Departmen. of Computer Science
University of Warwick
Coventry
CV4 7AL
England
EM00C101204.

DEfinitive NOTAtions: an interpreter for data abstraction

by
Samia Meziani
(B.Sc. Computer Science, Algiers)

A thesis submitted in part of the requirements for the awards of a Master's degree by research in Computer Science

University of Warwick
Department of Computer Science

December 1987

To my dearest mother, with all my love.

Acknowlegments

I wish to express my gratitude to my friends, and colleagues who helped me and encouraged me during the prepartion of this thesis.

In particular, many thanks to Edward Yung for his useful comments on **EDEN**, Dr. M. Campbell-Kelly for his advice on the way research should be conducted. Glwadys and Terrence Pugh for the family atmosphere, love, and attention they offered me.

I am also indebted to the British Council, and the Algerian Department for Higher Education for enabling me to carry out this research.

Above all, I am most grateful to my supervisor, Dr. W. M. Beynon, for the continuous help, encouragement, and constructive critisisms he has given me during my work.

Finally, I wish to thank profoundly my mother, and sisters for the love, patience and moral support they showed during my stay in Great Britain.

DEfinitive NOTAtions: an interpreter for data abstraction

by
Samia Meziani
(B.Sc. Computer Science, Algiers)

Computer Science Dept.
University of Warwick,
Coventry
CV4 7AL.

ABSTRACT

This thesis presents an overall picture of the tool to be developed for a generating interpreters for definitive notations that support data abstraction. The first chapter gives an introductory approach to the definitive paradigm by examining the potential of existing programming methodologies for interaction. The second chapter illustrates the definitive programming paradigm through some implemented systems. The third chapter concentrates on the issue of data abstraction in Computer Science, the need for formal models to specify it, and the role definitive programming has played so far in providing suitable tools for data abstraction. The fourth chapter covers the major ingredients of the interpreter such as the abstraction of definitive variables, the assignment of values to declared variables using formulae incorporating a wide range of operators, and the extraction of either the mode, or the value information of any definitive variable. The fifth chapter discusses the implementation of the present tool by translating definitive statements into some intermediate code where relationships between variables can be regularly maintained, and updated by a primitive definitive programming implementation tool. The sixth chapter reviews the work that has been done, and includes proposals for future research.

Keywords: Definitive notation, Algebraic specification, Data abstraction, Human-Computer interaction, Non-procedural programming, Software tools. interrogation.

Table of Contents

Chapter 1: Introduction 1

1.1 History and motivation 1

1.2 The components of a dialogue 3

1.3 Programming paradigms potential for interaction 9

1.2.1 The procedural paradigm 9

1.2.2 The applicative paradigm 11

1.2.3. The equational paradigm 14

1.2.4 The object-oriented paradigm 17

Chapter 2: The definitive programming paradigm 22

2.1 The spreadsheet 22

2.2 The unconventional desk calculator 24

2.3 ARCA: a definitive notation for combinatorial diagrams 26

2.4 DONALD: a definitive notation for line drawings 30

2.5 EDEN: an evaluator of definitive notations 32

2.5.1 Definitions 33

2.5.2 Functions 35

2.5.3 Actions 36

Chapter 3: Definitive principles for abstraction 37

3.1 The abstraction concept 37

3.2 Formal abstraction models 41

3.2.1 Abstract data types 41

3.2.2 Definitive notations 47

3.3 Algebraic abstraction methodology 53

Chapter 4: The design of DENOTA 57

4.1 Abstraction in DENOTA 60

4.1.1 Atomic moding 61

4.1.2 Abstract moding 62

4.1.3 Explicit moding 62

4.2 Value definitions 63

4.3 Variable interrogations 66

Chapter 5: The implementation of DENOTA 68

5.1 EDEN constraints 68

5.2 Implementing variable modes 70

5.3 Implementing variable values 72

Chapter 6: Conclusion 75

Bibliography 76

APPENDICES 84

Appendix A: Data models for abstraction 84

Appendix B: DENOTA - a user specification manual 86

CODES 92

CODE A:
Examples - a *List* based definitive notation definitive variables

CODE B:
DENOTA - the interpreter source listing

§1 Introduction

In this first chapter we are concerned with the importance of interactive and incremental computing. We aim to investigate the capabilities, and restrictions of traditional programming paradigms for conducting dialogues, and the reasons behind the birth of a hybrid programming methodology that meets the requirements of interactive computing.

