

Original citation:

Beynon, Meurig (1983) A definition of the ARCA notation. University of Warwick.
Department of Computer Science. (Department of Computer Science Research Report).
(Unpublished) CS-RR-054

Permanent WRAP url:

<http://wrap.warwick.ac.uk/60757>

Copyright and reuse:

The Warwick Research Archive Portal (WRAP) makes this work by researchers of the University of Warwick available open access under the following conditions. Copyright © and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable the material made available in WRAP has been checked for eligibility before being made available.

Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

A note on versions:

The version presented in WRAP is the published version or, version of record, and may be cited as it appears here. For more information, please contact the WRAP Team at: publications@warwick.ac.uk



<http://wrap.warwick.ac.uk/>

The University of Warwick

THEORY OF COMPUTATION

REPORT NO. 54

A DEFINITION OF THE ARCA NOTATION

BY

W. M. BEYNON

183

Department of Computer Science
University of Warwick
Coventry, CV4 7DT
England

September 1983

A definition of the ARCA notation.

W.M. Beynon.

Department of Computer Science, University of Warwick.

ABSTRACT

ARCA is a programming notation intended for interactive specification and manipulation of combinatorial graphs. The main body of this report is a technical description of ARCA sufficiently detailed to allow an interpreter to be developed. Some simple illustrative programs are included.

ARCA incorporates variables for denoting primitive data elements (essentially vertices, edges and scalars), and diagrams (essentially embedded graphs). A novel feature is the use of two kinds of variable: the one storing values (as in conventional procedural languages), the other functional definitions (as in non-procedural languages). By means of such variables, algebraic expressions over the algebra of primitive data elements may represent either explicit values or formulae. The potential applications and limitations of ARCA, and more general "algebraic notations" defined using similar principles, are briefly discussed.

Contents.

Introduction.

§1. Generalities.

- 1.1. Basic concepts.
- 1.2. Program structure and scope rules.

§2. The primitive data algebra.

- 2.1. Primitive data-types.
- 2.2. Primitive operators.
 - a) Arithmetic operators.
 - b) Vector operators.
 - c) Permutation operators.
 - d) Geometric operators.
 - e) Construction/projection operators.
 - f) Conversion operators.
 - g) Amalgamation operators.
 - h) Indexing operators.
 - i) The weight operator.

§3. Expressions of primitive type.

- 3.1. Syntax for expressions of primitive type.
- 3.2. Precedence rules.
- 3.3. Modulus coercion rules.
- 3.4. Scaling and coordinate information.

§4. Primitive actions.

- 4.1. Primitive variables.
- 4.2. Actual and abstract expressions and formulae.
- 4.3. Assignment to primitive variables.
- 4.4. Values of abstract variables and formulae.

§5. Diagrams: variables and assignment.

- 5.1. Diagram variables.
 - 5.1.1. Explicit diagrams.
 - 5.1.2. Vertices and colours in explicit diagrams.
 - 5.1.3. Subdiagrams of explicit diagrams.
 - 5.1.4. Implicit diagrams.
- 5.2. Diagram assignment.

§6. Diagrams and algebraic expressions.

- 6.1. Expressions derived from a diagram.

6.2. Diagram expressions.

6.3. Diagrams as parameters and results of user-defined operators.

§7. Composite actions.

7.1. Boolean relations.

7.2. Integer lists.

7.3. General syntax for actions.

§8. User-defined operators.

8.1. Syntax for user-defined operators.

8.2. Semantics of operator definition.

§9. Programming in ARCA.

9.1. Examples of ARCA programs.

9.2. Miscellaneous remarks on ARCA programming.

§10. A parser for ARCA expressions.

Epilogue.

Acknowledgements.

References.

Introduction.

The ARCA* notation is intended for the description and interactive manipulation of combinatorial diagrams such as graphs or lattices. The main body of this report comprises a definition of the notation hopefully precise enough to allow implementation. Some simple ARCA programs for describing Cayley diagrams are included for illustration. The primary purpose of undertaking this design exercise is to indicate how a general class of "algebraic notations" might be modified for practical use. The design suggests a number of theoretical and practical issues for further research, which are briefly discussed in the final section of the report. This is the principal justification (or perhaps excuse) for presenting the technical description of ARCA as a Theory of Computation report.

§1. Generalities.

The purpose of an ARCA program is to describe appropriate internal representations of combinatorial diagrams - specifically, realisations of graphs in Euclidean space - for the purpose of display and manipulation. The ideal environment for the development of such a program would provide both an interactive medium for editing and interpreting ARCA programs (as is available for POP2 or APL), and a means of displaying diagrams at each stage of the descriptive process.

It should be emphasised that the abstract description of a diagram which an ARCA program provides is not merely a means to the end of display. In effect, it describes a simple conceptual model of the diagram which may not be apparent in a display, but can aid the user in its subsequent manipulation or application. As a trivial illustration of this, consider the diagram consisting of a unit square with vertices OABC. Under one conceptual model, C might be constrained so that OC is the vector sum of OA and OB. In this case, the effect of moving the point A to the present position of C would be to transform the original square into a parallelogram. Under another model, C and B might be constrained to be the vertices of a square OACB in a particular orientation. In this case, the same movement of A would result in rotation and magnification of the original square.

This report focuses on the abstract specification and interpretation of ARCA programs, and deals only superficially with the problems of implementation and display.

1.1. Basic concepts.

The ARCA notation is intended for describing geometric realisations of graphs in Euclidean space. It includes means of specifying

- (1) incidence information for vertices and (possibly directed and/or coloured) edges.
- (2) geometric information about the coordinates of vertices (if necessary in dimension higher than 2).

To a limited degree, it also allows the abstract specification of relationships between coordinates for vertices, and incidence and colouring information for edges.

* ARCA is named after Arthur Cayley (1821-1895). It may be charitably regarded as "An Aid for the Realisation of Combinatorial Artefacts". Experience of ARCA programming suggests novel interpretations of old and neo-English epithets such as arcane, Arcadian, arcaic, arcangelic, and arcatypal.

As will be explained in more detail later, there are three primitive data types in ARCA: integer, for representing scalar information; vector, for representing coordinate information; colour, for representing incidence information. A variable of primitive type also has a 'kind': actual or abstract. Actual variables are similar to variables in conventional programming languages; they are used to store explicit values which may be re-assigned in the traditional manner. Abstract variables are used to store 'implicit' values; that is, values which are specified indirectly by means of formulae. Such variables serve to represent relationships between values. (As a trivial example, the coordinates of the vertex C of the parallelogram OACB referred to above might be represented by an abstract variable to which a simple formula such as "coordinates of A + coordinates of B" had been assigned.) A formula is attached to a variable by means of an "abstract assignment"; just as specific values may be altered by re-assigning to actual variables, so relationships between values can be altered if desired by re-assigning to abstract variables.

There is one complex data type in ARCA: the diagram. A variable of type diagram is used to represent a geometric realisation of an edge-coloured digraph in Euclidean space. Diagram variables can be of several different kinds: actual or abstract and explicit or implicit. This reflects the fact that a realisation of a graph may be specified at several levels of abstraction; for instance: its size, coordinate and incidence information may be known explicitly; it may be known as an abstract graph, but have coordinates implicitly specified; it may be specified as a complete graph of unknown size.

Like primitive assignments, assignments to diagram variables are of two types: 'explicit' assignments in which appropriate values are assigned (implicitly or explicitly) to the individual vertices and edges of an explicit diagram variable, and 'implicit' assignments, in which a formula which implicitly represents a diagram is assigned to an implicit diagram variable. (See §8 for more details, and illustrative examples.)

In all assignments, the RHS is an 'algebraic expression' built up out of constants, actual and abstract variables by means of operators. There are numerous standard operators, and these can be augmented if necessary by the introduction of user-defined operators. In an actual assignment to a primitive variable, the RHS must be an algebraic expression in which no abstract variable appears, since it is to be interpreted as a value rather than a formula. In an abstract assignment, the RHS will generally be an algebraic expression in which abstract variables are present; such an expression is to be interpreted as a formula in so far as the value it represents (if defined) is dependent upon the current values of these abstract variables.

1.2. Program structure and scope rules.

An ARCA program is a sequence of declarations, actions and definitions. In general terms: *declarations* are used to specify the nature of variables represented by particular identifiers, *actions* serve either to assign explicit values to variables (e.g. to give explicit coordinates to a vertex, or specify a set of edges in a graph) or to establish relationships between variables (e.g. to ensure that four vertices to lie at the corners of a square, or that the orientation of edges on a directed cycle is determined by the parity of an integer variable); *definitions* permit the user to specify more complex relationships between variables by introducing additional operators.

The precise syntax and semantics for declarations, actions and definitions will be described later. Informally, the syntax of declarations and actions in ARCA resembles that of declarations and statements in traditional ALGOL-like

languages, but declaration and formula assignment to abstract variables enriches the semantics. Definitions comprise a heading for parameter specification, followed by the "definition body"; though they superficially resemble ALGOL function declarations syntactically, the definition of an operator is in no sense a general ARCA subroutine, and is subject to rigorous restrictions. For instance, only actual declarations of primitive variables may appear in the body of a definition, whilst the actions are constrained to simple forms, and must prescribe how to compute a result from the set of parameters without side-effects. The parameter passing conventions are unusual, and will be explained later.

In ARCA, declarations and actions are either local to a definition body, or global, and declarations may not be embedded in actions. The scope of the declaration of a variable is the set of statements in which use of the variable is valid, and is defined by two "scope rules":

- (1) A global statement is in the scope of any global declaration which precedes it.
- (2) A statement which is local to a definition body is in the scope of any declaration which precedes it within the same body, and any global declaration of an actual primitive variable which precedes it.

The order of declarations, actions and definitions in an ARCA program is constrained only by the above scope rules, and the additional requirement that the (complete) definition of an operator must precede its use in any action.

In the interpretation of an ARCA program, global statements are interpreted as they are encountered, whilst statements local to the body of an operator definition are compiled for interpretation on each subsequent evaluation of the operator.

§2. The primitive data algebra.

ARCA is largely based upon a small set of primitive data-types and operators, which (with some poetic licence) may be conveniently described as the "primitive data algebra". This algebra is specified in detail below. At times, the emphasis on interpreting notation by means of operators may appear pedantic (see e.g. vector and colour constructors), but the reasons for this will become apparent later.

2.1. Primitive data-types.

There are 3 primitive data-types in ARCA:

integer (int) , vertex (vert) , colour (col).

Type 'int' is used to represent a scalar quantity, such as an index, an angle of rotation, or a coordinate of a point. (All scalar information is treated as discrete, as will be explained in more detail later.) Each 'int' has an associated *modulus* which is a non-negative integer. An integer of modulus m will be called an 'm-int'. For $m > 1$, an m-int represents a residue modulo m . By default (that is, unless a positive modulus is explicitly specified), integer constants in ARCA are integers in the traditional sense; they are regarded as 0-ints. The class of 1-ints is used syntactically for "scaled integers"; integers which denote a number of geometrical units of length, but which are operationally interpreted as 0-ints by scaling.

Type 'vert' is used to represent a vector quantity, such as a sequence of indices, or the coordinates of a point. In general, coordinates of points which are explicitly specified will be vectors of 1-ints. Formally, a vertex represents a m-int valued function on $\{1,2,\dots,n\}$. It is then said to have *modulus* m , and

dimension n.

