DEFINITIVE NOTATIONS FOR INTERACTION

Meurig Beynon
University of Warwick, Coventry CV4 7AL, England

Abstract.

This paper explores some methodological and pragmatic aspects of the design of the human-computer interface. In particular, it argues that many interactive dialogues can be formulated conveniently and clearly using notations based upon sequences of definitions ("definitive notations"). Such a notation is an implicit ingredient in the "spread-sheet" packages which have recently become so popular in business applications. To apply similar principles to more complex tasks, such as CAD applications, requires abstraction and generalisation, and poses challenging technical problems.

The three sections of the paper respectively consider: background and motivation, elementary definitive notations (illustrated by an unconventional desk calculator), and complex definitive notations (with particular reference to the design of ARCA, a definitive notation for the interactive description and manipulation of combinatorial diagrams).

Introduction.

The purpose of this paper is to investigate a generic class of programming notations for conducting dialogues. Use of the term "programming notation" rather than "programming language" - as advocated by Dijkstra ([4] p.8) - is more than usually appropriate here, since the applications contemplated are special purpose rather than general purpose. Indeed, the dialogues under consideration are of a relatively simple kind, typically resembling the use of a spread-sheet or some generalisation thereof. Because dialogues in these notations consist of sequences of definitions, the term "definitive notation" is adopted; a pun for which "functional language" is perhaps a good precedent - if indeed definitive notations are as definitive as functional languages are functional. Since a description of the concept of "a definitive notation" is beyond the scope of this brief introduction, the reader may find it helpful to refer to §2 below for details at this point.

The paper is in three sections. §1 is concerned with background and motivation, and considers in particular how both procedural and declarative elements are useful in formalisms for dialogue. In §2, the concept of an elementary definitive notation is introduced and illustrated by a simple but unconventional desk calculator based on principles abstracted from spread-sheets. §3 examines the problems encountered in developing generalised definitive notations suitable for more ambitious applications, and is illustrated with reference to ARCA, a definitive notation for the interactive description and manipulation of combinatorial diagrams.

Background and motivation.

Definitive notations are conceived as a suitable medium for dialogues over some limited universe of discourse. Such a dialogue can be viewed as a sequence of human-computer / computer-human transactions which (it may be reasonably assumed) have common points of reference. In the context of this paper, the term "interaction" is used in its narrow sense to mean "a dialogue in which many transactions occur". In view of the variety of connotations which "interaction" has acquired in computing, some clarification may be helpful. Firstly, the *nature* of the physical interface (e.g. graphical or textual) is not considered here. Secondly, it is important to recognise that interaction often plays a more incidental role than it might superficially appear. For instance, in solving a problem, interaction may merely entail editing a computer program which is then used directly to determine the required solution. In such a case, the dialogue itself is quite unrelated to the problem to be solved.

Typical pertinent examples of interaction are activities (such as the maintenance of a personal file system, the use of a relational data-base, or the use of a spreadsheet) which are primarily concerned with describing and manipulating a conceptual model with the aid of a computer, rather than the solution of a specific problem. As far as "problem solving" is concerned, the emphasis is on problems in which human experimentation and possibly aesthetic judgement play a part in the solution.

As a motivating example, consider the interaction involved in maintaining a personal system of computer files. The most rudimentary file system provides only a *variable* for each file (viz. a filename) and at any time simply stores the *value* of each variable (viz. the contents of the appropriate file). In practice, the user has a conceptual model of her file system far richer than the mere set of "variable values" stored by the computer. For example, certain families of files relate to particular subjects, and within such families there are functional relationships between files, such as

"S is the source of the object file O".

Problems of interaction typically arise when a user resumes a dialogue about her file system after a lapse of time. Though file/subject association can be assisted by filename mnemonics on the part of the user, or the existence of directories in the file system, the identification of functional relationships often remains difficult.

The example illustrates difficulties which are common to many dialogues. When a suspended dialogue is resumed, it is desirable that - as far a possible - the conceptual model which the user requires can be readily reconstructed. Taking the above example as a paradigm, it will be assumed that the user's conceptual model can be represented by a set of variables together with

(a) a set of current values

and (b) a set of functional relationships between variables.