1.1 History and motivation

Since the cost of computer hardware has decreased because of new semiconductor technologies, computer systems are being incorporated in more and more products. Extensive efforts have been expended in the design of formal methods for producing software. However, as long as these efforts were restricted to conventional Von Neumann computers, and conventional languages they fell far short of the ideal. Programs were fat and weak [Ba78], and traditional programming notations created unnecessary confusion in the way we think about programs. Moreover, the notion of a global state that may undergo complicated transformations throughout a computation has proved to be both intuitively and mathematically intractable. A growing number of research workers have realised these problems, and are seeking suitable means to help us reason about programs.

One result has been the development of new programming paradigms in lieu of ad-hoc notations. These are based on robust mathematical foundations, and with a better capacity for specific applications. Amongst commonly used programming styles are procedural/definitional, applicative/functional, equational, and object-oriented. Procedure-based notations are used to convey a list of commands, to be executed in some particular order, such that on completion of the commands, some required results are produced. Applicative notations are based on one of the most important notions of

mathematics, namely a function. A solution to a given problem is defined by the evaluation of an expression in terms of some basic functions. In that way, a set of functions defines a generic tool for finding the solutions to a class of problems, while an expression corresponds to the selection of a specific problem. They include sophisticated systems exploiting higher-order functions but similar principles can be seen in other programming paradigms based on relations which might come under the heading of equational languages. They cover both logic languages based on predicates and truth values, and constraint-based languages built upon algebraic equations. Object-oriented notations are the result of a recent design methodology that handles data abstraction by encapsulating data objects together with accessing functions into a single module.

Humans communicate with computers in several ways. One of the simplest of these is by pressing a button that starts, and stops, or in some way changes the action of the machine. On a higher plane, computer programmers communicate with their machines by writing sequences of commands that the computer is then asked to execute. As traditionally carried out, this form of communication is a monolog i.e. the programmer writes his commands, and then the machine executes them without further interaction. However, interactive communication, or dialogue, between the user and the computer has become increasingly important.

Interactive communication is two-way, each party providing feedback to the other, usually each indicating whether the last utterance of the other was understood, and supplying results, or progress indications for any action that was requested.

The aforementioned programming language models have not been designed with interactive computing in mind. They all focus on computing the answers to a particular problem, and involve respective strategies to do so. As a result, a new approach to programming based on sequences of definitions has been proposed which borrows ideas from previously cited models; relevant not only to dialogue, but also reasoning as a form of interaction between the user, and the computer.

1.2 The components of a dialogue

The form and structure of computer programs vary widely, there being many different notations available. However, we note that the stress in modern programming methodologies previously cited has been on problem solving.

People who are problem-solvers, whether physicists, physicians, programmers, or ship's navigators expect to be provided with certain basic tools to communicate with the computer. Depending on the profession, these may include microscopes, slide rules, telephones, drawing pens, keyboards, and other equipment. Problem-solvers directly use their instruments to tell the computer what measurements to make and what calculations to perform. The choice of medium is wide since the message can be in textual, pictorial, or graphical form, or it may be so simple as the unlocking of the keyboard to enable the user to communicate. For this, research on interactive systems has focused on developing techniques, and programming notations to help the user and the computer understand each other. In other words, the effort has been on passing a sequence of consistent, and meaningful messages in both directions.

Regardless of the purpose of the conversation, we find the following elements almost invariably involved.

Firstly, a message is sent from the computer to the user, by way of a communications terminal. This message can be in any meaningful form to the user.

Secondly, the user responds to the machine-generated message. Again the choice of communication mode is broad. For instance, the user may type in a text message or use a mouse.

Thirdly, the computer analyses the user's response. This is where the suitability of the programming notation comes into play, for the sophistication (perhaps the entire success of the conversation) depends, in the next step, on the computer's ability to "speak" meaningfully to the user at a nontrivial level of discourse. Restriction to "yes/no"

or other multiple choice replies necessarily limits the effectiveness of a conversation. The computer is programmed in such a way that it is able to distinguish between a wide variety of responses, and even to recognise subtle differences between actual, and anticipated responses.