Type 'col' is used to represent 1-1 partial functions ("partial perms") from the set $\{1,2,\dots,n\}$ to itself. The integer n is then the *degree* of the colour. The main purpose of colours is to represent edge information within diagrams (see §5), but they may also be used for representing permutations.

2.2. Primitive operators.

There are a number of standard operators on primitive data-types; these are classified and described below.

a) Arithmetic operators.

The infix binary operators '+', '-', '*', '%' respectively denote addition, subtraction, multiplication and mod-reduction. The postfix unary operator '⁻' is used to denote 'inverse within residue class' (where defined).

Semantic rules.

The arguments for '+', '-' and '*' must have the same modulus, and those of '%' must be of modulus 0. The inverse of an m -int n is defined if and only if m and n are co-prime.

b) Vector operators.

The infix binary operator '+' : $\text{vert} \times \text{vert} \rightarrow \text{vert}$ denotes vector addition.

Scalar product of vectors is defined by $\langle , \rangle : \text{vert} \times \text{vert} \rightarrow \text{int}$, and scalar multiplication by the infix binary operator '.' : $\text{int} \times \text{vert} \rightarrow \text{vert}$.

Semantic rules.

The arguments of '+' must have the same dimension and modulus. The arguments of '.' must have the same modulus. The arguments of a scalar product or scalar multiplication must be of the same modulus, which will also be that of the result.

c) Permutation operators.

The infix binary operators '.' : $\text{col} \times \text{col} \rightarrow \text{col}$ and '@' : $\text{col} \times \text{int} \rightarrow \text{col}$ respectively denote composition (product) and exponentiation of perms. The postfix unary operator '⁻' : $\text{col} \rightarrow \text{col}$ denotes the inverse of a partial perm.

Semantic rules.

The arguments of '.' must have the same degree. The int argument to '@' must be a 0-int. If p is a partial perm then " $p@0$ " will denote the perm obtained by restricting the identity perm to those indices on which p is defined.

d) Geometric operators.

Scalar multiplication by a rational number defines a "scaling operator":

$$\text{int} \times \text{int} \times \text{vert} \rightarrow \text{vert}.$$

If v is a vector, and m and n are integers, then $(m/n).v$ denotes the image of (m,n,v) under this map.

The ternary operators

$$\text{rot}(, ,) : \text{vert} \times \text{int} \times \text{vert} \rightarrow \text{vert} \text{ and } \text{ref}(, ,) : \text{vert} \times \text{vert} \times \text{vert} \rightarrow \text{vert}$$

are used to specify rotation and reflection in the plane.

Semantic rules.

The scaling operator is defined for parameters (m,n,v) such that $n>0$ and v has modulus 1 or 0. Note that if $m<0$ scaling entails reflection of the vector v in the origin.

The arguments for $rot()$ are to be interpreted as
(point to be rotated , angle of rotation , centre of rotation).

The vert arguments must be of dimension 2, and of modulus 1 or 0. Anticlockwise rotation through $2m\pi/n$ is specified by int argument ' $m\%n$ '.

The arguments for $ref()$ are to be interpreted as
(point to be reflected , < pair of points on axis of reflection >).

All the arguments in this case are of dimension 2 and modulus 1 or 0, and the pair of points on the axis of reflection must be distinct.

Since vectors in ARCA have integer components, the use of geometric operators will frequently lead to approximation. As explained more fully in §3.4, the integer arithmetic with coordinates should be carried out on an internal scale large enough to ensure that approximation errors introduced through the reasonable use of geometric operators are not significant when points are displayed.

e) Construction/projection operators.

Vectors can be constructed by specifying their component lists explicitly. Formally, a family of 'constructors' is introduced for this purpose. A vertex constructor is a map

$$m\text{-int}^n \rightarrow n\text{-vert.}$$

The image of (I_1, \dots, I_n) is denoted by $[I_1, \dots, I_n]$.

There is also an "interval operator"

$$'\cdot' : 0\text{-int} \times 0\text{-int} \rightarrow \text{vert.}$$

If m and n are 0-ints, then $m..n$ is defined if and only if $m \leq n$, when it denotes the vector $[m, m+1, \dots, n]$ of modulus 0 and dimension $m-n+1$.

The projection operator $\text{vert} \times \text{int} \rightarrow \text{int}$ mapping (V,I) to $V[I]$, where the modulus of I is the dimension of V , is used to specify a particular component of a vertex.

Constructors for colours, though syntactically similar to vertex constructors, are semantically different, and are based upon a form of cycle notation for partial perms. The 'colour constructor'

$$\{ , \dots , \} : m\text{-int}^n \rightarrow \text{col}$$

is used to specify a partial perm which is a "complete" cycle. Formally

$$\{I_1, \dots, I_n\}$$

denotes the partial perm p of degree n which is defined by

$$p(I_i) = I_{i+1} \text{ for } 1 \leq i \leq n$$

(where addition of suffices is mod n), and is otherwise undefined. A second type of constructor, defined by a map

$$\{ , \dots , ? \} : m\text{-int}^n \rightarrow \text{col}$$

is used to specify an "incomplete" cycle. Formally

$$\{I_1, \dots, I_n, ?\}$$

denotes the partial perm p of degree m which is defined by

$$p(I_i) = I_{i+1} \text{ for } 1 \leq i \leq n-1$$

and is otherwise undefined.

Note that the need for an adequate notation for colours which are partially defined precludes the use of conventional cycle notation. Indeed, to obtain a convenient notation for most partial perms it is necessary to use constructors in

conjunction with the 'vert-to-col conversion' and 'superposition' operators to be introduced below.

The projection operator for colours is the map
 $\text{col} \times \text{int} \rightarrow \text{int}$

which maps (C, I) to $C\{I\}$. This projection is meaningful provided that I has modulus $m = \text{degree of } C$; the result $C\{I\}$ then also has modulus m . The value of $C\{I\}$ is the result of evaluating the partial perm C at I , and may be undefined if C is undefined at I .

f) Conversion operators.

There are two operators for "integer conversion". They are the postfix unary operators $': 1\text{-int} \rightarrow 0\text{-int}$ and $': m\text{-int} \rightarrow 0\text{-int}$.

If I is a 1-int, then I' is a scaled integer, which represents a number of geometric units; I' is derived by multiplying I by the appropriate scale factor.

If I is an m -int, then I' denotes the 'principal value' of I : the unique integer congruent modulo m in the range $1 \leq I' \leq m$.

Note that there are syntactic conventions which enable implicit conversion of 0-ints to m -ints (see below), but all other conversions must be explicit.

There is also a "vert-to-col conversion" operator
 $\{ \} : \text{vert} \rightarrow \text{col}$

which maps the vector $[a_1, a_2, \dots, a_m]$ to the colour C of degree m defined by $C\{I\} = a_i$ for $1 \leq i \leq m$. For this purpose, the a 's must be m -ints which define a permutation of the set of residues $1, 2, \dots, m \bmod m$; thus the colour $\{[a_1, a_2, \dots, a_m]\}$ is defined only if it represents a conventional permutation C for which $C\{I\}$ is defined for $1 \leq i \leq m$.

g) Amalgamation operators.

There are two infix binary operators

$**$ (smash) : $\text{vert} \times \text{vert} \rightarrow \text{vert}$ and $::$ (join) : $\text{col} \times \text{col} \rightarrow \text{col}$

which are intended to be used in the context of diagram products and joins respectively.

If V and W are verts of dimension m and n respectively, and having the same modulus r , then $V**W$ is the $(m+n)$ -vert of modulus r whose first m components are the components of V and whose last n components are the components of W .

If A and B are cols of degree m and n respectively, then $A::B$ is the $(m+n)$ -col defined component-wise by the formula

$$A::B \{i\} = \begin{cases} A\{i\}' \% (m+n) & \text{for } 1 \leq i \leq m \\ m+B\{i-m\}' \% (m+n) & \text{for } m < i \leq m+n \end{cases}$$

with the convention that $A::B\{i\}$ is undefined if the corresponding RHS is undefined.

If f and g are colours of degree m , then $f\$g$ ("superposition of f and g ") denotes a colour of degree m which is defined by

$$f\$g\{I\} = \text{if } g\{I\} \text{ is defined then } g\{I\} \text{ else } f\{I\}.$$

h) Indexing operators.

There are two infix binary operators

$**$: $\text{int} \times \text{int} \rightarrow \text{int}$

$\% \%$: $\text{int} \times \text{vert} \rightarrow \text{int}$

which are intended for specifying indices in diagram products and joins

respectively.

If I and J are integers of modulus m and n respectively, then I**J denotes the mn-int whose principal value is $m(I-1)+J$ modulo mn.

Let $V=[I_1, \dots, I_k]$ be a vertex whose components are positive 0-ints, and let S_i denote the sum

$$\sum_{j=1}^k I_j.$$

If I is an integer of modulus S_k , then $I\%V$ is defined as an I_j -int for some j. The value of j is the unique integer such that $S_{j-1} < I \leq S_j$ and $I\%V$ then has the principal value $I-S_{j-1}$.

i) The weight operator.

There are unary operators

$\text{int} \rightarrow 0\text{-int}$, $\text{vert} \rightarrow 0\text{-int}$ and $\text{col} \rightarrow 0\text{-int}$

which respectively return the modulus of an int, the dimension of a vert, and the degree of a col. The postfix unary operator '#' is used to denote each of these operators.

If X is an object of primitive type, $X\#$ will be called the 'weight' of X.

§3. Expressions of primitive type.

All the operators above can be used to form algebraic expressions. The primary operands in such expressions are constants and/or variables of primitive type, including possibly constituent variables of diagrams (see §6).

As explained in §8 below, the set of operators can also be augmented by means of user-defined operators.

The syntax of expressions of primitive type is summarised below. A form of BNF grammar is used for syntactic specifications, and the following conventions are adopted throughout:

- terminal tokens (e.g. operators, identifiers, reserved words) are roman
- grammar variables are distinguished from terminal tokens by italic
- meta-symbols (| , [,] , { , } and ::=) are bold.

The symbol "**|**" is used for alternatives in grammar rules, square brackets (**[]**) to indicate an optional term, and braces (**{ }**) to indicate one or more instances of a term.

3.1. Syntax for expressions of primitive type.

Integer expressions.

```
I_exp ::= ( I_exp )
        | | I_exp |
        | NUMBER | I_ID | I_OP ( exp_list )
        | I_exp # | V_exp # | C_exp # | D_ID #
        | I_exp + I_exp | I_exp - I_exp
        | I_exp * I_exp | I_exp ^
        | I_exp % I_exp | I_exp '
        | I_exp ** I_exp | I_exp %% V_exp
        | < V_exp , V_exp >
        | V_exp [ I_exp ] | C_exp { I_exp }
        | I_exp "
```

Vertex expressions.

```
V_exp ::= ( V_exp )
        | | V_exp |
        | [ I_exp_list ] | V_ID | V_OP ( exp_list )
        | D_ID ! I_exp
        | V_exp + V_exp | I_exp . V_exp
        | I_exp .. I_exp
        | ROT ( V_exp , I_exp , V_exp )
        | REF ( V_exp , V_exp , V_exp )
        | ( I_exp / I_exp ) . V_exp
        | V_exp ** V_exp
```

Colour expressions.

$$\begin{aligned}
 C_exp & ::= (C_exp) \\
 & \mid | C_exp | \\
 & \mid \{ I_exp_list \} \mid C_ID \mid C_OP (exp_list) \\
 & \mid LETTER _ D_ID \\
 & \mid col_seq LETTER _ D_ID [_] [@ I_exp] \\
 & \mid C_exp . C_exp \mid C_exp _ \mid C_exp @ I_exp \\
 & \mid C_exp \$ C_exp \\
 & \mid C_exp :: C_exp \\
 & \mid \{ V_exp \} \\
 col_seq & ::= \{ LETTER [_] [@ I_exp] . \} \\
 I_exp_list & ::= I_exp \{ , I_exp \} \\
 exp_list & ::= exp \{ , exp \} \\
 exp & ::= I_exp \mid V_exp \mid C_exp \mid D_exp
 \end{aligned}$$

3.2. Precedence rules.

As the syntax above shows, ARCA expressions can be very complex, and a large number of precedence rules are needed for disambiguation. The syntactic rules given in this report have been used as the basis of a YACC-generated LR-parser for ARCA expressions and relations; the interested reader is referred to the source for this parser (see §10) for a suitable disambiguating set of precedences.

