the head and the term t_2 is the tail. An empty list has no components and is therefore denoted by a constant (in this paper, as is usual, by nil).

The logic program specifying the append relation is the conjunction of the following two clauses:

```
append(nil,*y,*y).
append(*u.*x,*y,*u.*z) ← append(*x,*y,*z).
```
(3.1)

In general we are concerned with clauses of the form

$$A \leftarrow B_1 \ \& \ \ldots \ \& \ B_n, \qquad n \geq 0$$

where $A,B_1,\ldots,B_n$ are atomic formulas containing the variables $x_1,\ldots,x_p$, where $p \geq 0$. The clause should be read as

$$\text{for all } x_1,\ldots,x_p, \ A \ \text{ if } \ (B_1 \text{ and } \ldots \text{ and } B_n)$$

In case $n = 0$ we drop the left arrow. In that case the clauses should be read as an unconditional assertion. It should now be clear that the clauses (3.1) are true of the append relation between lists.

A program, which consists of clauses, is activated by a goal statement, which has the form

$$\leftarrow A_1 \ \& \ \ldots \ \& \ A_k, \qquad k \geq 0$$

where $A_1,\ldots,A_k$ are atomic formulas, called goals. One of the goals in a non-empty goal statement is distinguished and is called the selected goal.

From a goal statement

(3.2) $\quad \leftarrow A_1 \;\&\; \ldots \;\&\; A_k$

with selected goal A_i there may be derived the goal statement

(3.3) $\quad \leftarrow (A_1 \;\&\; \ldots \;\&\; A_{i-1} \;\&\; B_1 \;\&\; \ldots \;\&\; B_n \;\&\; A_{i+1} \;\&\; \ldots \;\&\; A_k)t_0$

if the program contains a clause

$$A \leftarrow B_1 \;\&\; \ldots \;\&\; B_n, \qquad n \geq 0,$$

which matches the goal A_i. This is said to be the case if $At = A_it$ for some substitution t of terms for variables. If such a t exists, then there also exists a 'most general' one, here called t_0, which is such that At can be obtained by substitution (possibly null) from At_0; t_0 is the substitution produced by the derivation of (3.3) from (3.2).

A proof is a sequence of goal statements, ending in an empty goal statement, and such that each successive goal statement is derived from the preceding one. Suppose now that a proof exists with $\leftarrow A_1 \;\&\; \ldots \;\&\; A_k$ as initial goal statement and with $t_0,\ldots,t_N$ as substitutions. What is now proved by the proof is that

$$\text{for all } x_1,\ldots,x_q,\ (A_1 \;\&\; \ldots \;\&\; A_k)t_0 \ldots t_N$$

follows from the conjunction of the clauses in the program, where $x_1,\ldots,x_q$ $(q \geq 0)$ are the variables in $(A_1 \;\&\; \ldots \;\&\; A_k)t_0 \ldots t_N$.

Let us look at an example, using the logic program (3.1), where we require the result of appending the three lists c.nil, a.nil, and b.nil. This requirement is specified by the initial goal statement

← append(a.nil,b.nil,*x) & append(c.nil,*x,*y)

If a proof is found, then the composition of all substitutions in the proof substitutes for *y the required result.

For the Prolog interpreter the selected goal is always the leftmost. Hence the interpreter will attempt initially to match the leftmost goal in the above goal statement with the first clause of (3.1), which is not possible. The second clause does match, deriving the goal statement

← append(nil,b.nil,*x1) & append(c.nil,a.*x1,*y)

Now the first clause matches the leftmost goal, deriving

← append(c.nil,a.b.nil,*y)

The next goal statement is

← append(nil,a.b.nil,*y1)

with a substitution replacing *y by c.*y1. The selected goal now matches the first clause, so that the next goal statement is empty. A proof has been found. The variable *y in the initial goal statement is replaced by c.a.b.nil.

So much for the basic mechanism of Prolog. Here we are concerned with relational programming; that is, we want to make use of the fact that (3.1)

specifies a relation between the three arguments of append, rather than a function from the first two to the third. Take for example the goal statement

← append(a.nil,*y,a.b.c.nil)

In finding a proof, Prolog will substitute b.c.nil for *y, thus performing list subtraction. Below (3.4, 3.5, 3.6) we list several other examples of goal statements causing Prolog to compute functions other than the append function, all by means of the same relational specification (3.1).

(3.4) ← append(*x,c.nil,a.b.cnil)

substitution: *x := a.b.nil

(3.5) ← append(*u,c.*v,a.b.c.nil)

substitution: *u := a.b.nil

*v := nil

(3.6) ← append(*u,b.c.nil,*v) & append(*v,*w,a.b.c.d.nil)

substitution: *u := a.nil

*v := a.b.c.nil

*w := d.nil

The goal statement (3.4) causes another form of list subtraction. The goal statement (3.5) has the effect of checking whether c occurs in a.b.c.nil; this suggests the following definition of list membership:

```
append(nil,*y,*y).
append(*u.*x,*y,*u.*z) ← append(*x,*y,*z).
member(*c,*w) ← append(*u,*c.*v,*w).
```

The goal statement (3.6) has the effect of checking whether b.c.nil is a sublist of a.b.c.d.nil; this suggests the following definition of the sublist relation:

```
append(nil,*y,*y).
append(*u.*x,*y,*u.*z) ← append(*x,*y,*z).
sublist(*x,*z) ← append(*u,*x,*v) & append(*v,*w,*z).
```

This definition of sublist is not restricted to completely specified lists as first argument. For example,

```
← sublist(*x.*y.*x,nil, m.a.d.a.m.nil)
```

will result in

```
*x := a
*y := d
```

In other words, sublist can be used to search for incompletely specified sublists: things that may well be called "patterns".