It will be convenient to refer to (a) as the set of *transient values*, and (b) as the set of *persistent relations*. This terminology reflects the fact that, in the

context of a particular dialogue, relationships between variables tend to be stable, whilst values are generally subject to change. Thus (in the above example) the contents of the source file S may be revised many times, but the relationship between O and S will be maintained through re-compilation. As explained below, it is significant that neither "transience" nor "persistence" is interpreted too literally. In practical terms, fixing the value of a variable can be viewed as "defining a functional relationship", whilst it is sometimes appropriate to modify an existing functional relationship in the course of a dialogue e.g. O might be defined by compiling S with different options at different stages of program development. More generally, a mode of dialogue which does not permit convenient modification of both values and relations is too inflexible.

The manipulation of "transient values" and the specification of "persistent relations" are the respective characteristics of the *procedural* and *declarative* approaches to programming. Strictly speaking, the term "declarative" is legitimate only where "referential transparency" pertains, which in this context would preclude re-definition of variables once defined. It is nevertheless useful to think in terms of these classifications, using "declarative" in a relative sense to describe definitions which apply throughout a series of transactions, but are not necessarily eternally valid.

Subject to this qualification, the above discussion suggests that a good formalism for a dialogue requires both *procedural* elements (viz. re-assignment of transient values) and *declarative* elements (viz. functional definition of persistent relationships). In particular, a purely procedural formalism is not well-adapted for dealing with persistent relationships. For instance, the traditional procedural emphasis of most operating systems means that the user must update the object file O explicitly whenever the source file S is modified. In effect, monitoring persistent relations is the responsibility of the user. Even where some provision is made for relationships between files to be recorded (as in the "make" utility in UNIX), the updating procedure still has to be invoked explicitly.

As for purely declarative formalisms for dialogue, referential transparency is a strong restriction. A paradigm for a declarative program is "specifying a system of equations to be solved by the computer": a single transaction which - if dialogue is to be meaningful - may either be seen as defining a function (as in "programming with equations" [5]) or as defining a functional relationship between variables (as in "programming with constraints" [3]). In the former case, such a function might correspond to the aggregate of (strictly) persistent relations, and its parameters to the (implicit) variables to which transient values can be assigned, but not in general in such a way as to make the updating required for dialogue convenient. In the latter case, the cumulative adjunction of equational constraints, whilst appropriate when iterating towards the solution of a specific problem, inhibits dialogue unless some provision is made for modifying or forgetting constraints. Despite these difficulties, the use of declarative principles has a distinguished pedigree in such remarkable "object-oriented language" systems as APT (Douglas Ross [8]), SKETCHPAD (I.Sutherland

[10]) and ThingLab (Alan Borning [3]). The objectives of ThingLab in particular are topical in this context, and it would be interesting to understand how the use of definitive notations is related to the constraint-oriented approach taken in [3].

The formalism for interaction proposed in this paper reflects the ideas above. It will be assumed that the user's conceptual model is defined with reference to an underlying algebra appropriate for the universe of discourse. This algebra might be "integers with arithmetic operators" (using a spread-sheet), "relational algebra" (using a relational data-base) or "files with file operations such as concatenation, compilation, compaction etc." (maintaining a file system). The dialogue between user and computer is then expressed in terms of variables which may if defined have associated values in the underlying algebra. The values of these variables play the role of the "transient values", and the persistent relations are formulae which define the value of a variable in terms of the values of other variables and constant values using the operators of the underlying algebra. In the context of the illustrative example above: the file variables S and O respectively denote source and object files, the values of the variables S and O are their respective contents, and "O=compile(S)" is a persistent relation defining the contents of the corresponding object file.

Use of a spread-sheet is a simple example of interaction along these lines. In a spread-sheet system, in its simplest form, scalar variables can be prescribed to correspond with entries in a displayed table. Each variable can then be given an explicit scalar value, or assigned a arithmetic formula specifying how its value is to be computed from the values of other variables. Spread-sheets have proved to be a very popular and effective medium for numerical calculations in which experiments with parameters play a part. Perhaps the most attractive feature of this approach is that both persistent relations and transient values are stored by the computer, and the effect of changing the value of a particular variable can be automatically computed and displayed.