The fourth step encompasses the processing of responses, in which we include making the decision on what to do next. Processing may include storing, performing some elaborate computations, or constructing a reply. The branching decision is that of deciding which conversational step to operate next. However, the decision of branching is made, not just on the last user's response, but on the basis of an analysis of all the previous messages. For instance, an architect designing a building may signify that he/she desires to add a new line, or that he/she wishes the existing drawing of the building to be rotated on the viewing screen, or that he/she wants a photograph made of it, or that he/she would like to change a previous line. The conversational program makes it easy for the user to specify her/his intent which subsequently enables the computer to recognise, and carry out the decision. Once the computer completes, and makes its decision, a new message-response-process cycle begins.

When interacting one is unconsiously considering two distinctive ingredients of the dialogue: its *semantics* ("conceptual content" [Foy87]) which encapsulates all "computational elements" illustrated by some knowledge representation together with proper computational processes , and its *mechanics* ("interaction techniques" [Foy87]), which involves the "communication interface" itself described by all the communication processes that go on concerning user interaction with the computer.

Traditionally, conventional programming notations for problem-solving have been primarily concerned with the mechanics of a dialogue in the sense that one matter of interest has been the transmission of the message in a manner which is comfortable to the user e.g. menus, windows, mouse, colour graphics, or detailed prompts. And even when semantic issues are addressed, little emphasis has been placed upon capturing the

semantics of the dialogue at some intermediate stage in order to summarise what has been achieved thus far.

On the whole, the criteria examined in the design of a good interface have been appropriate communication modalities, and computer responsiveness to user vagueness, imprecision, and error. In other terms, the computer has been considered to be a tool to extend the user's memory, and reasoning power. The only concern has been to protect the user from low level issues, in order to present an understandable and sympathetic interface.

The proliferation of interactive computing has generated progressively more intricate styles of interaction between the user and the computer. As a result, the concept of "user interface management system" was introduced in order to help the design of interactive applications. When using conventional notations, and user interface management systems there is a transparent separation between the semantics of the dialogue from its mechanics. The dialogue is supported by a series of rules that would take care of isssues such as picture editing, window management, or event queue manipulation, and leave the user completely unaware of the details of the underlying system. A snapshot of an interactive environment is only considered at the beginning, and at the end of a dialogue when some control over a graphical image, for example, has been reached.

The problem of computing answers to given questions is not the only application of computers. There is a distinct difference between a person who wants to "draw" a picture, and one who needs to "get" a picture, or between the "original composer" of a piece of music, and the "pianist" of that particular "tune". When painting scenery, a painter is progressively building a decor by adding different colours, and items whenever she/he wishes. In that way, the painter is interactively modifying the "meaning", or semantics of the picture. In the same manner, a composer plays few music notes, listens to them, changes some of them, then plays few more until a whole concerto is written.

In such instances, one is interested in an interactive dialogue whose semantics can be relevant at every stage. In that case, the user is concerned with the way the dialogue can be suspended after each operation to examine what has been achieved so far, and restart again.

User-computer interaction is a very subtle issue. Hitherto, users have generally been more interested in the mechanics of the dialogue i.e. the manner in which it is conducted, rather than its semantics i.e. the computation itself. It is helpful at this point to consider the semantics of an interaction more critically.

Traditionally, the initiative for the interaction has been with the user. In general, the machine has no knowledge of any "plan" or structure to which a sequence of commands corresponds, and has no way of applying such knowledge. Since it has no information of the overall goals of the user, the computer is constrained to obey dumbly. It cannot check whether particular actions are furthering or hindering progress towards a goal. In that sense the nature of interaction generally considered in Computer Science circles has been a "passive" one i.e. master/slave relationship, since the computer does not in fact participate in the dialogue. The principal agent in this form of interaction is the user who edits a program which is then used directly to determine the required solution leaving the computer totally "dumb". By this, we understand that the computer does not take any initiatives in the management of the semantics of the dialogue.