3.3. Modulus coercion rules.

Integer variables and constants of modulus 1 are intended to simplify the specification of coordinates; those of modulus > 1 for indexing of diagram vertices and components of colours. In both these contexts, it can be inconvenient to have to specify the moduli of integer constants explicitly, and modulus coercion rules have been devised to alleviate this problem.

As mentioned above, coercion of a 0-int to an m-int may occur within an expression, but no other modulus coercions are permitted. Such coercions may extend to the coercion of a vector of modulus 0 to a vector of positive modulus. There are three general contexts in which coercion of a 0-int occurs:

- [1] Suppose that a binary operator requires operands of the same modulus, and one operand is of positive modulus. The other operand, if of modulus 0, will be coerced. (The operators to which this rule applies are: integer addition, subtraction and multiplication; addition and scalar multiplication of vertices; multiplication and juxtaposition of colours.) In the same spirit, in an *I_exp_list*, all entries with positive modulus must be of the same modulus, and any entries of modulus 0 will be coerced so that *all* entries have the same modulus.

- [2] An integer I which is used to index a component of a colour C (as in $C\{I\}$), or to select a vertex from a diagram D (as in $D\{I\}$), is coerced to the appropriate modulus viz. the degree of the colour C , and the size of the diagram D , respectively.
- [3] In any assignment to an integer or vertex variable of positive modulus, the expression on the RHS, if of modulus 0, will be coerced to the appropriate modulus. (In a similar fashion, it is permissible for an abstract integer variable of positive specified modulus to evaluate to a 0-int.)

3.4. Scaling and coordinate information.

As mentioned above, scalar information in ARCA is treated as discrete, even though in principle geometric operators should introduce points with non-integral, even irrational, coordinates. It is intended that coordinate arithmetic is in fact done (where necessary with approximation) on an integer internal grid far larger than the grid any physical device used for display would permit. For purposes of display, physical coordinates would then be calculated by means of an appropriate mapping of the internal grid onto the display grid. On the internal grid, the point produced (for instance) by rotating a gridpoint V through an angle of π/n , n times over, might very well differ from V , but it is intended that this discrepancy should be small enough to lie beyond the resolution of the display. The precise choice of internal and external scales, and the mapping between them, is a pragmatic issue of implementation. It would be unreasonable to expect such a display system to be proof against the effect of cumulative approximation errors, but in practice a simple mechanism of the above kind should be sufficient if used with discretion.

In an ARCA program, vectors of modulus 0 which are used to specify coordinates are to be interpreted on the *internal* scale. This means that a vector of modulus 0 such as $[1,1]$ will represent a point indistinguishable from $(0,0)$ on display. Thus, the effective use of literal vectors of modulus 0 for coordinate specification will require reference to the internal coordinates of other points, or special knowledge of the relationship between internal and external coordinate scales. As an alternative, vectors of modulus 1 may be used to specify coordinates on some pre-set geometric scale. Thus a vector of modulus 1 such as $[1,1]$ will be represented on the internal grid by the point $[\mu,\mu]$, where μ is a (large) pre-set integer of modulus 0, and displayed at the point whose displacement from the x and y axes is a standard length such as a centimetre or an inch. The conversion operator ' ' then maps the 1-int m to the 0-int $m\mu$.

§.4. Primitive actions.

Within each primitive data-type, there are "actual" variables which are used to represent and manipulate values in the conventional manner, and "abstract" variables which represent abstractly defined values i.e. algebraic expressions which evaluate to the appropriate type. A primitive action in ARCA is then the assignment of a value to an actual variable or an expression to an abstract variable.

4.1. Primitive variables.

In ARCA, each variable has a *type* viz.

'int', 'vert', 'col' or 'diag'.

Primitive variables (those of type int, vert or col) also have a *kind*: 'actual' or 'abstract'. There is an important distinction between the two kinds of primitive variable, and (for the purposes of this report) a lexical convention is adopted

whereby identifiers of the form AX, BX, CX, ... are used for actual variables, and identifiers of the form ax, bx, cx, ... for abstract variables. (A similar convention is useful in ARCA programs - c.f. §9.)

Declaration of primitive variables.

```
declaration ::= [ abst ] typename [ I_exp ] : id_list
typename   ::= int | vert | col
id_list    ::= ID { . ID }
```

Examples of declarations.

The simple examples below are supplemented by the example programs in §9. For clarity, related declarations have been placed on a single line and separated by semi-colons.

```
vert 2 : OX
int 0 : AX ; int 2*AX : MX, NX
abst int 2 : px
abst col : ax, bx, cx
abst int : rx ; abst vert rx : vx
```

Semantics of declaration of primitive variables.

The keyword "abst" is used when declaring abstract variables; variables are otherwise actual. In a declaration, the *typename* specifies the type of a variable, and where necessary the integer expression *I_exp* specifies its weight.

The weight of an actual variable must be specified on declaration by means of an integer expression which is 'actual' in a sense to be explained below, and defines an appropriate non-negative value. On declaration, actual variables are assigned default values: int's are initialised to 0, vert's to the zero vector, and col's to the totally undefined partial perm.

It is not necessary to specify the weight of an abstract variable on declaration. If a weight is specified, it may be an arbitrary integer expression (not necessarily actual). It then serves a purely 'defensive' function; on any subsequent evaluation of the variable, its weight must be consistent with its declared weight.

Abstract variables have no default value on declaration.

4.2. Actual and abstract expressions and formulae.

Actual variables are used to store explicit values; abstract variables values which are "abstractly defined", that is, for which a formula rather than an explicit value is supplied. For this purpose, a *formula* is defined to be an algebraic expression in which each operand is either a constant or an abstract variable, and each operator is either primitive, or user-defined (see §8). The full syntax of expressions of primitive type is given in §3.

A formula is *constant* if it contains no abstract variables. If *f* is a formula, the *value* of *f* (to be discussed formally below) is denoted by $|f|$.

There is no special notation for formulae. A general algebraic expression will be interpreted as a formula by replacing each actual variable by its value, and performing any evaluation of subformulae explicitly stipulated by means of " $|\dots|$ ". Such an algebraic expression is *actual* if it corresponds to a constant

formula, and is otherwise *abstract*. Actual and abstract expressions can be distinguished statically; that is, without the need for explicit evaluation. In this connection, note that an abstract expression, such as 'ax-ax', may have a constant value.

4.3. Assignment to primitive variables.

Syntax for assignment to primitive variables.

$$\begin{aligned} \text{assignment} & ::= I_l_name := I_exp \\ & \quad | V_ID := V_exp \\ & \quad | C_ID := C_exp \\ I_l_name & ::= V_ID [I_exp] \\ & \quad | C_ID \{ I_exp \} \\ & \quad | I_ID \end{aligned}$$

Note: In the syntax rules above, the identifiers and expressions are typed, so that e.g. V_ID is an identifier of type vert. Similarly, the meta-variable I_exp denotes a primitive expression which defines an integer value. Type checking of identifiers and expressions is conveniently handled syntactically by the ARCA expression parser in §10, but might alternatively be managed by semantic rules.

Examples of assignments.

The assignments below are to be interpreted in conjunction with the examples of declarations given above, and similar conventions have been used.

```
Ox := [0,1%1]
Ax := 3; Mx := 5; Nx := 2*Mx%6
ax := bx @ (1-2*px'); px := rx%2
rx := 1; vx := [px'%1, 1-px']
```

Semantics for assignment to actual variables.

Actual variables are similar to variables in conventional imperative languages; they are used to store a value of the appropriate type, and are overwritten on each assignment. In any assignment to an actual variable name, the RHS must be an actual expression, and define a value of the same weight as the LHS.

Actual vert's and col's can be also assigned "component-wise". In this case, the I_l_name must be defined by an actual vert or col name indexed by an actual I_exp of an appropriate weight. Component-wise assignment to a vert or col is subject to the same semantic rules as assignment to an actual integer variable of the appropriate modulus.

The value of an actual variable (or component of an actual variable) is determined by the last value assigned.

Semantics for abstract assignment.

As explained above, abstract variables are used to represent implicitly defined values. This is achieved by assigning formulae to abstract variables; the value of an abstract variable is then obtained by evaluating the formula most recently assigned.

For abstract assignments, the only valid *l*-names are abstract variable names, so that component-wise assignment to abstract vertices and colours is impossible. In an abstract assignment, the RHS is interpreted as a formula, which may be constant.

4.4. Values of abstract variables and formulae.

Abstract variables have no default values. Even the assignment of a formula to an abstract variable does not guarantee that its value exists. For instance, it is quite legitimate to introduce a formula which contains uninitialised abstract variables on the RHS of an assignment. It is only necessary to ensure that an abstract variable has a value (if appropriate, of a specified weight) when required; that is, when its evaluation is stipulated in the context of an expression.

Suppose that the abstract variable *vx* is "currently defined by formula *f*" (ie. *f* was the formula most recently assigned to *vx*) and that *S* is the set of abstract variables in *f*. The (current) value of *vx* (if it exists) is then the value of the formula *f*, which is recursively defined as "the result of replacing each abstract variable in *f* by its value, and evaluating *f*". Whether or not this value exists, it is useful to consider "the set of abstract variables on which the current value of *vx* notionally depends", defined as

$$\text{dep}(vx) = \text{if } S \text{ is } \varnothing \text{ then } \varnothing \text{ else } S \cup \{ \text{dep}(wx) \mid wx \in S \}.$$

Meaningful use of abstract variables may then be ensured by monitoring the sets *dep()* and checking that

- (i) at no time does a variable *vx* belong to *dep(vx)*
- (ii) at no time is a variable *vx* such that *dep(vx)* contains an undefined variable evaluated.

To guarantee condition (i), it is enough to avoid assignments to an abstract variable *wx* which lead to *wx* becoming dependent upon itself.

Coercion rules.

In principle, it is always possible to convert an abstract expression *e* to the actual expression $|e|$ by evaluation. In practice, it may be helpful to allow conversion of an abstract expression to an actual expression by means of such an evaluation whenever a value rather than an abstract formula is expected, though warning of any such coercion would be desirable.

§5. Diagrams: variables and assignment.