The same pattern of interaction is used in the relational algebra query language ISBL ([11] p.177-181). In this context, the relationship between the "intension" of a data-base relation (or equivalently, a *view*) and its extension at a particular time corresponds precisely to the relationship between a defining formula and its current value. The definition of views by "delayed evaluation" is then semantically equivalent to formula definition in a definitive notation based upon relational algebra. Indeed, the distinction between ISBL and a definitive notation is purely syntactic - the syntax in ISBL being biased towards value assignment rather than formula assignment.

Elementary definitive notations.

In this section, the basic concept of a "definitive notation" is described and illustrated with reference to a simple but unconventional desk-calculator (udc) based upon principles abstracted from spread-sheets. In the interest of brevity and readability, formal details are omitted, but the reader should require little

imagination to supply these.

Informally, a definitive notation is associated with an "underlying algebra" which is chosen to reflect the universe of discourse. Variables in the definitive notation are used to denote elements of the underlying algebra, and a typical program consists of a sequence of statements, each of which defines a variable or interrogates a variable. Definitions in such a notation can be conceptually regarded as being of two types: *formula assignment* such as

$$a = f(b,c,...z)$$

(semantically: unless and until a is re-defined, its value is determined as required by evaluating the formula f(b,c,...,z) over the underlying algebra), and *value assignment*

$$a = C$$

(semantically: unless and until a is re-defined, it has the explicit value C). In this context, formulae are defined in terms of the operators of the underlying algebra, and the expression |g| can be used to denote "the value of the formula g" as required. After a sequence of definitions, the value of a particular variable - which may be undefined - will in general depend upon the definitions of a set of variables in a non-recursive manner. Interrogation of a variable or formula can then be used to determine the family of definitions upon which its value notionally depends, and to determine its value if it exists.

In the simple case of the udc, the underlying algebra is the set of integers together with an appropriate set of operators, which is here assumed to contain standard arithmetic and relational operators, and an arithmetic "**if** ··· **then** ··· **else**". It should be clear that the udc is essentially a primitive spread sheet from which the tabular interface has been removed. It may also be observed that in any udc dialogue, the current values of variables and algebraic relationships between variables are easily determined at any stage. The essential characteristics of udc dialogue are illustrated below; the formal specification of the udc is left as an exercise to the imaginative reader.

In the udc, the assignment "a = b+c*4" associates the *formula* "b+c*4" with the identifier a, specifying that (until a is re-defined) the value of a is to be determined, as and when required, by evaluating the formula "b+c*4". Thus the value of a is implicitly rather than explicitly defined, and the values of b and c at the point of definition are irrelevant, and need not be defined. There is then a distinction between "a is undefined" (ie. there is no formula currently associated with a), and "the value of a is undefined" (ie. either a is undefined, or the value of a is defined by a formula which currently has no evaluation).

As a simple example of a udc dialogue, consider the following sequence of definitions:

1. e=**if** b<d **then** b **else** d
2. b=7
3. a=b+c*4
4. c=b+d-2
5. d=8
6. b=3+d

Definition 1 ensures that, throughout the dialogue, e has the same value as the smaller of b and d. Assuming that all variables are initially undefined, definitions 2-4 will ensure that b is defined by the formula '7' and has value '7', a and c are respectively defined by the formulae "b+c*4" and "b+d-2", but have undefined values, and the variable d is still undefined. Definition 5 defines the variable d explicitly, and in the process the values of a and c become defined. Thus, the value of c is now

(the value of b) + (the value of d) - 2 = 7 + 8 - 2 = 13

and the value of a is (similarly) 7 + 13 * 4 = 59. Definition 6 then re-defines b, so that the values of a,b,c and d now become 79, 11, 17 and 8 respectively.