Nonetheless, some attempts have been made to allow the computer to decide whenever the mechanics of the dialogue is to be altered. For instance, adaptive user interface management systems deal with situations where a sequence of responses is changed to suit the user's level of experience such as the suppression of menus, or the modification of the speed of typing. Nevertheless, one might consider a more "active" form of interaction where the user, and the computer share the total management of both ingredients of the dialogue. In that case, the two partners exploit a common environment which includes all the necessary information related to the dialogue. This requires a

suitable formal medium in which to express the semantics of the suspended dialogue. What would be the requirements of such a medium?

During an interactive dialogue a user should at any time be able to find out what the current state of the dialogue is, and how to proceed further with the dialogue i.e where am I? where do I go next? In other words, he/she should be capable of reviewing what has and what has not been computed. Another question is to know who is responsible for supplying this information, and what it really means.

An example of what the current state of dialogue might be, is illustrated by text-formatters. These can be classified into two categories : The first class of text-formatters is known as **What You See Is What You Get**, and would display on the screen the content of your text as it will be printed. The second type of text-formatters is described as **You Asked For It And Now You've Got It**, and would process the text according to some included formatting commands. The latter is also known as a format code driven formatter, and does not correspond to an interactive application since the text is written, read and then executed. However, **WYSIWYG** formatters constantly allow the user to know what the state of dialogue is. In fact, it is transparently visible on the screen, and is changed automatically. In other terms, the text in its intermediate final shape is seen on the screen which allows the user to evaluate what has been done and what still needs to be achieved. Thus, the program of **WYSIWYG** formatters is responsible for remembering and displaying the current state of dialogue which make them very popular with micro users who want to be relieved of keeping record of some sequence of formatting commands, and their respective effect on the text. In the case of **YAFIANYGI** formatters, the current state of dialogue is the responsibility of the user. He/she has to record which formatting commands has been used, and what current state of dialogue to expect. This task is difficult, and is usually carried out by computer scientists who are familiar with formatting commands, and are interested in good quality output.

Additional examples of software tools for interactive applications are spreadsheets, relational data-base query systems, or file manipulators such as text-formatters. These aim to describe and consult a conceptual model of the real world with the help of a computer. An interaction dealing with files, for example, would designate every object by a variable viz. the filename, and a value viz. the content of the respective file. Another interactive application dealing with telephone communications would describe each connection by its phone number, and its number of units. However, such basic interactions put aside a great deal of relevant information such as the way objects are correlated i.e O is the object of the source file S, 3193 and 2424 are extension numbers of the same line, and so on.

These examples illustrate common problems of interaction when a user wants to resume a dialogue about a particular object after a while. Information is needed to refer to things which have been calculated, and to describe the abstractions which have been made. The conceptual model of a particular interactive application is then used to determine the current state of dialogue in order to progress step by step towards some satisfactory solution.

The approach to represent the semantics of a dialogue which we adopt considers the user's conceptual model to be a set of variables (e.g. filenames) together with

(a) a set of values

(b) a set of functional relationships between those variables [Be85].

The set of current values is usually refered to as *transient values* since variables tend to have changing values during the course of a particular dialogue. The set of functional relationships between variables will be described as *persistent* by the fact that although values of variables are unstable, the relationships between them are more or less fixed. However, both transient values and persistent relationships should be easily modified to allow a flexible mode of dialogue. The conventional "edit-compile-execute" mode is then replaced by a "define/redefine-evaluate-print" style ("incremental

programming"), based on the ever present transient values and declarations or persistent relationships.

When thinking about progressing in the dialogue, a summary of the conceptual model has to be accomplished and references have to be made. The dialogue is then broken into smaller pieces which are easier to deal with. We have proposed a simple representation for the semantics of a suspended dialogue using functional relationships. It is of interest to examine the potential of existing programming paradigms for addressing this problem. In our view, although existing programming paradigms may be used to support a good dialogue, they are not well adapted for describing the suspension of a dialogue according to the simple model we have proposed above.

1.3 Programming paradigms potential for interaction

The way in which a system based on an existing programming language supplies the current state of dialogue is by a suspension of the system, and an output of some intermediate results on what has been done so far. Each programming method has its own characteristics and would have its unique way to convey transient values, and specify persistent relationships in the context of some incremental application.