We have shown that a single specification can be used to compute a variety of functions, each of which would require a different program in a conventional language. We call relational programming the technique of using this phenomenon. Another advantage is that the more general relational specification may be easier to find than the particular function required.

Prolog is far from perfect as a vehicle for relational programming. Finding a proof depends on having in each goal statement the correct choice of selected goal or, given the selected goal, using the correct choice of clause in case more than one matches. Prolog often fails to find a proof because it always selects the leftmost goal and because it always tries to match the clauses in the order in which they occur in the program. IC-Prolog [ICP] will find proofs in cases where Prolog does not because it is more flexible in determining the selected goal.

4. The game of Mastermind

In the abstract game of Mastermind the following types of object exist:

CODE = PROBE = the set of ordered 4-tuples with elements in a set of colours

SCORE = the set of ordered pairs with elements in the set of numbers 0 through 4

f: PROBE x CODE → SCORE; we call f the 'scoring function'.

The elements of the ordered 4-tuples correspond to the 'code pegs' of the concrete game, and they may be black, blue, green, red, white, or yellow. In the abstract game we take the set of colours to be

{BLACK, BLUE, GREEN, RED, WHITE, YELLOW}

The first (second) component of an element of SCORE corresponds to the number of 'black (white) key pegs' of the concrete game. We will find it convenient to represent in the abstract game these numbers in successor notation, because then the relation between predecessor and successor can be specified succinctly, without explicitly referring to the sum relation. The successor

function is denoted by + so that +(x) is the successor of x in functional notation. However, suffix notation is more concise and traditional; therefore we represent the set of numbers 0 through 4 as

$$\{0,\ 0+,\ 0++,\ 0+++,\ 0++++\}$$

The scoring function has the property that the higher the score, the greater the similarity between its arguments. This statement is only intended to help the intuition, as it is formally meaningless without a definition of order among scores or of similarity between codes and probes. The value $f(p,C)$ of the scoring function contains a black key peg for every position where p and C have the same colour. Such an occurrence is called a 'strong match'. For every one of the remaining positions, $f(p,C)$ contains a white key peg for every element of p with the same colour as an element of C. Such an occurrence is called a 'weak match'.

The game is played as follows. There are two players, the Codemaker and the Codebreaker. The Codemaker selects a code C which is concealed from the Codebreaker. The Codebreaker can obtain information about C by selecting a probe p in response to which the Codemaker reveals the result $s = f(p,C)$ of the scoring function.

This is repeated until the Codebreaker has selected a probe equal to C. In other words, the Codebreaker constructs a sequence $p_1,\ldots,p_n$ of probes with $p_i \neq C$ for $i \neq n$ and $p_n = C$. The Codemaker constructs a sequence $s_1,\ldots,s_n$ such that $s_i = f(p_i,C)$. The selection of p_i by the Codebreaker may depend on $(p_1,s_1),\ldots,(p_{i-1},s_{i-1})$. It is the Codebreaker's objective to make n as small as possible.

5. A logic program for the scoring function

Logic programs compute relations. Therefore, if one wants to compute a function, it has to be expressed as a relation. The logic program for the scoring function defines a relation MM such that

MM(p,c,s) iff f(p,c) = s,

where f is the scoring function discussed before. The relation MM is defined by the clause

```
MM(*P,*C,*S1.*S2) ← BLACKS(*P,*C,*P1,*C1,*S1)
                    & WHITES(*P1,*C1,*S2).
```

The first component of the score is s1, the number of black key pegs. BLACKS is true if s1 is the number of strong matches between p and c and if p1 and c1 are the results of removing the strongly matching elements from p and c respectively. WHITES is true if s2 is the number of weak matches between p1 and c1. This clause reflects the informal description of f given in the previous section.