Some semantic restrictions on assignment are of course needed. For example "a=a+2" is meaningless, as it leads to a circular definition of the value of a. Indeed, it is necessary to prevent sequences of assignments after which a variable is implicitly defined in terms of itself. For instance, in the context of the above illustration, it would be meaningless to follow the sequence of definitions 1-6 by "d=a", since the value of a is indirectly defined in terms of d. It is not difficult to detect such circularity of definition automatically.

As in a conventional calculator, it is often convenient to specify an explicit value as "the value of an algebraic expression". Thus (after the sequence of definitions 1-6 above) the assignment "c=|a+b|*d" will re-define c by the formula "90*d", and (it should be noted!) is not circular, though the value of "a+b" depends upon the value of c at the point of definition.

Facility in handling undefined values is a feature of definitive notations, and it is useful to be able to erase redundant definitions as necessary. For this purpose, the symbol '@' is used to represent 'undefined', so that "a=@" deletes the current definition of the variable a.

The semantics of definitions and expressions have been explained in general terms above, but there are some subtle points. The rules for associating values with expressions are complicated when (in the course of evaluation, or through user-unfriendliness) undefined values occur in the context of conditional expressions.

The semantic rule adopted by the udc assigns values to conditional expressions by "lazy evaluation". Under this convention, an expression has a value provided that all subexpressions which need to be evaluated have a value. Thus

" **if** 1 < 2 **then** 3 **else** @ "

has value 3, as the alternative supplied by the **else**-clause is not needed for evaluation. Such semantic considerations also bear upon validating

assignments: thus

" a = **if** 1 < 2 **then** a **else** @ "

would be valid if the RHS were interpreted as 'undefined', but is invalid when lazy evaluation is invoked.

Limited as an elementary definitive notation is, it has some merits. The success of spread-sheets probably depends on two characteristics of the udc: its suitability as a medium for experiments with numerical parameters, and the conceptual simplicity of the transient values and persistent relations.

It is natural to consider whether persistent relations can be better modelled by equational constraints, but there are two main problems. Firstly, constraints can be hard for the user - if not infeasible or undecidable for the computer - to interpret, and the semantic content of persistent relations can easily become obscure. Secondly, a graceful method of eliminating or modifying constraints is needed if relations are not to be *too* persistent.

In this context, purely declarative principles may be more appropriately used to permit the definition of additional arithmetic operators, as in

" fact(n) = **if** n=0 **then** 1 **else** n*fact(n-1) ".

Complex definitive notations.

The essential characteristics of a definitive notation have been outlined in the previous section. In most practical applications, the underlying algebras which are required for useful interaction are very much more complicated than have so far been considered. To illustrate this, and indicate how definitive notations may be developed for more ambitious applications, some of the problems of designing a definitive notation for a graphical application will now be discussed. This discussion focuses on general principles rather than specific details, but is loosely based upon experience in the design of ARCA, a definitive notation for the description and manipulation of combinatorial diagrams. (For more background on ARCA, the interested reader should refer to [1] and [2].)

Graphics is an obvious application area for definitive notations. Displaying a diagram is a task which may involve human experimentation and judgement. It must be emphasised however that if displaying a diagram is seen as an end in itself (which it sometimes *is*), the value of a definitive notation is dubious. Only if diagram display is the prelude to further interaction can the use of an appropriate definitive notation be clearly advantageous, for only in these circumstances is the a need for a conceptual model of the final diagram apparent.

In practice, it is often the case that a diagram has semantic content of a non-geometric nature, and that graphical systems which can depict a diagram easily provide no assistance when subsequent interaction and interpretation is required. In designing a definitive notation to display such diagrams, the underlying algebra is chosen to reflect all relevant semantic information. In using the notation, the emphasis is not on facile generation of an image, but on systematic specification of a conceptual model (i.e. a set of transient values and persistent relations) which enables display and interaction.