The non-primitive data-type diagram (*diag*) is used to represent a realisation of an edge-coloured digraph. A *diag* comprises an array of *verts* and a list of *cols* whose degree is the size of the *vert* array. The *size* of a diagram is the size of its vertex array.

If the digraph *G* is represented by the *diag* *D*, the array of *verts* of *D* supplies an index and coordinates for each node of *G*. Each *col* *c* of *D* then denotes a partial perm of the node indices; if *c* maps index *i* to index *j*, then there is a directed edge "of colour *c*" from the node with index *i* to the node with index *j*. Note that only digraphs *G* in which at most one edge of a particular orientation and colour is incident at any vertex are representable by a *diag*.

5.1. Diagram variables.

Like primitive variables, variables which represent diagrams are of two kinds: actual and abstract. There is an additional distinction between 'explicit' and 'implicit' diagram variables (see below); this reflects two ways in which abstractness can be generalised. The terms 'actual/abstract/explicit/implicit diagram' will be used as synonyms for 'actual/abstract/explicit/implicit diagram variable'. A single letter will be used to denote a diagram variable: upper-case for explicit, lower-case for implicit.

An *explicit* diagram is comprised of an array of variables of type vertex ("vertices") and a list of variables of type colour ("colours"). Both the colour list and the size of an explicit diagram are fixed on declaration. The constituent variables in an explicit diagram have a kind which is specified (independently for vertices and colours) on declaration. An *actual* diagram is an explicit diagram in which both vertex and colour variables are actual.

A diagram which is explicit but not actual is abstract in as much as its vertices and/or colours may be implicitly described by means of formulae. An *implicit* diagram is a variable whose value is a diag defined by means of a formula involving standard and/or user-defined operators. As explained and illustrated below, "implicit diagram" rather than "abstract diagram" is the more direct generalisation of "abstract primitive variable".

The *evaluation* of a diagram δ is a diag denoted by $|\delta|$. If δ is an explicit diagram, $|\delta|$ is derived by replacing each of the constituent vertex and/or colour variables of δ by its evaluation. (If the evaluation of a component of δ is undefined, so also is $|\delta|$.) If δ is an implicit diagram, its value is represented by a formula which returns the diagram $|\delta|$ (if defined) on evaluation.

The evaluation of a general formula (in which diagrams may appear as operands) is then unambiguously defined as in §4 viz. as the result of recursively replacing all operands by their evaluation, bearing in mind that an explicit abstract diagram and an implicit diagram are distinguished on declaration.

5.1.1. Explicit diagrams.

Syntax of explicit diagram declaration.

```
exp_diag_dec ::= [ ' colour_list ' - ] diag diagram_spec : id_list
colour_list  ::= letter { letter }
letter       ::= a | b | c | ... | z
diagram_spec ::= ( vertex_spec , colour_spec )
vertex_spec  ::= abst vert [ I_exp ] | vert I_exp
colour_spec  ::= [ abst ] col I_exp
id_list      ::= ID { , ID }
```

Examples of declarations.

```
'abc'-diag (vert 3, col 5) : D
'ab'-diag (abst vert, col 6) : E
```

diag (vert 2, col 3) : T

Semantics of explicit diagram declaration.

In the declaration of an explicit diagram, the *colour_list* is used to specify the set of valid colour-names, each of which is a single lower-case letter. The *vertex_spec* (respectively *colour_spec*) determines whether the constituent vertices (respectively colours) are actual or abstract variables.

Within an explicit diagram, the colours (whether abstract or actual) all have the same degree, which must be specified on declaration by an actual integer expression in the *colour_spec*. The size of an explicit diagram is the common degree of its colours, which is also the number of vertices in the diagram. Note that the size of an explicit diagram is fixed on declaration.

If the vertices of an explicit diagram are declared to be actual, they must all have the same dimension, which must be specified by an actual integer expression in the *vertex_spec*. If the vertices are declared to be abstract, a dimension may be specified by means of an optional integer expression, possibly abstract.

5.1.2. Vertices and colours in explicit diagrams.

Vertices and colours of an explicit diagram are specified by means of selection functions "!" and "_". Formally, an explicit diagram D of size k has associated functions, prefix and postfix respectively:

'D!': k-int → (abst or act) vertex variables

'_D': {set of colour names} → (abst or act) colour variables,

where the vertex variables may have specified dimension, and the colour variables are of degree k.

Syntax for vertices and colours of D.

vertex_of_D ::= D ! I_exp

colour_of_D ::= letter _ D

Semantics.

The definition of *vertex_of_D* and *colour_of_D* is subject to two semantic rules:

- [1] the integer expression which defines the index of a *vertex_of_D* must be actual, and define an integer of the appropriate weight.
- [2] the *letter* which identifies a *colour_of_D* must be a valid colour name for the diagram D.

Each vertex and colour of an explicit diagram is semantically equivalent to a variable of the appropriate type and kind, and all assignments to vertices and colours are subject to the same conventions and semantic rules for assignment to primitive variables explained in §4 above. In particular, in any assignment to an actual diagram vertex, or abstract diagram vertex of specified weight, the RHS must define a value of the same weight. On declaration, actual vertices (resp. colours) in an explicit diagram are initialised to the zero vector (resp. totally undefined partial perm) of the appropriate dimension (resp. degree).

When evaluated, the set of vertices and colours of an explicit diagram D of size k defines a geometric realisation of an edge-coloured digraph G. The

vertices $D!1, D!2, \dots, D!k$ of G are located at the points $|D!1|, |D!2|, \dots, |D!k|$ respectively, and there is a "c-coloured" edge (or 'c-edge') from $D!i$ to $D!j$, whenever 'c' is a valid colour name for D , and i and j are indices such that $|c_D\{i\}|=j$.

It should be emphasised that the vertices of an explicit diagram are variables of type `vert` not to be identified with the geometric points they may represent. Distinct vertices define distinct variables, even though (by accident or design) they may represent the same geometric point. Similarly, the 'edges' which colours define are abstract as opposed to geometric, that is, they connect vertices with particular indices, not geometric points with particular coordinates.

Note also that the choice of a particular colour name in a diagram is of purely "local" significance; if X and Y are two diagrams, the colours "a_X" and "a_Y" in no sense necessarily represent "edges of the same colour". A colour name merely identifies a particular colour variable associated with a diagram, in the same way that an index identifies a vertex variable.

5.1.3. Subdiagrams of explicit diagrams.

Suppose that D is an explicit diagram which has k vertices and m colours: a "(k,m)-diagram". A *subdiagram* of D is a diagram derived by restricting and/or re-ordering the colour list and/or vertex array of D .

Syntax for subdiagrams of D .

$$\text{subdiagram} ::= [\text{ ' colour_list ' } -] D [/ V_exp]$$

Examples of subdiagrams.

In the context of the diagram declarations above, the expressions 'ba'-D/[1,4,2], 'b'-E/[1,3,5] and T/[3,2,1] denote subdiagrams of D , E and T respectively.

Semantics.

Let D be a (k,m)-diagram, as above. Within an ARCA program, the identifier 'D' alone will denote the (0,m)-diagram consisting of the vertex array of D . The *colour_list* is used in order to select colours for a subdiagram; it must be a sequence of distinct valid colour names for D . The *V_exp* is used to specify the set of vertices to which the subdiagram is restricted; it must be an actual expression defining a vector of distinct m-ints (or 0-ints). In a subdiagram defined by restriction to a proper subset of the vertices, only edges which join vertices within the restricted vertex set are retained. As will be clear from the discussion of diagram operators and assignment below, the order in which colours appear in the *colour_list* of a subdiagram is significant.

5.1.4. Implicit diagrams.

As illustrated in §9, Example 3, an abstract explicit diagram is adequate for representing a graph whose structure is known, but whose geometric realisation is to be determined experimentally, or a graph in which incidence information is a simple function of an independent variable. In practice, it is also useful to be able to describe a parametrised family of graphs, such as "complete graphs on n vertices", or "Cayley diagrams of dihedral groups (appropriately generated)". An abstract description of an entire graph, rather than a family of abstract descriptions of vertices and edges is then required. Implicit diagrams are intended for this purpose (see §9, Example 4 for an illustration).

Syntax of implicit diagram declaration.

$$\begin{aligned} \textit{imp_diag_dec} &::= \text{abst} [\textit{colour_list} \textit{ - }] \textit{diag spec_opt} : \textit{id_list} \\ \textit{spec_opt} &::= [(\textit{wt_spec})] \\ \textit{wt_spec} &::= \text{vert } \textit{I_exp} [, \text{col } \textit{I_exp}] \quad | \text{col } \textit{I_exp} \end{aligned}$$

Semantics of implicit diagram declaration.

The keyword 'abst' is used to distinguish implicit from explicit diagram declaration. The *spec_opt* is an optional weight specification for the vertices and colours of the declared diagram(s). It has a defensive function analogous to the optional weight specification for abstract primitive variables; on evaluation, vertices and/or colours of the declared diagram must be of the weight specified in the *wt_spec*. The *I_exp*'s in the *wt_spec* may accordingly be abstract.

5.2. Diagram assignment.

There are two kinds of diagram assignment: 'explicit' and 'implicit'.

Syntax for assignment.

$$\begin{aligned} \textit{explicit_assignment} &::= \textit{subdiagram} := \textit{D_exp} \\ \textit{implicit_assignment} &::= \textit{D_ID} := \textit{D_exp} \end{aligned}$$

Semantic rules for assignment.

The RHS of a diagram assignment is a 'diagram expression'; an expression in standard and/or user-defined operators which returns a diagram. The informal references to diagram expressions in this paragraph will be supplemented by the formal rules for their construction and interpretation given in §6.

In an *explicit* assignment, the LHS is an explicit diagram, and the RHS supplies an appropriate family of formulae or values which are to be assigned to the constituent variables on the LHS. For this purpose, the RHS must be an *explicit* diagram expression: an expression which 'returns a diagram explicitly', in that it can be interpreted as an array of *V_exp*'s and a list of *C_exp*'s. For such an assignment to be valid, the RHS must comprise an array of *V_exp*'s and a list of *C_exp*'s of the same size respectively as the vertex array and colour list of the subdiagram on the LHS. Explicit diagram assignment is then semantically equivalent to "parallel assignment of formulae or values to some or all of the constituent variables of an explicit diagram"; each constituent primitive assignment being subject to the semantic rules given in §4.

In an *implicit* assignment, the LHS is an implicit diagram, and the RHS an arbitrary diagram expression. Semantically, implicit assignment is similar to assignment to an abstract primitive variable; the RHS is interpreted as a formula by substituting values for actual variables, and carrying out partial evaluation. The value of the variable on the LHS is subsequently defined by this formula until re-assignment.

Diagram assignment can only be fully understood in conjunction with the rules governing diagram expressions explained in §6. For simple illustrative examples, see §9.

§6. Diagrams and algebraic expressions.

Diagrams may appear in algebraic expressions in a number of ways. A diagram or component of a diagram may be used as a parameter for a standard or user-defined operator. There are also operators which return diagrams for their result; these may be used to form *diagram expressions* (*D_exp*'s) analogous to integer, vertex and colour expressions.

The semantics of expressions which involve diagrams but represent a value of primitive type is based upon the semantics of primitive expressions as set out in §4. The definition of kind (actual or abstract) for such expressions is sufficient to allow the semantic rules for primitive expressions to be generalised.