The following statements in logic, which are true of MM and of the auxiliary relations, have been run as a Prolog program for computing the scoring function. Note that we represent an ordered n-tuple consisting of the colours $c_1,\ldots,c_n$ by the term $c_1.\ \ldots\ .\ c_n.nil$; that is, just as we represented lists in section 3.

```
OP(+,SUFFIX,150).
OP(=,RL,150).

MM(*P,*C,*S1.*S2) <- BLACKS(*P,*C,*P1,*C1,*S1) & WHITES(*P1,*C1,*S2).

BLACKS(NIL,NIL,NIL,NIL,0).
BLACKS(*U.*P,*U.*C,*P1,*C1,*S+) <- BLACKS(*P,*C,*P1,*C1,*S).
BLACKS(*U.*P,*V.*C,*U.*P1,*V.*C1,*S) <- NOT(*U=*V)
                                        & BLACKS(*P,*C,*P1,*C1,*S).

WHITES(NIL,*C,0).
WHITES(*U.*P,*C,*S+) <- DEL(*U,*C,*C1) & WHITES(*P,*C1,*S).
WHITES(*U.*P,*C,*S) <- NONMEM(*U,*C) & WHITES(*P,*C,*S).

DEL(*U,*U.*Y,*Y).
DEL(*U,*V.*Y,*V.*Y1) <- NOT(*U=*V) & DEL(*U,*Y,*Y1).

NONMEM(*U,NIL).
NONMEM(*U,*V1.*V) <- NOT(*U=*V1) & NONMEM(*U,*V).

*X=*X.
```

Fig. 1: Prolog program for the scoring function

6. A codebreaker obtained by relational programming

Suppose we want to obtain a program playing the part of the Codebreaker. Then we have to devise a playing strategy, that is, some function with the sequence $(p_1,s_1),\ldots,(p_{i-1},s_{i-1})$ as argument and with a value which can be used as value for p_i, the next probe. We will use a strategy reported in [EFP] which is to take any p_i such that $f(p_j,p_i) = s_j$ for $j = 1,\ldots,i-1$, if $i \geq 2$. The first probe is arbitrary. In the first place, such a p_i always exists, because, for example, the unknown code has this property. In the second place, such a p_i can be expected to be, in some sense, close

to the unknown code, the more so the larger i is.

This last observation can be made more precise if we consider the first component of the score, namely the number s' of black key pegs. Note that $4-s'$ is the so-called Hamming distance, which is a metric, between fourtuples of colours. The set of p_i such that for $j = 1,\ldots,i-1$, $f(p_j,p_i) = s_j$ is therefore contained in the set of codes having given distances to the points $p_1,\ldots,p_{i-1}$ in a metric space. This set contains the unknown code and it can be expected to be smaller for larger i.

The strategy therefore requires the following equation to be solved for x: $f(p,x) = s$. We already wrote a program for solving for x: $f(p,c) = x$. Apparently, the relation MM, introduced for a logic program to compute the scoring function, also specifies the computation needed by a Codebreaker.

It should now be clear why we have chosen the game of Mastermind as a case study in relational programming: we aim to obtain a program for the more difficult Codebreaker's part by specifying in relational form the easily programmed scoring function and then to use this relation with the probe and score as given arguments to obtain a guess at the unknown code.

However, it would be a mistake to believe that the program in section 5 can be used as a code-breaking program. The reason is that we have inadvertently specified a relation different from the one intended. We need a relation which is a subset of the Cartesian product PROBE x CODE x SCORE. PROBE and CODE contain only fourtuples of colours. It is apparent that the relation specified in section 5 can have in its first two argument places tuples containing arbitrary elements, not necessarily colours. The relation is usable for computing the scoring function because then the first two arguments are given, and can be given (as required in this

particular application) as tuples of colours.

The relation specified in section 5 is too large, but a goal specifying that the scoring function is to be computed happens to restrict the relation in the desired way. However, if we want to use a goal ← MM(p,x,s) to solve for x the equation f(p,x) = s, with p and s given, then we cannot expect the desired result, because according to the specification in section 5, x can be a tuple containing arbitrary elements. However, if we would extend the specification so that indeed x is forced to consist of colours only, then we would be able to use the modified specification to solve for x both f(p,c) = x and f(p,x) = s. That is, we could then use the relation MM to play both sides of Mastermind.

Let us see how we can correct this deficiency in our previous specification of MM . The condition

```
not(*u = *v)
```

with u equal to a colour is satisfied by values for v which are arbitrary objects which are not necessarily colours. Hence we change occurrences of this condition to, say, diff(u,v) and we add the clause

```
diff(*u,*v) ← colour(*u) & colour(*v) & not(*u = *v).
```

and specify also explicitly which colours exist by adding the clauses

```
colour(black). colour(blue). colour(green).
colour(red). colour(white). colour(yellow).
```

A specification of the relation MM, which is suitable for playing both the Codemaker's and the Codebreaker's parts, can be found as part of the complete Prolog program for Mastermind listed in the next section.

7. The complete program

We now have a Prolog program which can solve for x f(p,c) = x by the goal statement

```
← MM(*p,*c,*x)
```

and also can solve for x f(p,x) = s by the goal statement

```
← MM(*p,*x,*s)
```

We continue towards a complete program for Mastermind. As a first step we define the relation between a sequence of (probe,score)-pairs.

$$(p_1,s_1),\ldots,(p_i,s_i)$$

and a candidate code p having the property that

$$f(p_1,p) = s_1,\ldots,f(p_i,p) = s_i$$

The desired relation is defined by

```
candcode(nil,*).
candcode((*pl.*sl).*ps,*p) ← mm(*pl,*p,*sl) & candcode(*ps,*p).
```

Let us call a candidate solution a sequence

$$(p_1,s_1),\ldots,(p_i,s_i)$$

of (probe,score)-pairs such that

$$f(p_1,p_k) = s_1,\ldots,f(p_{k-1},p_k) = s_{k-1}, \text{ for } k = 2,..,i-1.$$

That is, each probe is a candidate code with respect to the preceding sequence of (probe,score)-pairs. A candidate solution is a solution if the last score has at least four black pegs, that is, if the last probe is equal to the code.