Where ARCA is concerned, the objects to be displayed are combinatorial graphs of a type introduced by Arthur Cayley in connection with algebraic research in group theory. A Cayley diagram is a geometric model of a group which encodes the multiplication table and gives direct visual interpretation to algebraic relations within it. For example, Fig.'s 1 and 2 below are Cayley diagrams respectively representing the cyclic group C_6, and the symmetric group S_3: the two abstract groups of order 6. The mathematical details are unimportant here: suffice to say that *all* semantic information in a Cayley diagram can be expressed in a definitive notation for which the underlying algebra is chosen to include scalars, vectors, permutations, and a class of graphs resembling partial Cayley diagrams. In this algebra, there is a plethora of operators (c.f. [1]), including

arithmetic operators on scalars,
scalar product of vectors,
vector addition,
product of permutations,
selection of a vector component,

together with special operators (such as are needed to combine subgraphs) reflecting the peculiar character of the semantic domain.

The choice and design of the underlying algebra for a particular application area is a very significant and difficult exercise in general; the adoption of relational algebra as a framework for interaction with data-bases illustrates this. The algebra of arrays and operators in APL, designed by Iverson as a medium for computational linear algebra, is another interesting precedent. It is clear that in many potential application areas the underlying algebra will be many-sorted, and include a wide range of operators. A definitive notation which made it possible (say) to describe a VLSI circuit comprehensively would doubtless require a very complicated underlying algebra, but clear semantics of expressions is guaranteed provided that each operator is semantically simple.

When the underlying algebra is many-sorted, the variables of the associated definitive notation have to be typed. Thus, in ARCA, there are variables to represent scalars, vectors, permutations and graphs. Declarations can be used to distinguish variables of different types, but there are additional complications where higher data-types are concerned. (The limitations of ISBL, which does not cater for manipulation of relations ([11] p.181), probably testify to this.) To illustrate the difficulties, it will be enough to consider a definitive notation in which scalar and vector variables co-exist; the reader should be able to imagine how the problems and proposed solutions generalise.

Suppose first that there are *precisely* two types of variables: scalar and vector. If the underlying algebra has operators

$$[\ ,\ ] : S \times S \to V \text{ and } '.' : S \times V \to V$$

(where S and V respectively denote the scalar and vector elements), v is a vector variable and s and t are scalar variables, then a typical definition of v assigns an expression of vector type, as in

"v = s.[t,2]".

Though a formalism based on such definitions is both simple and elegant, it has serious practical limitations (c.f. ISBL). One of the main reasons for using higher-order types is to allow incremental definition of values, as in

"v[2] = s*2 ; v[1] = s*t".

It is also very convenient to be able to re-define a single component of a vector (or more generally "move a single vertex of a graph"). However, a sequence of definitions such as

"v = s.[t,2] ; v[2] = 3"

is semantically quite unacceptable.

The above argument suggests the need for two distinct kinds of vector variable: the ***abstract*** variable, whose value is defined by a formula of vector type, and the ***explicit*** variable, which is semantically equivalent to an indexed family of scalar variables, and whose value is defined component by component by formulae of scalar type. (This distinction between "abstract" and "explicit" variables is a refinement of the "abstract" vs. "actual" distinction described in [1] and is incorporated in the revision of ARCA to which the illustrations below refer.) Where abstract and explicit vector variables are concerned, different semantic conventions then govern declaration and assignment. Thus, if v and w respectively denote abstract and explicit variables, v can only be appear in definitions of the form "v = ...", whilst w has a fixed dimension δ specified on declaration and is to be defined component by component using definitions of the form "w[α] = ...", where $1 \leq \alpha \leq \delta$, and α and δ denote parameterless scalar expressions. (With a little ingenuity, some assignments of the form "w = ..." can also be interpreted in this manner.)

Introducing explicit vector variables makes a limited form of recursive definition possible. For instance, it is reasonable to constrain components within an explicit vector variable w by definitions such as "w[2]=w[1]+w[3]". The semantic conventions which apply in this context are subtle: if i were a scalar variable, it would not be acceptable to introduce the definition

"w[2]=w[i]+w[3*i]"

since the possibility of circularity cannot be ruled out. In interpreting formulae, a subexpression of the form w[f(x,y,...,z)] is deemed to depend functionally upon the vector variable w unless the indexing expression f is parameterless. In the latter case either f defines an invalid index for w, or w[f(x,y,...,z)] is deemed to depend functionally on the *component* w[α] of the variable w which it represents, but not upon the variable w itself. The definition

"w[1] = g(x,y,...,z)"

is then valid subject to the formula g being neither functionally dependent upon w nor w[1].