Where diagram expressions are concerned, the semantic rules for diagram assignment also require a distinction between an 'implicit' *D_exp*, which provides "a recipe for an array of vertices, and a list of colours" and an 'explicit' *D_exp*, which is "a family of independent recipes, one for each of its vertices and colours".

To specify kind for all expressions involving diagrams, it suffices to consider expressions which are derived from a diagram variable (e.g. by selection of a vertex, or restriction to a subdiagram), expressions which return diagrams, and expressions in which diagrams occur as parameters.

The precedence rules for general *D_exp*'s can be inferred from the ARCA expression parser in §10.

6.1. Expressions derived from a diagram.

To exploit diagrams, it is necessary to refer to their constituent variables. If *X* is a diagram (of arbitrary kind), there are a number of rules by which algebraic expressions can be derived from *X*. It should be carefully noted that derived expressions are defined for diagram variables rather than diagram expressions (see below).

Syntax for expressions derived from *X*.

```

size_of_X      ::= X #
V_exp_of_X     ::= X ! I_exp
C_exp_of_X     ::= letter _ X
C_path_of_X    ::= colour_seq letter _ X [ - ] [ @ I_exp ]
colour_seq     ::= { letter [ - ] [ @ I_exp ] . }
D_exp_of_X     ::= X / V_exp | ' colour_list ' - X
    
```

Semantic rules for derived expressions.

The expression *X#* denotes the size of the diagram *X*. If *X* is explicit, *X#* is an actual integer expression; otherwise, *X#* is an abstract integer expression whose value (if defined) is the size of the diag *X*.

Let γ denote an *I_exp*. The interpretation of *X!* γ when *X* is explicit and γ is actual is explained above. If *X* is implicit, or γ is abstract, then *X!* γ is an abstract vertex expression whose evaluation is the result of selecting the vert indexed by $|\gamma|$ from the diag *X*. For this purpose, $|\gamma|$ must be an appropriate index.

Let α denote a *letter*. The expression α_X is defined provided that α is a

valid colour name for X. For explicit X, the interpretation of α_X is explained above. If X is implicit, α_X is an abstract colour expression whose evaluation (if defined) is the result of selecting the col referenced by α from the diag |X|.

Since the value of a colour in a diagram is a partial perm, there is at most one edge of a particular colour into and out of a vertex. This makes it possible to reference one vertex from another in the same diagram as "the result of following a specified path made up of edges and/or reverse edges". The syntax for *C_path* is designed to simplify such referencing, so that, for example

$"a_X\{a_X\{b_X\{a_X\{i\}\}\}\}"$

(which is equivalent to $"a_X@2.b_X.a_X\{i\}"$) may be abbreviated to $"a@2.b.a_X\{i\}"$.

Abbreviation of a colour expression in this fashion does not affect its kind.

Let ν denote an vertex expression. If X is explicit, and ν is actual, then (as explained above) X/ν is a subdiagram of X: an explicit diagram whose kind (abstract or actual) is that of X. If X is implicit, or ν is abstract, then X/ν is an implicit diagram, necessarily abstract, whose evaluation (if defined) is the result of restricting the diag |X| to the vertices associated with the sequence of indices specified by $|\nu|$. For this purpose, $|\nu|$ must define a sequence of distinct valid indices for X.

Let β denote a list of distinct valid colour names for X. If X is explicit, then $\beta-X$ is an explicit subdiagram of X of the same kind as X. If X is implicit, so also is $\beta-X$. Its evaluation (if defined) is then obtained by restricting the diag |X| to the list of colour names β .

6.2. Diagram expressions.

There are four principal methods for specifying diagrams by means of operators; by restriction (defined for diagram names) as explained above, by standard operators 'join' (::) and 'product' (**), and by user-defined operators.

Syntax.

$$\begin{array}{l}
 D_exp ::= (D_exp) \\
 \quad | \quad | D_exp | \\
 \quad | \quad [' colour_list ' -] D_ID [/ V_exp] \\
 \quad | \quad D_exp :: D_exp \\
 \quad | \quad D_exp ** D_exp \\
 \quad | \quad D_OP (par_list)
 \end{array}$$

Standard operators on diagrams.

There are two standard binary operators on diag's: join (::) and product (**).

Join is defined as a map

$(m,s)\text{-diag} \times (m,t)\text{-diag} \rightarrow (m,s+t)\text{-diag}$.

If X and Y are diag's of size s and t respectively, with colour lists

$'a_1a_2 \dots a_m'$ and $'b_1b_2 \dots b_m'$

respectively, the join of X and Y has a vertex array of size s+t whose i-th element is "Xi" if $1 \leq i \leq s$, and "Y!(i-s)" if $s < i \leq s+t$, and a list of m colours, the j-th of which is specified to be $"a_j :: b_j"$.

Product is defined as a map

$$(m,s)\text{-diag} \times (n,t)\text{-diag} \rightarrow (m+n,st)\text{-diag}.$$

If X and Y are diag's of size s and t respectively, with colour lists

$$'a_1 a_2 \dots a_m' \text{ and } 'b_1 b_2 \dots b_n'$$

respectively, the product of X and Y has a vertex array of size st whose (i**j)-th element is specified to be "X!i**Y!j" for $1 \leq i \leq s$, and $1 \leq j \leq t$, and the colour list ' $c_1 c_2 \dots c_{m+n}$ ', where

$$c_k(i**j) = \begin{cases} a_k\{i\} & \text{for } 1 \leq k \leq m \\ i**b_{k-m}\{j\} & \text{for } m < k \leq m+n \end{cases}$$

for $1 \leq i \leq s$, and $1 \leq j \leq t$.

In the context of an ARCA program, there is no way of introducing a literal diag value into a formula, and join and product are used to combine *D_exp*'s. The join or product of *D_exp*'s δ_1 and δ_2 is explicit if δ_1 and δ_2 are both explicit, and is otherwise implicit. Formally, suppose that δ_i ($i=1,2$) is an explicit diagram expression comprising an array A_i of *V_exp*'s and a list L_i of *C_exp*'s. The join $\delta_1::\delta_2$ is defined provided that L_1 and L_2 have the same length, and is the explicit *D_exp* comprising the array of *V_exp*'s obtained by concatenating A_1 and A_2 , and the list of *C_exp*'s whose i-th entry is the join of the i-th entries in L_1 and L_2 .

6.3. Diagrams as parameters and results of user-defined operators.

As explained above, a diagram may be specified as a parameter or result of a user-defined operator. If an operator π requires a diagram as a parameter yet returns a result of primitive type, the kind of an expression $\pi(x_1, x_2, \dots, x_k)$ is determined in the usual fashion; it is abstract if and only if at least one of its operands x_i is an abstract expression.

If an operator π returns a diagram, the expression $\pi(x_1, x_2, \dots, x_k)$ will denote an actual diagram if all x_i 's are actual, and an abstract diagram otherwise. The semantic rules needed to distinguish between implicit and explicit diagram expressions are more complex, and will be formulated in §8, where further explanation of the conventions governing user-defined operators is given.

§7. Composite actions.

Apart from assignment, there are three standard constructs for defining actions. They are the conventional "iterative" (*while*) and "alternative" (*if*) constructs, and a *with*-construct resembling the usual *for*-statement.

There is some paraphernalia associated with complex constructs, viz: notation for boolean relations and integer lists, which is explained below. It is important to note that abstract variables and expressions cannot be assigned to control variables or appear as relations in complex constructs; their use is confined to declarations and assignments, and the complexity of a composite action derives simply from its composite nature. In particular, neither relations nor integer lists can be specified implicitly by means of abstract expressions.

7.1. Boolean relations.

Syntax.

```
relation ::= ( relation )
          | I_exp rel_op I_exp
          | exp ≠ exp
          | exp = exp
          | relation && relation
          | relation || relation
          | relation ^
          | true
          | false

rel_op ::= < | ≤ | ≥ | >
```

Semantic rules.

In all relations, expressions must be actual. In any comparison, the integer expressions must denote 0-ints. Equality of expressions naturally entails that both have the same type, and that their values are identical component for component. The operators &&, || and ^ respectively denote conjunction, disjunction and negation.

7.2. Integer lists.

Syntax.

```
integer_list ::= I_exp
              | I_exp to I_exp
              | integer_list , integer_list
```

Semantic rules.

In an *integer_list*, all *I_exp*'s must be actual and represent integers having a common modulus *m*. If *i* and *j* are 0-ints, or are principal values of *m*-ints ($m \geq 2$), then "i to j" denotes the list of integers *i*, *i*+1, ..., *j* if $i \leq j$, and is otherwise empty. The comma is used to indicate concatenation of integer lists.

Integer lists may be used in the specification of (actual) colours and vertices. The constructions

[*integer_list*], { *integer_list* } and { *integer_list* , ? }

denote vertices and colours following the conventions explained in §2.2 e).

7.3. General syntax for actions.

```
action ::= assignment | while | if | with
while ::= while relation block
if      ::= if relation block else fi
else    ::= { elseif relation block }
```

```
with ::= with int [ I_exp ] : ID := integer_list block  
      | with vert I_exp : ID := V_exp { , V_exp } block  
      | with col I_exp : ID := C_exp { , C_exp } block  
block ::= do [ action { ; action } ] od
```

Semantics.

The interpretation of *while* and *if* is conventional; in both cases, the *relation*'s which appear involve only actual expressions. In essence, an *if* specifies a list of (*relation* , *block*) pairs; each relation is evaluated in turn until a valid relation is (perhaps) encountered, when the corresponding block is executed.

A *with* contains an embedded "specification of a control variable", whose scope is the set of statements in the associated block. The type of this variable is specified by the *typename* , and its weight by the *I_exp* .

A *with* is interpreted by successively assigning each of the values specified by the associated expression list to the control variable and executing the block, as in a traditional *for*-statement. The expressions in the expression list must be actual, and represent values of the appropriate weight.

Assignment to the control variable, or to a component of the control variable of the *with*-statement is not permitted within the *with*-block.

§8. User-defined operators.

The definition of primitive operators in §2 provides a semantic paradigm for "user-defined operators" as described in this section. Like primitive operators, user-defined operators have strongly typed parameters, and are without side-effects. Note also that in defining a new operator it is only necessary (as with primitive operators) to specify the result of applying the operator to evaluated parameters. The semantics governing use of the operator in formulae (i.e. as a mechanism for combining abstractly defined values) can then be inferred.

8.1. Syntax for user-defined operators.

```
definition ::= op_header is op_body si  
op_body ::= [ action { ; action } ]  
op_header ::= op ( par_spec_list ) → type_spec : op_name  
op_name ::= ID  
par_spec_list ::= [ par_spec { ; par_spec } ]  
par_spec ::= type_spec : par { , par }  
par ::= $ NUMBER  
type_spec ::= I_spec | V_spec | C_spec | D_spec  
I_spec ::= int [ I_exp ]  
V_spec ::= vert [ I_exp ]  
C_spec ::= col [ I_exp ]
```

$D_spec ::= [colour_list] \text{diag } spec_opt$

where *spec_opt* is as specified in the syntax of implicit diagram declaration (§5.1.4), and *action* is as specified in §7.3.

8.2. Semantics of operator definition.

A *definition* is used to introduce a new operator. There are two parts to the definition; the *op_header* and the *op_body*. The *op_header* supplies details of the parameters required (via the *par_spec_list*), the type of the result (via the *type_spec*) and the name of the new operator (*op_name*). The *op_body* describes the sequence of actions which must be carried out when the operator is evaluated with given parameters.