For us it is important that a candidate solution be extendable to a solution. Of the property of being extendable we can say that

```
extendable((*.(*++++.*)).*).
extendable(*cs) ← candcode(*cs,*cc) & score(*cc,*s)
                    & extendable((*cc.*s).*cs).
score(*p,*s) ← code(*c) & mm(*p,*c,*s).
```

A note on notation: Because each clause is, separately from the other clauses, universally quantified, a variable name is only meaningful within a clause. It follows that the name of a variable which occurs only once in a clause, is immaterial, and hence can be omitted. Only the asterisk is written; the variable is anonymous. Conversely, each occurrence of an anonymous variable in a clause stands for a variable different from any other variable in the clause, anonymous or not.

With respect to a given candidate solution there are typically several possible candidate codes. The above simple definition of 'extendable' has the disadvantage that it does not extend with a best, but rather with any, candidate code. There is hence no guarantee that only reasonably short solutions are specified by 'extendable'. Our experience shows that with most codes 'extendable' gives a solution of length five. An exception was found with a code consisting of equal colours. D.E. Knuth was quoted [EFP] as having found that a solution of length five is always possible. In order to guarantee that our solutions do not exceed a given bound, we have restricted the above definition of 'extendable' to mean: extendable within the number of steps determined by an additional third argument.

The complete program is listed below in two parts. Only the part needed for the definition of 'extendable' is of interest from the point of view of relational programming. Yet a fairly large additional part if required for a program that interfaces with a client not familiar with its inner mechanisms. This part, labelled 'interactive manager' is also done in Prolog, though it is hardly an example of definitional programming. It has also been listed in full in order to show that for this kind of programming task Prolog is at least serviceable, although usually not particularly inspiring.

An exception is the way in which the backtracking of Prolog allows one to program a check on the correctness of input. For example, in PLAY it is desirable to check whether the *X produced by READ is correct. If not, CHECKSEED causes a complaint to appear and fails, so that backtracking causes READ to be reactivated, giving the user another opportunity for entering something.

In order to be able to understand the interactive manager one has to know some of the built-in predicates of the Waterloo Prolog interpreter; see Appendix 1 for the relevant excerpts from [IOP]. See Appendix 2 for the control flow of the interactive manager.

```
OP(+,SUFFIX,150).      /*  + DEFINED AS SUFFIX OPERATOR  */
OP(=,RL,150).          /*  = DEFINED AS INFIX OPERATOR ASSOCIATING
                           FROM RIGHT TO LEFT  */

EXTENDABLE((*P.(*++++.*)).*,*+)
          <- WRITECH('THE CODE MUST BE: ') & CHECKCODE(*P0,*P) & WRITE(*P0).
EXTENDABLE(*CS,*N+) <- CANDCODE(*CS,*CC) & WRITECH('MY NEXT PROBE IS: ')
                     & CHECKCODE(*C,*CC) & WRITECH(*C) & WRITECH('; SCORE: ')
                     & SCORE(*CC,*S) & WRITESCORE(*S)
                     & *M+=*N & IS(*M,*MD) & WRITECH(*MD)
                     & WRITE(' TRIES TO GO')
                     & EXTENDABLE((*CC.*S).*CS,*N).

CANDCODE(NIL,*).
CANDCODE((*P1.*S1).*PS,*P) <- MM(*P1,*P,*S1) & CANDCODE(*PS,*P).

SCORE(*P,*S) <- CODE(*C) & MM(*P,*C,*S).


/*************************************************************************/
/* BEGINNING OF DEFINITION OF SCORING RELATION */

MM(*P,*C,*S1.*S2) <- BLACKS(*P,*C,*P1,*C1,*S1) & WHITES(*P1,*C1,*S2).

BLACKS(NIL,NIL,NIL,NIL,0).
BLACKS(*U.*P,*U.*C,*P1,*C1,*S+) <- BLACKS(*P,*C,*P1,*C1,*S).
BLACKS(*U.*P,*V.*C,*U.*P1,*V.*C1,*S) <- DIFF(*U,*V)
                                      & BLACKS(*P,*C,*P1,*C1,*S).

WHITES(NIL,*C,0).
WHITES(*U.*P,*C,*S+) <- DEL(*U,*C,*C1) & WHITES(*P,*C1,*S).
WHITES(*U.*P,*C,*S) <- NONMEM(*U,*C) & WHITES(*P,*C,*S).

/* DEL(U,Y,Y1) IF Y1 IS THE RESULT OF DELETING U FROM LIST Y */
DEL(*U,*U.*Y,*Y).
DEL(*U,*V.*Y,*V.*Y1) <- DIFF(*U,*V) & DEL(*U,*Y,*Y1).

/* NONMEM(U,V) IF U IS NOT A MEMBER OF LIST Y */
NONMEM(*U,NIL).
NONMEM(*U,*V1.*V) <- DIFF(*U,*V1) & NONMEM(*U,*V).

DIFF(*U,*V) <- COLOUR(*U) & COLOUR(*V) & NOT(*U=*V).

COLOUR(BLACK). COLOUR(BLUE). COLOUR(GREEN).
COLOUR(RED). COLOUR(WHITE). COLOUR(YELLOW).

*X=*X.

/* END OF DEFINITION OF SCORING RELATION */
/*************************************************************************/
```