Introducing vector variables has other consequences. Constructs for iteration are required for convenient indexing, and permit sequences of definitions of the form

" **for** I=2 **to** |dim(w)| **do** w[I] = I*w[I-1] ".

A capital letter is used here to emphasise that 'I' is a dummy variable, which is used purely for notational convenience. The scope of the variable I is confined

to the **for**-loop, and it can take only explicit scalar values. Such dummy variables can neither hold transient values, nor participate in persistent relations.

The above discussion indicates how complex data-types can be treated in a definitive notation. In generalisation, the difficulties are largely cosmetic; the syntax of declarations and assignments can easily become cumbersome. Similar considerations apply to the most appropriate way to enhance a definitive notation: by adjunction of user-defined data-types and operators to the underlying algebra.

In ARCA, there are two types of vector variable (**vert**, **col**), whose values are interpreted as vectors and permutations respectively, and a variable type (**diag**) for representing entire graphs. (User-defined operators are also available in ARCA, but their use is not illustrated. For more details, see [1].) The following program generates an ARCA diagram which represents the abstract graph of Fig.1 or Fig.2 according to whether the integer variable i is 0 or 1, and defines different planar realisations subject to the current values of v, D!1 and D!2 :

```
1    vert : v
2    int : i
3    'ab'-diag (abst vert , abst col 6) : D
4    a_D = {1,3,5}${2,4,6}@(1-2*i)
5    b_D = {1,2}${3,4}${5,6}
6    with int : I = 1,2 do
7            D!(2*I+2) = rot(D!2,I%3,v)
8            D!(2*I+1) = rot(D!1,I%3,v)
9    od
```

Line 3 is the declaration of an explicit **diag** variable D to represent a graph with 6 vertices, and edges of two colours: 'a' and 'b'. Both coordinate and incidence information for D are represented by "abstract vector variables" in the sense explained above. The vertices of D are represented by the variables D!1,...,D!6. Incidence information in ARCA is represented by (partial) permutations; the variables a_D (resp. b_D) represent the permutation of the vertex indices defined by the edges of colour 'a' (resp. 'b'). Lines 4-5 specify the incidences by formula assignment; '$' and '@" denote "superposition" and "inversion" of partial permutations. Lines 7-8 specify the coordinates of vertices by formula assignment: '**rot**' is a ternary operator in which the parameters are respectively

(point to be rotated , angle of rotation , centre of rotation).

The angle of rotation is represented by an integer modulus, and the parameter "2%3" denotes an angle of $4\pi/3$ radians.

Concluding remarks.

In a panel discussion on "Programming language issues for the 1980's" [9], Terry Winograd frequently refers to current interest in "profession-oriented languages": special purpose programming formalisms intended for a particular computing application, such as geology or auditing. In this context, Winograd cites the need for "a coherent theory and understanding and collection of tools for developing [such] languages". Definitive notations are conceived as a framework within which some profession-oriented notations for interactive use can perhaps be developed.

Over any universe of discourse, "human computer interaction" and "problem solving" can be viewed as separate concerns. A definitive notation is not intended as a formalism for problem solving per se, but rather as a mode of interaction. Its limitations are clear, but these appear to complement those of existing "problem solving systems" in some respects. Its potential merits, like those of spread-sheets, have as much to do with human psychology as computational novelty, and must be evaluated in these terms (cf.[7]).

Though the derivation of a definitive notation from a specification of the underlying algebra still requires some clarification, it is essentially a simple process, which could doubtless be automated. The main contention of this paper is that in many application areas an underlying algebra *can* be chosen so that the associated definitive notation is a convenient framework for interaction within which different semantic issues can be clearly separated, and independently addressed. To illustrate the latter point, it is appropriate to think of the underlying algebra as abstractly specified, and of "dialogue in the context of a particular concrete model". From this perspective, "changing directories in a file system" may be seen as "selecting another model". Assuming further that the definitive notation is interpreted with reference to the axioms: "formula manipulation" is "working over the abstract model", whilst "imposing a constraint" - as might be appropriate in solving a specific problem - is "adding a relation to the underlying algebra".