If π is a user-defined operator of arity k , and x_1, x_2, \dots, x_k are actual expressions of appropriate type and weight (as specified in the *op_header*, as explained below), the expression $\pi(x_1, x_2, \dots, x_k)$ will denote the result of evaluating π with the parameters x_1, x_2, \dots, x_k . As with a primitive operator, if one or more of the x_i 's is an abstract expression, then $\pi(x_1, x_2, \dots, x_k)$ is regarded as an abstract expression, which may be used in specifying a formula rather than an explicit value.

Within the definition of an operator, pre-declared primitive actual global variables may be used to denote values in expressions, but may not be re-assigned. A reference to an actual global variable in a definition will be replaced by its current value (c.f. the interpretation of an expression as a formula).

In the *op_header* of an operator π of arity k , the parameters (which must be specified in a fixed order in every subsequent use of π) are referred to by "formal names" $\$1, \$2, \dots, \$k$. The header supplies a *template* for the new operator: a type for each parameter and the result, and a set of integer expressions prescribing how the weights of the parameters and result are constrained on evaluation of the operator. The *type* of an operator is the type of its result.

Formally, the set of *par*'s appearing in the *par_spec_list* must be $\{\$1, \$2, \dots, \$k\}$; each parameter then has an associated *type_spec* which comprises type and weight information. If a parameter is of primitive type, its *type_spec* resembles the declaration of an abstract variable, in that the specification of weight is optional. Similarly, if a parameter is of type diagram, its *type_spec* resembles that of an implicit diagram in that the weights of its constituent vertices and colours may be partially specified. An *I_exp* which appears in the *type_spec* of a particular parameter $\$j$, where $1 \leq j \leq k$, may involve any pre-declared primitive actual global variable, and any parameter $\$t$ which is specified prior to $\$j$ in the *par_spec_list*.

By contrast, the *type_spec* of the result, as specified in the *op_header*, must resemble the declaration of an actual variable of the appropriate type, in that specification of weight information is obligatory rather than optional. For instance, if an operator returns a diagram, the *type_spec* in its *op_header* must include *I_exp*'s to represent the dimension of its vertices and the degree of its colours. The *I_exp*'s which appear in the *type_spec* of the result may involve any pre-declared primitive actual global variable, and any parameter $\$t$.

In the *op_body* of a definition, the parameters and *op_name* are treated syntactically as though they were declared actual variables of the appropriate type and weight. This is consistent with the fact that execution of the *op_body* is only required in a context in which values for the parameters are explicitly known. Auxiliary local variables may be declared within the *op_body*, but these must be primitive and actual.

The actions in the *op_body* are subject to the usual semantic rules, together with three additional restrictions on assignments:

1. a diagram which occurs as a parameter or result of an operator can only be referenced via its constituent vertices and colours in the *op_body*
2. only the *op_name*, locally defined variables, or components thereof may appear on the LHS of an assignment
3. the *op_name* may not appear on the RHS of an assignment.

The simplicity of the actions in an *op_body* reflects the fact that a definition is intended to add an operator to the underlying algebra, rather than a subroutine to a program. The sole purpose of the *op_body* is to indicate how the parameters should be processed in order to compute a result.

In evaluating an operator for which appropriate parameter values are supplied, the specification of the result in the *op_header* is to be interpreted as a declaration of an actual variable; all assignments to the *op_name* in the *op_body* are then made to this variable, and its value is returned on termination of the *op_body*. Note that there is no special provision for exceptional termination of the *op_body* (e.g. by means of a *return* statement).

Rule 2 above eliminates side-effects. Rule 3 ensures that when an operator returns a vertex, colour or diagram, there are no implicit relationships between the components of the result.

Rule 1 relates to the use of diagrams as parameters and results of user-defined operators. As explained above, the result of applying a user-defined operator is an abstract expression if at least one of its arguments is abstract; this rule is consistent with Rule 1 in that the constituent variables of a diagram may be abstract only if the diagram is itself abstract. In the specification of a parameter of type diagram, the colour names supplied by the *colour_list* are formal. For instance, it is legitimate to substitute any expression which denotes a diagram with two colours (such as 'ba'-D or 'ab'-D::'bc'-E) in place of a parameter $\$i$ formally specified as a 'ab'-diagram. The interpretation of constituent colours within the definition body is then as would be expected on assignment of the substituted expression to a diagram variable declared as an 'ab'-diagram. Thus, under the specimen substitutions given above, $a_ \$i$ would respectively denote b_D and $a_D::b_E$.

When a diagram is specified as the result of a user-defined operator, the need to distinguish between implicit and explicit diagram expressions poses special problems. Semantic rules are required to determine when an expression defined by a user-defined operator of type diagram is to be interpreted as an explicit diagram, and how to exhibit it as a family of *V_exp*'s and *C_exp*'s in this event. The following rule is adopted: a user-defined operator of type diagram defines an explicit diagram if its arguments are such that

- (a) the size of the result (as specified in the *op_header*) is actual,
- (b) all diagram expressions supplied as arguments are explicit,
- (c) all expressions in the expression lists of *with*-statements, and all relations in *if* and *while*-statements are actual. (This condition can be easily checked provided that the parameters and components of parameters which appear in *with*, *while* and *if*-statements are monitored during the compilation of the definition body.)

When these constraints are satisfied, the definition body may be interpreted as a family of *V_exp*'s and *C_exp*'s (as the semantic rules for diagram expressions require) by "formal evaluation". The family will have known size, and expressions (possibly abstract) for each of the constituent vertices and

colours of the result can be determined. To justify this claim, it suffices to observe that all references to the input parameters in the definition body can be replaced by expressions (all arguments of type diagram are explicit, and Rule 1 applies). Moreover, the precise sequence of assignments specified in the definition body is explicit (the relevant control information in all composite statements is known), and expressions for the results of all assignments can be successively determined.

§9. Programming in ARCA.

Some simple program fragments illustrating the main principles of ARCA are presented below. In all cases, the diagrams described are "Cayley diagrams" (also called "Van Kampen diagrams") of small groups (see [DJ] Ch.5 for background details). The design of ARCA borrows ideas from Cayley diagrams (e.g. referencing vertices by colour paths, and forming products), and was originally conceived for their description. As emphasised in §1, an ARCA program is not intended merely for "generating a picture", and Cayley diagrams are a useful medium for illustrating how the ARCA description of a diagram encodes both pictorial and conceptual (viz. group-theoretic and geometric) information.

9.1. Examples of ARCA programs.

Example 1:

A program to represent a Cayley diagram for the presentation $\langle x, y \mid x^2 = y^3 = 1 \text{ and } xy = y^{-1}x \rangle$ of the dihedral group D_3 of order 6 (see Fig.1) by means of an actual ARCA diagram.

```
1  vert 2: OX;
2  'ab'-diag (vert 2, col 6) : D;
3  OX := [0,0];
4  a_D := {1,3,5}{2,6,4};
5  b_D := {1,2}{3,4}{5,6};
6  D!1 := [0,1%1]; D!2 := [0,2%1];
7  with int 3 : IX := 2,3 do
8      D!(2*IX) := rot(D!2,IX-1,OX);
9      D!(2*IX-1) := rot(D!1,IX-1,OX)
10 od
```

Example 2:

The program above generates an actual diagram to represent a specific realisation of the abstract graph in Fig.1. By modifying line 2 so that the vertices of D are declared abstract, and deleting line 6, an abstract diagram to represent a family of realisations is obtained. In each realisation, the triples (D!1,D!3,D!5) and (D!2,D!4,D!6) are at the vertices of equilateral triangles centred on [0,0], but D!1 and D!2 are at points which can be independently specified. Thus the geometric configuration could be completely specified by

$D!1 := [0,3\%1]; D!2 := [0,4\%1]$ or $D!2 := 2.D!1; D!1 := [0,2\%1]$.

In the latter case, the coordinates of all the vertices of D would depend on the coordinates of D!1.

Example 3:

The Cayley diagram of Fig.1 is closely related to the Cayley diagram for the presentation $\langle x, y \mid x^2 = y^3 = 1 \text{ and } xy = yx \rangle$ of the Abelian group $C_2 \times C_3$ (see Fig.2). The geometrical relationship between the two reflects a group-theoretic relationship; the group D_3 is a 'semi-direct' product of C_2 and C_3 (see [MH] p.88-90). The program below generates an abstract ARCA diagram which represents the abstract graph of Fig.1 or Fig.2 according to whether the abstract integer variable px is 0 or 1, and defines different planar realisations subject to the current values of $D!1$ and $D!2$.

```

1  vert 2: OX;
2  abst int 2: px;
3  'ab'-diag (abst vert 2, abst col 6) :D;
4  OX := [0,0];
5  a_D := {1,3,5}§{2,4,6}@(1-2*px');
6  b_D := {1,2}§{3,4}§{5,6};
7  with int 3 : IX := 2,3 do
8      D!(2*IX') := rot(D!2,IX-1,OX);
9      D!(2*IX'-1) := rot(D!1,IX-1,OX)
10 od

```

Example 4:

Cayley diagrams for the presentations

$$\langle x, y \mid x^2 = y^n = 1 \text{ and } xy = y^{-1}x \rangle$$

of the dihedral group D_n ($n \geq 2$), and

$$\langle x, y \mid x^2 = y^n = 1 \text{ and } xy = yx \rangle$$

of the Abelian group $C_2 \times C_n$ ($n \geq 2$) form a simple 'generic' class of graphs; each member of this class comprises a nested pair of n -cycles, in similar or inverse orientation, with corresponding pairs of vertices linked by a bi-directed edge. Such a class cannot be represented by an explicit ARCA diagram, which has fixed size, but can be by an implicit diagram. This is demonstrated by the following program, in which the ternary operator Dd requiring arguments

(int 2: \$1, int : \$2, vert 2: \$3)

is defined so that $Dd(BB,NN,UU)$ denotes a specific realisation (depending on UU) of a direct or semi-direct product (depending on BB) of C_2 and C_{NN} . Thus $Dd(1,4,[1\%1,0])$ denotes the Cayley diagram of D_4 in Fig.3.

```

1  vert 2 : OX ; OX := [0,0] ;
2  op (int 2:$1; int:$2; vert 2:$3) -> 'ab'-diag(vert 2, col 2*$2):Dd is
3      vert 2 : VV,WW;
4      VV := $3; WW := 2*$3;
5      with int $2 : IX := 1 to $2 do
6          Dd!(2*IX) := rot(WW,IX-1,OX);
7          Dd!(2*IX'-1) := rot(VV,IX-1,OX);
8          b_Dd{2*IX} := 2*IX'-1;
9          b_Dd{2*IX'-1} := 2*IX;
10         a_Dd{2*IX'-1} := 2*IX+1;
11         a_Dd{2*IX} := 2*(IX+1-2*$1)
12     od
13 si
14 abst int : bb,nn; abst vert 2: uu;
15 abst 'ab'-diag(vert 2): d;
16 d := Dd(bb,nn,uu)

```

Example 5:

A program to represent a Cayley diagram for the presentation $\langle x^2 = y^5 = (xy)^3 = 1 \rangle$ of the alternating group A_5 of order 60 (see Fig.4) by means of an explicit diagram with abstract vertices is given overleaf. (Lines 15-16, of the form "//...", are comments.)