Fig. 2: Main part of Mastermind program.

```
/*INTERACTIVE MANAGER*/

PLAY <- WRITE('MASTERMIND AT YOUR SERVICE')
      & WRITE('ENTER AN ARBITRARY NUMBER BETWEEN 0 AND 16383')
      & READ(*X) & CHECKSEED(*X) & ADDAX(SEED(*X))
      & WRITECH('EXAMPLE FORMAT FOR ENTERING CODE: ')
      & WRITE('YELLOW.BLUE.WHITE.BLACK')
      & PLAY1.

PLAY1 <- WRITECH('DO YOU WANT TO MAKE OR BREAK CODES? ')
       & WRITE('ANSWER MAKE. OR ANSWER BREAK')
       & READ(*X) & CHECKMB(*X) & START(*X).

START(MAKE) <- / & WRITE('ENTER CODE; I PROMISE NOT TO LOOK') & READ(*C0)
             & CHECKCODE(*C0,*C) & ADDAX(CODE(*C)) & GENCODE(*P) & SCORE(*P,*S)
             & WRITECH('MY FIRST PROBE IS: ')&CHECKCODE(*P0,*P) & WRITECH(*P0)
             & WRITECH('; SCORE: ') & WRITESCORE(*S)
             & EXTENDABLE((*P.*S).NIL,0+++++) & DELAX(CODE(*)) & ASK.

START(BREAK) <- GENCODE(*C) & ADDAX(CODE(*C))
              & WRITE('ENTER FIRST PROBE') & READ(*P0) & CHECKPRST(*P0,*P)
              & SCORE(*P,*S) & CONTBR(*S).

CONTBR(*++++.*) <- / & WRITE('YOU GOT IT') & DELAX(CODE(*)) & ASK.
CONTBR(*S) <- WRITECH('YOUR SCORE: ') & WRITESCORE(*S)
            & WRITE('ENTER NEXT PROBE OR TYPE STOP') & READ(*X0)
            & CHECKPRST(*X0,*X) & RESPONDTO(*X).

RESPONDTO(STOP) <- / & WRITECH('I ASSUME YOU GIVE UP; THE CODE IS: ')
                   & DELAX(CODE(*C)) & CHECKCODE(*C0,*C) & WRITE(*C0) & ASK.
RESPONDTO(YES) <- / & PLAY1.
RESPONDTO(NO) <- DELAX(SEED(*))
               & WRITE('MASTERMIND WAS PLEASED TO SERVE YOU')
               & WRITE('YOU ARE NOW RETURNED TO PROLOG') & EXIT.
RESPONDTO(*P) <- SCORE(*P,*S) & CONTBR(*S).

ASK <- WRITE('DO YOU WANT ANOTHER GAME? ANSWER YES. OR ANSWER NO')
     & READ(*X) & CHECKYN(*X) & RESPONDTO(*X).

/* CODE GENERATOR */
GENCODE(*U.*V.*W.*X.NIL) <- RANDOMCOLOUR(*U) & RANDOMCOLOUR(*V)
                          & RANDOMCOLOUR(*W) & RANDOMCOLOUR(*X).

RANDOMCOLOUR(*X) <- RANDNUM(*R) & REM(*R,6,*N) & SUM(*N,1,*N1)
                  & AX(COLOUR(*), COLOUR(*X),*N1).

/* R IS PREVIOUS, S IS NEXT RANDOM NUMBER */
RANDNUM(*S) <- DELAX(SEED(*R)) & PROD(*R,125,*X) & SUM(*X,1,*Y)
             & REM(*Y,16384,*S) & ADDAX(SEED(*S)).
```