1.3.1 The procedural paradigm

Procedural languages consist of a sequence of iterations through a number of steps or commands towards a solution to a particular problem. The simplest commands such as assignment, and branching are also generalised to form procedures and functions as an abstraction mechanism. The procedural languages are based on the Von Neumann architecture, and form the mainstream of programming practice. Amongst them are **ALGOL, FORTRAN, PASCAL, PL/1**, and **C**. In procedural notations, a summary of the state of dialogue would be equivalent to taking snapshots of program variables,

control stacks, program counter, activation records, etc.

It is worth mentioning that procedural programming paradigms have the capability to update transient values, but are badly adapted for dealing with persistent relationships. The programmer would have to analyse the different snapshots of the application conceptual model to tell where he/she stands, and what point the program has reached so far. This task is generally very difficult, and can only be performed by a skilled programmer. Although variables are linked together by means of persistent relationships, the procedural paradigm does not infer actions upon persistent relationships whenever associated transient values are changed. For instance, if a file O is the object file of S; then O needs to be updated whenever S is updated. It is the duty of the programmer to recompile S in order to update O. The user is constrained to keep record of all persistent relationships, and modify them when necessary.

In order to proceed further, the programmer reviews what has been achieved, and references objects on which some computation has already been performed. This referencing problem is not easily accomplished in a procedural environment. The user usually adopts variable names, and pointers to reference different objects. However, this can be a complicated task since variable names may not refer to the objects which the user has in mind. Similarly, the use of pointers can lead to complex, and confusing expressions which might miss the user intended target. In general, the suspension of procedures necessitates debugging techniques that only an expert programmer can use effectively to restart the dialogue where it has been suspended. Using the current state, transitions to a new state are accomplished by redefining the transient values of variables which are the only components of the conceptual model in procedural environments.

The limitations of the aforementioned programming paradigm to deal with interactive applications by capturing a satisfactory image of the conceptual model i.e. the values of all used variables, and the relationships between them, is mainly due to the fact that a procedure is a recipe to solve a problem. The variables used in a procedural

paradigm are considered to refer to independent objects whose corporate behaviour is not examined. These objects are exploited as convenient tools to reach the aimed target. Many Computer Science research workers realised that most problems encountered in procedural programming were generated by the Von Neumann methodology [GHT84] where the programmer has to think of software as means of making a machine solve a problem, rather than a problem solution that can be executed by a machine. Moreover, conventional procedural notations violate the basic rules of mathematical notation. At the core of each traditional procedural language is the assignment statement, which destructively replaces the value of a variable with some new value, greatly complicating attempts to reason carefully about such programs without the help of persistent relationships amongst variables. (The important role that assertions play in disciplined procedural programming is worth mentioning at this point.)

The restrictions met with procedural programming led to the birth of an alternative, simple, and mathematically based programming paradigm.

1.3.2 The applicative paradigm

Applicative programming notations have been studied for many years as theoretical models of "real" programming languages. Now, they have become become practically valuable with the continuing decline in the cost of hardware (Fig. 1.1). Moreover, they offer simpler, clearer, and more concise programs than procedural notations (Fig. 1.2).

Amongst applicative programming notations are **SASL**, **MIRANDA**, **SUGAR**, and **HOPE** [GHT84]. The solution to a particular problem would be obtained after evaluation of a corresponding function. An applicative program would then be an expression in the formal mathematical sense since it owns the fundamental property known as *referential transparency* according to which

(1) an expression's only meaning is its value, and

(2) every occurence of a given expression always denotes the same value.

By virtue of this property, applicative notations conform to the fundamental axiom of mathematics, namely, the substitutivity of equals. Applicative programs can then be analysed statically, in terms of their texts, without reference to the temporal ordering of the operations they induce. This important feature of applicative programming relieves the programmer of all responsability for the sequencing of steps in program execution. Therefore the concept of a state is not relevant in functional programming.