The required diagram ('ab'-T) is constructed by first defining the skeletal subdiagram 'a'-T (lines 17-26), then inserting the edges of colour 'b'. The diagram 'a'-T is synthesised from four components: the innermost and outermost pentagons ('a'-P and 'a'-S), and the two "pentagons of pentagons" ('a'-Q and 'a'-R) which enclose 'a'-P and are enclosed in 'a'-S. The subdiagrams 'a'-Q and 'a'-R are defined using diagram product at lines 19-20.

The operator Cc is used to define regular pentagons, appropriately scaled, oriented and centred. Explicit diagram assignment is used at lines 17-18 to specify the subdiagrams 'a'-P and 'a'-S, and at lines 23-24 to specify the coordinates of Q and R. In all cases, coordinates of vertices are abstractly specified. Note that the coordinates of the vertices of Q and R specified at lines 19-20 (see the definition of diagram product in §8.2) are redundant; only the indexing and incidence information is required.

The skeleton of the final diagram ('a'-T), comprising 12 regular pentagons whose edges are of colour 'a', is defined as the join of its four components at line 26. The vertices of T are abstractly specified by formulae which depend on the abstract integer rr; in effect, rr is a 'scaling parameter' for T.

The edges of colour 'b' incident with the innermost and outermost pentagons are defined in lines 27-33, using the indexing operator '**' (§2.2 h)). Colour paths are used for subsequent vertex referencing. Referring to Fig.4: the *with*-statement at lines 27-33 introduces edges such as AB (introduced when KK=0, JJ=2 and YY=11), and the *with*-statement at lines 34-37 introduces edges such as CD (line 36, when KK=30, JJ=3, XX=42 and YY=50), EF (line 44, when KK=0, JJ=2, CC=a_T, XX=48 and YY=13) and FE (line 44, when KK=30, JJ=4, CC=a_T, XX=48 and YY=13). The usefulness of modulus coercion rules (§3.3) in conjunction with the principal value operator (§2.2 f)) is apparent here.

```
1  abst int : rr ;
2  int 60 : XX , YY ;
3  int 25 : MM , NN ;
4  vert 2 : ZZ ;
5  ZZ := [0 , 0];
6  'a'-diag (abst vert , col 5) : P , S ;
7  'a'-diag (abst vert , col 25) : Q , R ;
8  'ab'-diag (abst vert , col 60) : T ;

9  op (vert 2 : $1 , $2 ; int 2 : $3) -> 'a'-diag (vert 2 , col 5) : Cc is
10     with int 5 : HH := 1 to 5 do
11         Cc!HH := rot ($2 , HH-1 , $1);
12         a_Cc{HH} := HH - 1 + 2 * $3
13     od
14 si ;
15 // the above operator returns a regular pentagon with centre $1,
16 // first vertex at $2, and orientation specified by $3 (= 1 or 0).

17 'a'-P := 'a'-Cc(ZZ , [rr,0] , 0);
18 'a'-S := 'a'-Cc (ZZ , (-4).P!1 , 1) ;
19 'a'-Q := P ** 'a'-P ;
20 'a'-R := P ** 'a'-P ;
21 with int 5 : II := 1 to 5 do
22     MM := II**1; NN := II**5;
23     Q/(MM..NN) := Cc(2.P!II , (3/2).P!II , 0) ;
24     R/(MM..NN) := Cc((-3).P!II , (-7/2).P!II , 0)
25 od ;

26 'a'-T := 'a'-P :: 'a'-Q :: 'a'-S :: 'a'-R ;

27 with int 5: JJ := 1 to 5 do
28     MM := JJ ** 1 + 5 ;
29     with int 60: KK := 0, 30 do
30         b_T{KK+JJ}' := KK+MM' ;
31         b_T{KK+MM}' := KK+JJ'
32     od
33 od ;

34 with int 60: KK := 0, 30 do
35     with int 5: JJ := 1 to 5 do
36         XX := a.b_T{KK+JJ}' ;
37         YY := a^-b.a_T^{-}{KK+JJ}' ;
38         b_T{XX} := YY ;
39         b_T{YY} := XX ;
40         with col 60 : CC := a_T , a_T^- do
41             XX := CC@2.b_T{KK+JJ}' ;
42             YY := CC@2.b_T.CC^-@2{KK+30+JJ}' ;
43             b_T{XX} := YY
44         od
45     od
46 od
```

9.2. Miscellaneous remarks on ARCA programming.

The examples given above are an inadequate basis for assessing the merits of ARCA. Like many combinatorial notations, the ARCA notation appears complicated, though much of this complication derives from the plethora of standard operators, and is syntactic rather than semantic. Superficially, ARCA programs are tricky to write, but this may reflect the innate intricacy of many combinatorial diagrams. For instance, the construction of the abstract Cayley diagram of Example 5 implicitly involves generation of all even permutations of five symbols, and additional geometric and group-theoretic information is also encoded.

Procedural equivalents of ARCA programs are fairly easy to conceive, and seem unlikely to be less complex. The prospect of developing a functional language for describing combinatorial diagrams is more inviting, and gives more hope of simplification.

Where Cayley diagrams are concerned, coset enumeration may be used to derive the abstract Cayley diagram graph from the equational description implicit in the algebraic presentation, but the major problem of specifying a particular realisation remains. In general, it might be of interest to use coset enumeration in conjunction with an ARCA package, perhaps specifying some of the implicit geometric relations between vertices automatically (see [JFB] for some first experiments of this kind). In the absence of automatic layout heuristics, coset enumeration may lead to a haphazard indexing of vertices, necessitating re-indexing before realisation by the methods above can be ventured. (The operator '%%', whose use is not illustrated above, was conceived in this connection for assisting decomposition of an abstractly generated Cayley diagram into component subgraphs complementary to the synthesis illustrated at line 26 in Example 5.)

The use of ARCA for describing other combinatorial diagrams has not been considered in detail; following traditional practice, this exercise is left to the reader, but some hints are offered. As mentioned in §5.1.2, the colours of an ARCA diagram can have abstract rather than geometric significance, so that a diagram may be used as a "carrier" for a geometric configuration. Many symmetrical diagrams can be realised as Cayley diagrams in which edge information (such as direction or colour) has been discarded, or in which abstractly distinct vertices are geometrically coincident. (For example, the icosahedral graph is obtained by identifying the vertices of pentagonal faces in Example 5, and the Boolean lattice of subsets of an n element set is the abstract presentation of C_2^n .) The use of ARCA for the display of other symmetric objects such as "the lattice of partitions of an n -element set" is more problematical, but it should be possible to obtain some simplification by imposing geometrical constraints on the partitions within a symmetry class, thus reducing the number of degrees of freedom. One limitation of ARCA may be significant in this context: though some generic classes of graphs can be described by means of implicit diagrams (c.f. Example 4), the limited notation for diagram colours in ARCA precludes encoding classes (such as Cayley diagrams for presentations

$$\langle x_1^2 = x_2^2 = \dots = x_n^2 = 1 \text{ and } x_i x_j = x_j x_i \rangle$$

of C_2^n ($n \geq 2$)) which require a variable number of colours.

In practical ARCA programming, the nature of the display interface will be very important. It will not be enough to adjoin the command
"display(*diagram*)"
to ARCA, since additional information on the colour and direction of displayed

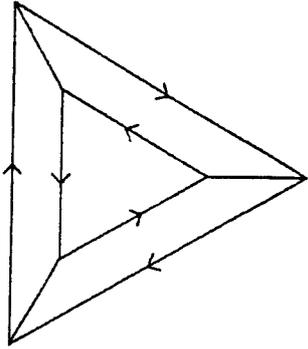


Fig. 1

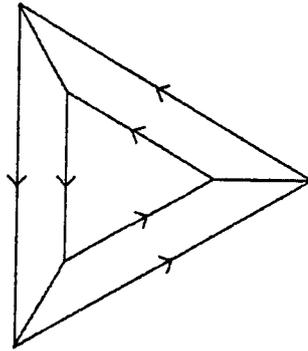


Fig. 2

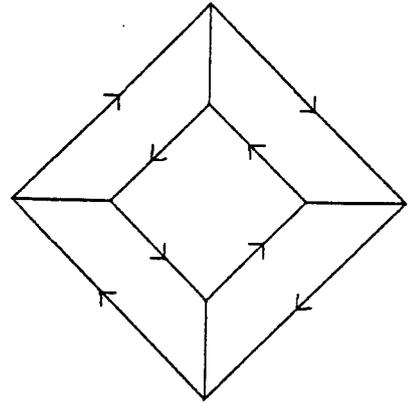


Fig. 3

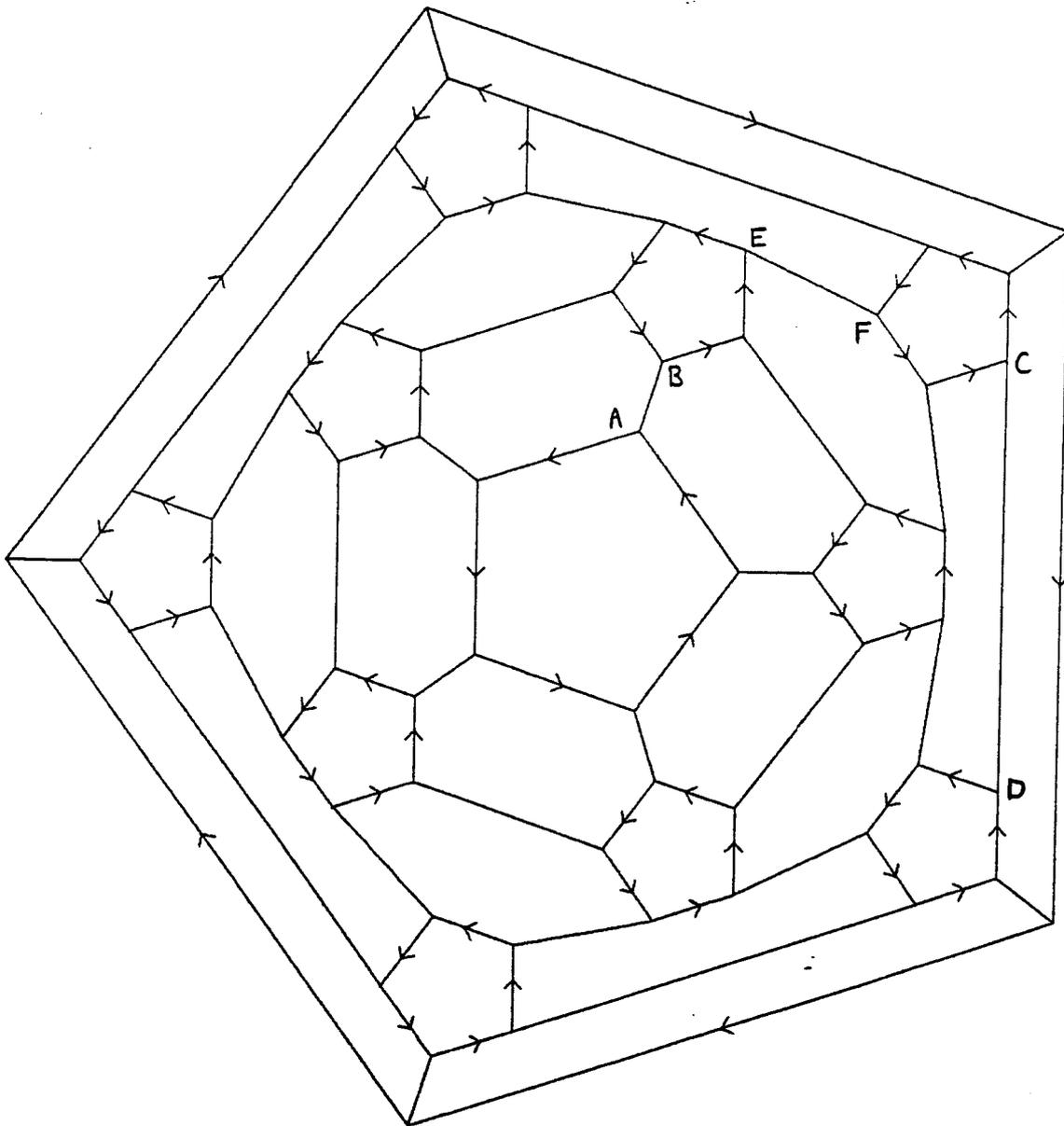


Fig. 4

edges is needed. ARCA is most conveniently used for describing the intrinsic geometry of a diagram, and auxiliary means of performing geometric transformations such as projection, translation or magnification of a component would be helpful.