Fig. 3: First part of Interactive Manager

```
/* CHECK WHETHER ARGUMENT IS CORRECT SEED FOR RANDOM-NUMBER
   GENERATOR
*/
CHECKSEED(*X) <- INT(*X) & GE(*X,0) & LE(*X,16383) & /.
CHECKSEED(*) <- COMPLAINT.

/* CHECK FOR 'MAKE' OF 'BREAK' */
CHECKMB(MAKE) <- /.
CHECKMB(BREAK) <- /.
CHECKMB(*) <- COMPLAINT.

/* CHECK FOR 'YES' OR 'NO' */
CHECKYN(YES) <- /.
CHECKYN(NO)  <- /.
CHECKYN(*) <- COMPLAINT.

/* CHECK FOR CODE OR PROBE AND CONVERSION TO OR FROM INTERNAL
   FORMAT, WHICH CONTAINS 'NIL'
*/
CHECKPRST(STOP,STOP) <- /.
CHECKPRST(*X,*Y) <- CHECKCODE(*X,*Y).

CHECKCODE(*U.*V.*W.*X,*U.*V.*W.*X.NIL)
       <- COLOUR(*U) & COLOUR(*V) & COLOUR(*W) & COLOUR(*X) & /.
CHECKCODE(*,*) <- COMPLAINT.

COMPLAINT <- WRITE('ERROR; TRY AGAIN') & FAIL.

WRITESCORE(*BLACKS.*WHITES) <- IS (*BLACKS,*X) & WRITECH(*X)
                               & WRITECH(' BLACK AND ') & IS(*WHITES,*Y)
                               & WRITECH(*Y) & WRITE(' WHITE').


/* CONVERSION FROM SUCCESSOR NOTATION TO DECIMAL NOTATION */
IS(0,0).
IS(*S+,*N1) <- IS(*S,*N) & SUM(*N,1,*N1).
```

Fig. 4: Second part of Interactive Manager

8. Related work

Sickel has investigated [INV] how to predict in general whether it is possible to compute a particular function from a relational definition with a given rule for selecting a goal.

A striking application of relational programming has been found by Colmerauer [GDM]. In his example of a compiler written in Prolog, the analyzer takes as input the source code and outputs a parse tree decorated with semantic information. The code generator takes such a tree as input and outputs object code. Both parts are written by Colmerauer as relations between strings and parse trees. The clauses defining the relation are rewrite rules in the traditional sense. For the analyzer the first argument is given; for the code generator the second argument is given. In this way it was possible to write the code generator as a set of rewrite rules, just as the analyzer was.

In [PRL] we showed that a logic program for quicksort could be inverted to a permutation generator by writing it as a relation between a possibly unsorted list and its sorted version.

9. Concluding remarks

It is widely accepted that definitional programming is more reliable and more productive in terms of human effort than imperative programming. It is also generally true that imperative programs are more productive in terms of processor time and memory space. Definitional programming has a promising future because computer processors and memories are expected to become considerably cheaper than they are at present; also, it should be kept in mind that not nearly as much effort has been spent on efficient compilation of

definitional languages as has been the case with imperative languages.

Of two approaches to definitional programming - functional and relational - the first has been explored much more intensively than the second. Lisp has been in use since about 1960 and was backed by massive and uninterrupted support from implementers and users, initially mainly at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Prolog arrived on the scene much later. Outside of Hungary, Prolog has been, at best, tolerated rather than supported. In addition to that, an entire category of applications, namely symbolic computation, has become Lisp territory; not because of an inherent superiority of functional over relational programming, but simply because Lisp was there first.

Because of the growing importance of definitional programming, it is now time to understand the relative merits of the functional and relational approaches. Far from presenting a comprehensive comparison, this paper has only attempted to contribute a small part which we expect to be relevant in such a comparison.

10. Acknowledgements

We owe a great debt of gratitude to Grant Roberts who made logic programming feasible in Waterloo. Roberts also suggested improvements to an earlier version of the Mastermind program. The suggestion of writing a program for Mastermind came from David Warren.

The Canadian National Science and Engineering Research Council has provided partial financial support.

11. References

[ABSET] E.W. Elcock, J.M. Foster, P.D.M. Gray, J.J.M. McGregor, and A.M. Murray: ABSET: A programming language based on sets. Machine Intelligence 6, B. Meltzer and D. Michie (eds.), Edinburgh University Press, 1971.

[ABSYS] J.M. Foster and E.W. Elcock: Absys 1: an incremental compiler for assertions, Machine Intelligence 4, B. Meltzer and D. Michie (eds.), 423-429, Edinburgh University Press, 1969.

[CDI] M.H. van Emden: Computation and Deductive Information Retrieval. E. Neuhold (ed.): Formal Description of Programming Concepts, North Holland, Amsterdam, 1978.

[CLP] D.H.D. Warren: Implementing Prolog-compiling predicate logic programs. DAI Research Reports 39 and 40. Dept. of Artificial Intelligence, University of Edinburgh, 1977.

[EFP] C. Wetherell: Etudes for programmers. Prentice-Hall, 1978.

[GDM] A. Colmerauer: Les grammaires de metamorphose; in L. Bolc (ed.): Natural Language Communication with Computers, Springer Lecture Notes in Computer Science, 1977.

[ICP] F.G. McCabe: Programmer's Guide to IC-Prolog. Dept. of Computation and Control, Imperial College, 1978.

[INV] S. Sickel: Invertibility of logic programs. Fourth Workshop on Automated Deduction. University of Texas at Austin, Feb. 1979.

[IOP] G.M. Roberts: An implementation of PROLOG; M.Sc. Thesis, Dept. of Computer Science, University of Waterloo, 1977.

[MOL] J.A. Robinson: A machine-oriented logic based on the resolution principle. J. ACM 12 (1965), 23-44.