In the same panel discussion referenced above [9] - to the amusement of the audience! - Alan Kay is quoted as having said that "the most important programming language development in the last 5 years was VISICALC". I am glad to escape the credit for having originated this particular joke, but - many a truth is taken for jest.

Acknowledgements.

I am grateful to Martin Campbell-Kelly for many helpful suggestions on the form and content of this paper. I am also indebted to Robin Milner, Roy Dyckhoff and Derek Andrews for ideas and encouragement.

References.

[1] W.M.Beynon, A definition of the ARCA notation, Theory of Computation Report 54, University of Warwick, Sept. 1983.

[2] W.M.Beynon, ARCA - a notation for displaying and manipulating combinatorial diagrams, (submitted)

[3] Alan Borning, The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory, ACM Trans. Prog. Lang. and Sys., Vol.3(4), October 1981.

[4] Edsger Dijkstra, A Discipline of Programming, Prentice Hall, 1976.

[5] Christopher M. Hoffman and Michael J. O'Donnell, Programming with Equations, ACM Trans. Prog. Lang. and Sys., Vol.4(1), January 1982.

[6] M.McDonald, VisiCalc, Practical Computing, June 1980, p.64-67.

[7] Thomas P. Moran, An Applied Psychology of the User, ACM Computing Surveys, Vol.13 (1), March 1981.

[8] Douglas T. Ross, Origins of the APT Language for Automatically Programmed Tools, ACM SIGPLAN Notices Vol.13 (8), August 1978.

[9] L.A.Rowe.(ed.) Programming Language Issues for the 1980's, ACM SIGPLAN Notices, Vol.19(8), August 1984.

[10] I.Sutherland, Sketchpad: A Man-Machine Graphical Communication System, Ph.D. thesis, Dept. Electrical Engineering, M.I.T., Cambridge, Mass., 1963.

[11] Jeffrey D. Ullman, Principles of Database Systems, 2nd ed., Computer Science Press, 1982.

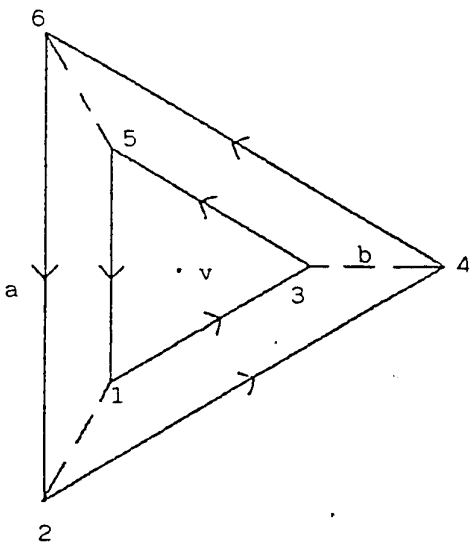

Fig.1: The cyclic group C_6

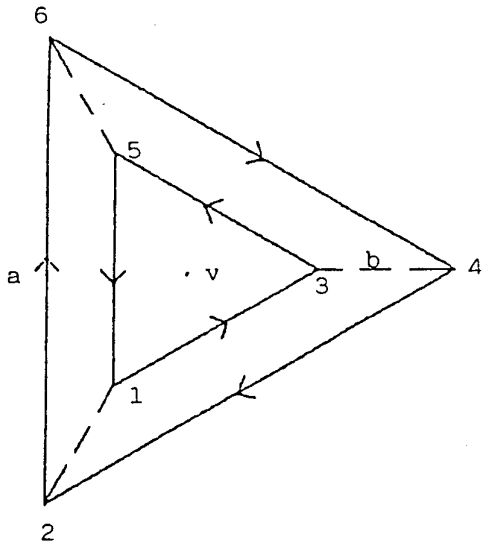

Fig.2: The symmetric group S_3.