Vertex referencing might also be simpler in practice. Whilst referencing by colour paths is effective in a highly connected graph, a more rudimentary mode of referencing is needed when synthesising a graph from many components (c.f. Example 5, lines 27-33). Such referencing might be done directly by identification of (presumably non-coincident) points on the display by means of a mouse or light-pen.

§10. A parser for ARCA expressions.

The file overleaf can be used as the basis for a YACC-generated parser for ARCA expressions and relations. It has been included in order to provide definitive precedence rules, and might not be appropriate for use in a full ARCA interpreter without modification. (For instance, ARCA relations are semantically different from other algebraic expressions, and cannot appear in a parameter list.)

The parser accepts typed identifiers and constants from the lexical analyser. A colour name specified by a lower-case letter is passed as a LETTER token. The lexical analyser is intended to recognise a construction of the form

'LETTER*'-

as a "colour string", and return the token STRING. Special names have been given to certain tokens; the symbolic equivalents are generally obvious, but for a few cases: AND(&&), OR(||), DOTDOT(..), MODMOD(%%), PRIME('), SMASH(**), JOIN(::). Note that the precedence rules do not distinguish between semantically different occurrences of the same token; the typing of expressions resolves any such ambiguities. Note also that specific precedences have been given to two nullary productions (using "%prec ...") in order to eliminate all parsing conflicts.

The format of the file has been modified slightly to save space; the alternatives in rules should be on separate lines.

```
%token I_ID V_ID C_ID D_ID I_OP V_OP C_OP D_OP NUMBER LETTER
%token AND OR symTRUE symFALSE
%token EQ GE LE NE DOTDOT MODMOD PRIME SMASH JOIN ROT REF STRING
%left AND OR '^'
%left '+' '-'
%left JOIN SMASH '*'
%left '$'
%nonassoc MODMOD DOTDOT
%nonassoc '['
%left '.' '/' STRING
%nonassoc '@'
%left '~' '#' '...' PRIME
%nonassoc '_' '!' '%'
%start exp
%%
exp      : I_exp | V_exp | C_exp | D_exp | rel ;
expl     : exp | expl ';' exp ;
I_expl  : I_exp | I_expl ';' I_exp ;
I_exp    : '(' I_exp ')' | '|' I_exp '|' | NUMBER | I_ID
          | I_OP '(' expl ')'
          | I_exp '#' | V_exp '#' | C_exp '#' | D_exp '#'
          | I_exp '+' I_exp | I_exp '-' I_exp | I_exp '*' I_exp
          | I_exp '~' | I_exp '%' I_exp | I_exp PRIME
          | I_exp MODMOD V_exp | I_exp SMASH I_exp
          | V_exp '[' I_exp ']' | C_exp '{' I_exp '}'
          | '<' V_exp ';' V_exp '>' | I_exp '...';
V_exp    : '(' V_exp ')' | '|' V_exp '|' | V_ID
          | '[' I_expl ']' | I_exp DOTDOT I_exp
          | V_OP '(' expl ')'
          | V_exp '+' V_exp | I_exp '.' V_exp | V_exp SMASH V_exp
          | ROT '(' V_exp ';' I_exp ';' V_exp ')'
          | REF '(' V_exp ';' V_exp ';' V_exp ')'
          | '(' I_exp '/' I_exp ')' '.' V_exp
          | D_exp '!' I_exp ;
C_exp    : '(' C_exp ')' | '|' C_exp '|' | C_ID
          | '{' I_expl '}' | '{' V_exp '}'
          | C_OP '(' expl ')'
          | LETTER '_' D_exp | C_seq LETTER '_' D_exp inv_o at_o
          | C_exp '.' C_exp | C_exp '~' | C_exp '@' I_exp
          | C_exp '$' C_exp | C_exp JOIN C_exp ;
C_seq    : LETTER inv_o at_o '.' | C_seq LETTER inv_o at_o '.' ;
inv_o    : %prec '.' | '~' ;
at_o     : %prec '.' | '@' I_exp ;
D_exp    : '(' D_exp ')' | '|' D_exp '|' | D_ID
          | D_OP '(' expl ')'
          | D_exp '/' V_exp | STRING D_exp
          | D_exp JOIN D_exp | D_exp SMASH D_exp ;
rel      : '(' rel ')' | I_exp relop I_exp
          | I_exp EQ I_exp | V_exp EQ V_exp | C_exp EQ C_exp | D_exp EQ D_exp
          | I_exp NE I_exp | V_exp NE V_exp | C_exp NE C_exp | D_exp NE D_exp
          | rel AND rel | rel OR rel | rel '^' | symTRUE | symFALSE ;
relop    : '>' | GE | '<' | LE ;
```

Epilogue.

The ARCA notation is neither purely procedural nor purely declarative, but combines features of both varieties of programming notation. The essential principles on which it is based are simple, and can be generalised. Given any algebra A, it is possible to define 'actual' and 'abstract' expressions over A, and develop a notation in which variables either have explicit values or values implicitly defined by formulae. The primary aim of this report is to indicate informally how such an "algebraic notation" might be designed and adapted for practical use, and prepare the way for a thorough evaluation of the merits and limitations of this approach.

The present work suggests both theoretical and practical issues for further investigation.

From a theoretical viewpoint, it would be helpful to formulate the concept of "an algebraic notation" more precisely. If a conventional form for composite actions is assumed, there are two principal considerations: how the underlying algebra should be specified, and what variables should be defined. Ideally, specification of the underlying algebra should perhaps be enough to determine the precise form of the associated notation, but this is not easy to ensure. For instance, the ARCA notation may be regarded as a tentative design for an algebraic notation based upon a concretely presented algebra whose data objects include integers, vectors, perms and diagrams. In that context, the fact that vectors and perms are arrays of integers prompts the introduction of variables to represent components of actual vertex and colour variables. Such component variables can be omitted only at the cost of practical inconvenience, as it is very often desirable to be able (for example) to specify incidence information incrementally. The hierarchical relationship between a diagram and its constituent vertices and colours leads to additional complications, as the component variables in this case may be of abstract kind. It is significant that the definition of a convenient practical notation (apparently) depends on knowledge of concrete data representations; it is not clear that a satisfactory characterisation of *l*-values in the algebraic notation can be inferred from a fully abstract specification of the algebraic operators relating integers, vertices, colours and diagrams. Perhaps the proper specification of the underlying algebra should comprise an abstract axiomatic algebraic specification together with information about data representations in the concrete algebras used for evaluation. This would combine the advantages of an abstract specification (e.g. enabling evaluation procedures to operate in conjunction with formula manipulation routines) with those of precisely defined data representations (e.g. means to define 'derived data-types' as well as derived operators).

From a practical viewpoint, experiment is needed to evaluate the merits of algebraic notations, and to explore potential further applications.

Superficially, at least, an algebraic notation is too restricted to provide the basis of a general purpose programming language. Recursive definition does not fit naturally into the semantic framework of value or formula assignment, and it is questionable whether a convenient formalism for handling values and relationships between values is adequate for general applications. Whatever the application, it may be argued that an algebraic notation is so constrained by its hybrid nature that it lacks the best features of both procedural languages (general data types and subroutines) and functional languages (clear semantics and higher order functions). Certainly, such a notation is very easily abused (e.g. by frequent re-assignment of formulae to an abstract variable). On the other hand, our conceptual models of our environment are rarely purely procedural or purely declarative in nature, and generally combine elements of

both kinds. It may even be that the clarity of such models derives from our ability to abstract functional and procedural aspects.

ARCA is intended to provide one context in which to examine and evaluate these arguments. It is based primarily upon the thesis that in conceiving combinatorial diagrams and developing geometric realisations interactively, it is convenient to imagine fixed relationships between points whose geometric locations are subject to change. In a 'good' ARCA program, it is to be expected that the relationships between values established by formula assignment are transparent, and not subject to frequent revision. It is significant here that the ARCA notation has been designed with an interactive environment in mind, and an implementation is essential to assess any advantages it may have over alternative notations as an interactive tool. This implementation may itself be of incidental practical interest, and might incorporate ways of avoiding redundant re-evaluation, and of presenting current information about variable dependences.

In any event, it may be worth seeking other applications for algebraic notations. Though ARCA itself is directed towards a specialised (perhaps esoteric) graphics application, the counterpoint between values and relations upon which it is based may have wider applicability. For instance, it is well-recognised that the dependencies within a data-base are conveniently described by means of functional relationships. If these are expressed in terms of algebraic operators appropriate to the particular type of data-base, the associated algebraic notation may provide the basis of a query language. As a clichéd example, it may be reasonable to define an "abstract algebra" based on the primitive relations of a library data-base, and to use concrete algebras for interrogation and manipulation associated with particular libraries. Conceptually, at least, axiomatisation of the basic algebra might be helpful in checking data integrity, and the distinction between value and formula assignment might assist security.

The complex problems of managing systems of files generated during project development might also be ameliorated by synthesising actual and abstract file definitions in an appropriate algebraic notation. This is obliquely illustrated by the UNIX 'make' utility: a procedural mechanism which in effect carries out the evaluations of abstractly defined files needed to maintain compatibility between versions. The relationships between files required by 'make' are important in the user's conception of the file system, and might be more conveniently manipulated by means of an algebraic notation implemented so as to allow inspection of abstract file definitions.

Acknowledgements.

I am much indebted to Nader Fahrenak for his patience and persistence in systematically recording and collating ideas during the initial phase of the development of ARCA. But for his third-year project [NF], this design would not have been attempted.

I am also grateful to Graham Exley for helping to identify some of the inadequacies of the design at intermediate stages, and for developing an interpreter for an ARCA subset [GE].

References.

- [DJ] D.L. Johnson
Presentation of Groups, LMS Lecture Note Series 22, CUP 1976.
- [MH] Marshall Hall
The Theory of Groups, Macmillan, 1959.

- [JB] John Buckle
Cayley diagrams by computer, 3rd year CS project, 1983.
- [GE] Graham Exley
An ARCA interpreter, 3rd year CS project, 1983.
- [NF] Nader Farahnak
The ARCA language, 3rd year CS project, 1982.