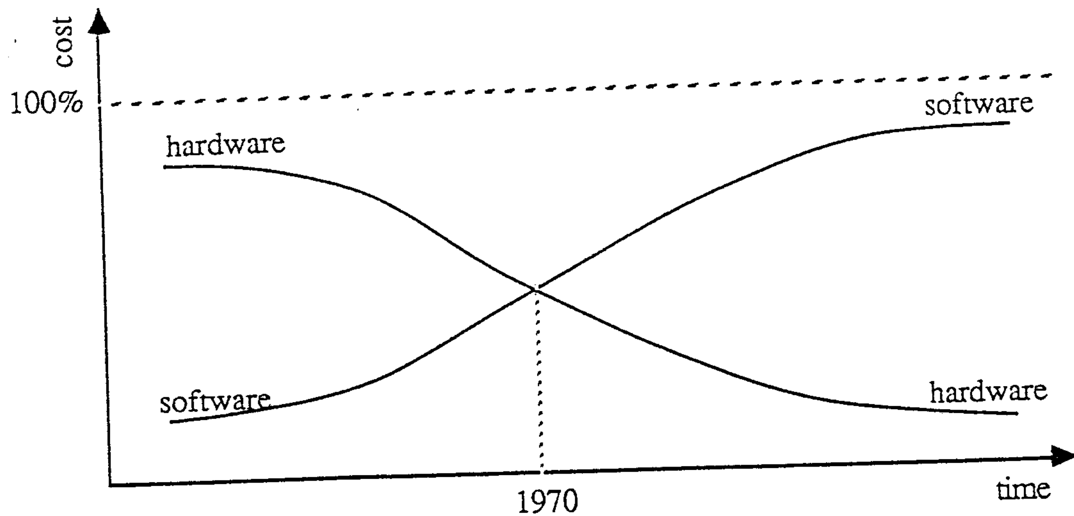

Fig. 1.1 Relative costs of hardware and software components.

As far as interactive applications are concerned, a functional program evaluation would be equivalent to the translation of an expression into a tree which is then reduced. A restoration of the state of dialogue after interruption would be a snapshot of a suspended evaluation of a function. This is equivalent to keeping record of all functions evaluated, and state of the function in process of evaluation. The incremental updating of a functional state of dialogue is not feasible since there are no defined relations between the initial program, and the intermediate evaluation tree. In addition, the user has no notion of a state since pure functional programming, prevents the user from analysing the way in which functions are evaluated. We must underline the fact that functional

programming notations are not oriented towards incremental programming.

```
sort[ ]=[ ]
sort(a:x)=sort[b; b<-x; b<a]
        ++ [a] ++
        sort[b; b<-x; b>=a]                    (a)


procedure quicksort;
procedure sort(l,r:index);
var i,j:index; x,w:item;

begin i:= l;j:=r;
x:= a[(l+r)div 2];

repeat
while a[i].key < x.key do i := i+1;
while x.key < a[j].key do j := j-1;
if i<= j then
          begin w := a[i];
               a[i] := a[j];
               a[j] := w;
               i := i+1;
               j := j-1;
          end
until i> j;
if l < j then sort (l,j);
if i < r then sort (i,r);
end;
begin sort (1,n)                               (b)
end{quicksort}
```

Fig. 1.2 The algorithm "Quicksort"(a) in SASL, (b) in Pascal.

In connection with the above discussion, it is appropriate to comment upon recent work by Thompson on applicative programming methodology for interaction [Th86]. Thompson begins by considering interactions of the form

input ⟶ (*input* , *ouput*)

where an interaction may read a portion of the *input*, and pass the rest to another interaction. He then motivates the introduction of encoded "state" information. For instance, if we consider interaction to get an integer from an input stream; we have no state information at the start of the interaction. Yet at its (successful) termination, we have an integer in the state; this he models by a parametric type definition of the form

interact * ** == (*input* , ** , *output*).

In addition to introducing encoded "state" information, Thompson also uses control structures to build interactions in an action oriented way to help programmers understand the full details of function implementations. As a result, the programmer can specify the order in which interactions occur such that the entire dialogue can be reduced into smaller more manageable states which are easier to understand, and manipulate. An example of two interactions where the invocation of the second waits for the completion of the first, could be defined by the following functional statements:

```
seq   ::      interact * ** → interact ** **** → * **
seq   ::      inter1 inter2 ×
      =       make-output out1 (inter2 (rest, st) )
   where      (rest, st, out1) = inter1 ×                  †
```

The effective use of encoded "states" and "actions" within the purely declaractive MIRANDA environment makes crucial use of lazy evaluation, whereby the arguments to a function are not evaluated until they are needed by the calling function. Since evaluation is "demand-driven", output can be given incrementally. This new approach to applicative programming oriented towards interaction promises a more disciplined methodology to capture the state of dialogue i.e. where am I/ where to go next?