[PLANNER] C. Hewitt: Planner: a language for manipulating models and proving theorems in a robot, Proc. First Int. Joint Conf. in Artificial Intelligence, pp. 295-301.

[PLPL] R.A. Kowalski: Predicate Logic as a programming language; Proc. IFIP 74, North Holland, 1974, 556-574.

[PPL] P. Szeredi: Prolog- a very high level language based on predicate logic. Proc. Second Hungarian Computer Science Conference, Budapest, July 1977.

[PRL] M.H. van Emden: Programming in resolution logic. Machine Intelligence 8 (eds. E.W. Elcock and D. Michie) by Ellis Horwood Ltd. and John Wylie, 1977.

12. Appendix 1: Some information on the built-in predicates of the Waterloo Prolog interpreter quoted from [IOP].

READ is a predicate with one or two arguments. The second argument is the optional file identifier. A term is read from the indicated file and unified with the first argument. The term must be delimited with the end of term character. If the end of the input file has been reached the predicate fails. If backtracking returns to the read then a read of the next term will be attempted. If the term read cannot be unified with the first argument or the format of the term is invalid then backtracking will cause a read of the next term to be attempted.

WRITE is a predicate with one or two arguments. The second argument is the optional file identifier. The term specified by the first argument is written on the indicated file. The term is delimited by the end of term character. The term is written using prefix, infix and suffix notation where appropriate, as indicated by the operator declarations at the time of writing.

WRITECH is a predicate with one or two arguments. The second argument is the optional file identifier. The first argument specifies a term which is formatted using the operator declarations (as for WRITE) and placed in the output buffer for the given file. If the buffer is filled then it is written to the given file (and emptied). If the buffer is partially filled then it is not written out.

There are several predicates which are included to provide the basic operations of integer arithmetic. Each of these predicates has three arguments. The first two are the input parameters and the last is the result parameter. The first two arguments must be integers. The appropriate integer function of the first arguments is unified with the third argument.

The arithmetic predicates are:

DIFF - difference (subtraction)

PROD - product

QUOT - quotient

REM - remainder

SUM - sum

The database built-in predicates provide the facility for updating the database (i.e. the set of axioms in the active work space).

The ADDAX predicate is used to add an axiom to the database. It has one or two arguments. The first argument must be a valid axiom. It may be:

(a) a unit axiom. In this case it is a skeleton or an atom.

(b) a non-unit axiom. In this case it is of the form

<head> ← <body>. <head> must be a skeleton or atom.

The axiom specified by the first argument is added to the database. If a single argument is specified then the axiom is added after all other axioms with the same predicate name and number of arguments.

The DELAX predicate is used to delete an axiom from the database. It may be called with one or two arguments. The first argument is a term representing an axiom. The first argument may be:

(a) a unit axiom. In this case it is a skeleton or an atom.

(b) a non-unit axiom. In this case it is of the form

<head> ← <body>. <head> must be a skeleton or an atom.

Thus the first argument specifies the name and number of arguments for the axiom to be deleted. If only one argument is specified then an attempt is made to unify the argument with each of the relevant axioms in the database. The axioms are selected in the order in which they appear in the database.

If no axiom is found which is unifiable with the first argument then the predicate fails. If the unification succeeds for an axiom then the axiom is deleted and the predicate succeeds. If backtracking subsequently returns to this point then the predicate will fail, thus preventing accidental deletion of further axioms.