1.3.3 The equational paradigm

For the pupose of this discussion, equational programming notations encompass logical languages based on first-order predicate calculus such as **PROLOG, LUCID**, and the **RED languages** [HO82], together with constraint-based notations based on algebraic expressions such as **JUNO, BANZAI**, and **Metafont-style languages** [Ne85].

† *The action make-output pushes a string onto the output stream, and seq combines two interactions in one.*

Equations are sufficiently powerful to describe a large number of computations. An equational program is usually given as a set of assertions, then the logical consequences of the program are all those additional which must be true whenever the assertions of the program are true. The assertions are typically represented as a systems of equations ("rules") of the form **L** = **R** in which **R** is a simpler expression equivalent to **L**. An equational program **D** then comprises a sequence of equations, followed by the question "What is E_0?". This program is interpreted by systematically applying the rules until a relation $E_0 = E_i$ such that E_i cannot be further simplified.

Constraint-based programming notations are based on the calculus of guarded commands defined as predicate transformers [Di76]. A constraint-based program can be viewed as a collection of predicate transformers acting upon the predicates of a mathematical theory such as geometry e.g. **JUNO** [Ne85]. A typical constraint-based command then has the form :

LET *variables* | *constraints* IN *command* END

where *command* is executed according to *constraints* to determine the values of the *variables* (this is analogous to a guarded command). A typical example in **Juno** is the construction of an equilateral triangle on a given line which uses Euclidean predicates such as "circle", "center", and "point" (Fig. 1.3).

How effective are equational methods for describing the state of a suspended dialogue?

Like functional programming notations, equational notations are declarative i.e. referentially transparent, so that state information must be supplied by a system of equations. It is clear that equational programming notations implicitly specify functional relationships between different objects, and these can model "persistent relationships" in the sense introduced above. In practice, systems based on these principles provide limited scope for manipulating "transient values" subject to these relationships.

However, constraint-based specifications may be ambiguous, since the semantics of guarded commands is non-deterministic i.e. different solutions can be given to a problem depending on the way in which constraints are interpreted. The fact that the user has control over the way a solution to a problem is to be expressed may also mean that she/he needs to have a premature idea of the nature of the expected solution. For instance, it may be important for the user to know that two circles can intersect in at most two points.