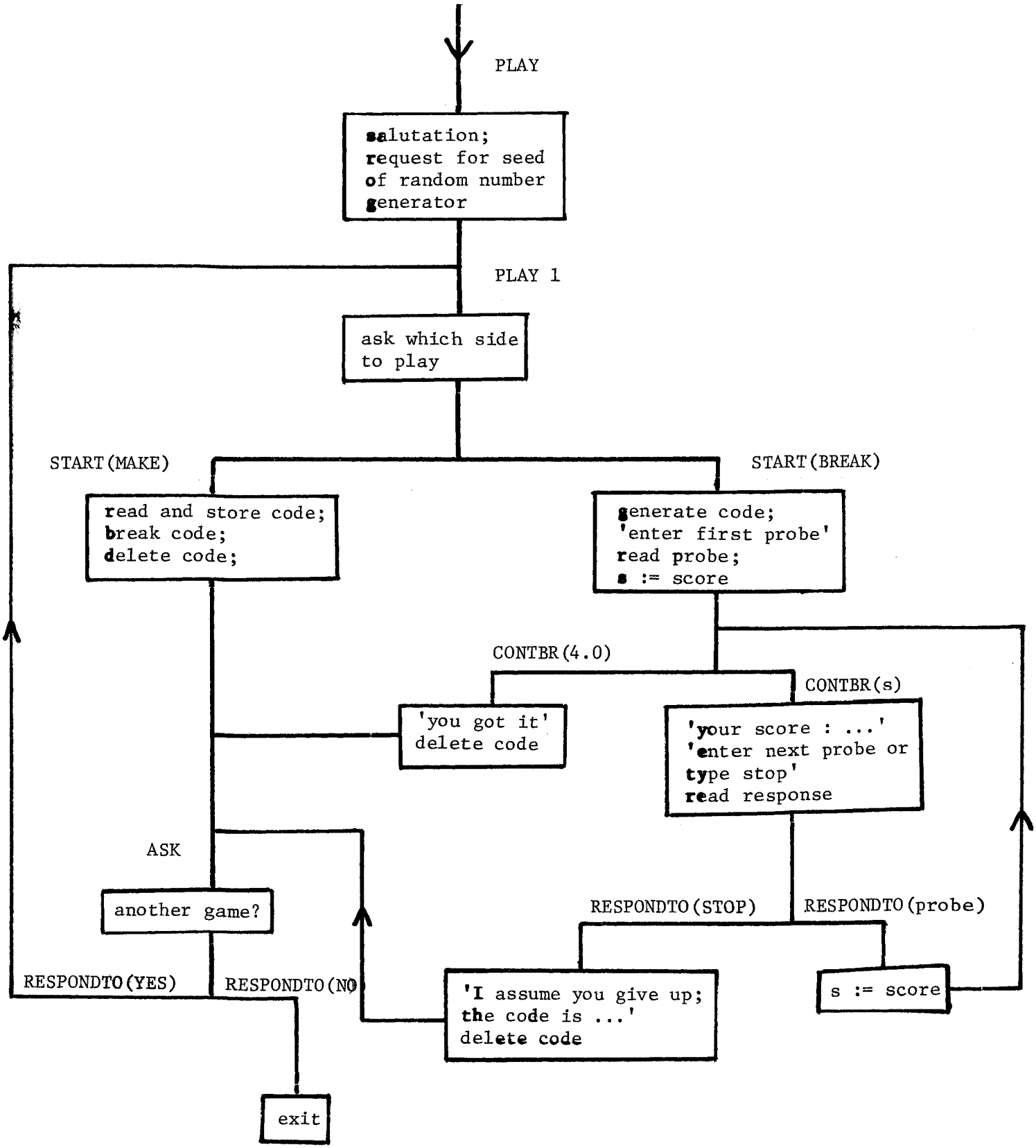
The AX predicate has two basic formats:

AX(<head>,<axiom>).

AX(<head>,<axiom>,<index>).

The AX predicate is used to retrieve axioms from the database. <head> is a model axiom head and may be a skeleton, an atom or a variable. If <head> is not a variable then it specifies a predicate name and number of arguments implicitly. The axioms for this name and number of arguments are retrieved. If an <index> is specified, then the i-th axiom that matches the <head> is unified with <axiom>, where i is the value of <index>.

13. Appendix 2: Control flow in the interactive manager



14. Appendix 3: An interactive session with the Mastermind Program.

```
WELCOME TO PROLOG 0.0
LOAD(MMIND)<-
<-play.
MASTERMIND AT YOUR SERVICE.
ENTER AN ARBITRARY NUMBER BETWEEN 0 AND 16383.
12345.
EXAMPLE FORMAT FOR ENTERING CODE: YELLOW.BLUE.WHITE.BLACK.
DO YOU WANT TO MAKE OR BREAK CODES? ANSWER MAKE. OR ANSWER BREAK.
break.
ENTER FIRST PROBE.
black.blue.green.red.
YOUR SCORE: 1 BLACK AND 0 WHITE.
ENTER NEXT PROBE OR TYPE STOP.
black.black.black.black.
YOUR SCORE: 2 BLACK AND 0 WHITE.
ENTER NEXT PROBE OR TYPE STOP.
black.black.white.white.
YOUR SCORE: 1 BLACK AND 1 WHITE.
ENTER NEXT PROBE OR TYPE STOP.
black.yellow.black.yellow.
YOU GOT IT.
DO YOU WANT ANOTHER GAME? ANSWER YES. OR ANSWER NO.
yes.
DO YOU WANT TO MAKE OR BREAK CODES? ANSWER MAKE. OR ANSWER BREAK.
make.
ENTER CODE; I PROMISE NOT TO LOOK.
red.white.bleu.yellow.
ERROR; TRY AGAIN.
so.you.have.been.looking.
ERROR; TRY AGAIN.
red.white.blue.yellow.
MY FIRST PROBE IS: WHITE.YELLOW.BLACK.RED; SCORE: 0 BLACK AND 3 WHITE.
MY NEXT PROBE IS: BLACK.BLACK.RED.WHITE; SCORE: 0 BLACK AND 2 WHITE.
3 TRIES TO GO.
MY NEXT PROBE IS: BLUE.RED.WHITE.YELLOW; SCORE: 1 BLACK AND 3 WHITE.
2 TRIES TO GO.
MY NEXT PROBE IS: RED.WHITE.BLUE.YELLOW; SCORE: 4 BLACK AND 0 WHITE.
1 TRIES TO GO.
THE CODE MUST BE: RED.WHITE.BLUE.YELLOW.
DO YOU WANT ANOTHER GAME? ANSWER YES. OR ANSWER NO.
no.
MASTERMIND WAS PLEASED TO SERVE YOU.
YOU ARE NOW RETURNED TO PROLOG.
PLAY<-
<-stop.
```