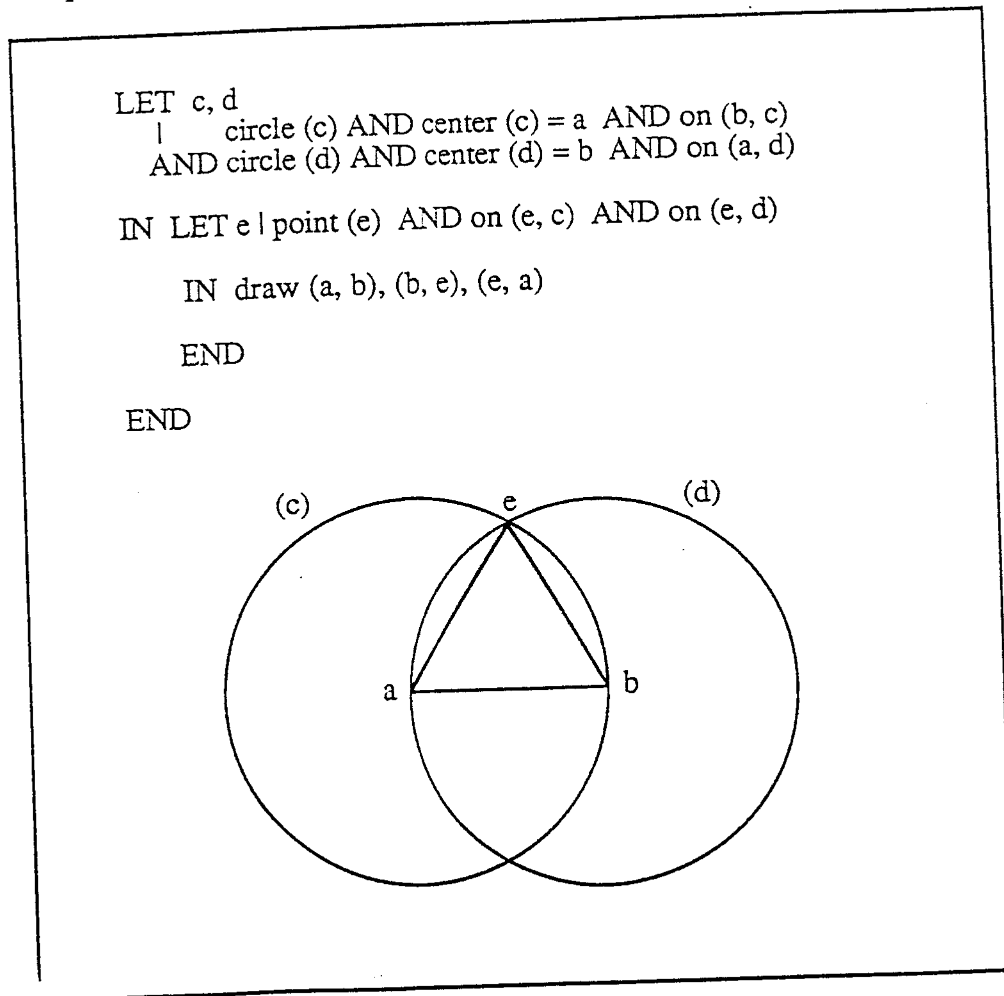

Fig. 1.3
Construction of an equilateral triangle on a given line in Juno

In the context of an equational programming paradigm, the current state of dialogue would be defined by the set of relations so far defined. Modifications of the current state would be achieved by defining new equations. Within this paradigm, the use of constraints leads to serious problems in describing and referencing the state of a

suspended computation.

In constraint-based programming, as with procedural, and functional methodologies, it is difficult to monitor a suspension of an interactive dialogue.

1.3.4 The object-oriented paradigm

Early work in structured programming concentrated on bringing clarity, and discipline to problem control flow i.e. the ordering of functions that manipulate data. One subsequent development was a new programming paradigm bringing modularity to data handling as well as procedures : object-oriented programming [Am86]. In functional modularisation, the key idea is to implement higher level functions in terms of the lower level ones. In the object-oriented programming paradigm data objects are encapsulated together with all accessing functions into one single module. No longer are functions, and data independent entities, but inseparably enclosed in an *object*. The data contained in an object is invisible to the object's outside world. A so called "client" can only access the data indirectly by invocation of an access function viz. operation (Fig. 1.4).

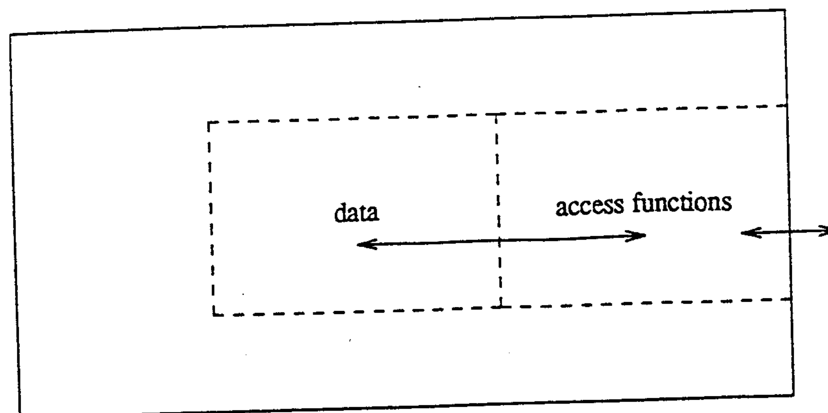

Fig. 1.4 Diagram of an object

Using objects, as will become clear, provides a natural approach to programming because it reflects the way in which humans interact with their world. D. L. Parnas described the first object-oriented programming notation where a module was designed

to group a small set of software design decisions in a module [Wi84]. Such decisions generally defined one or few data structures, and the ways they could be manipulated. By this definition, a module is not a single subroutine or data structure, but rather an interrelated collection of procedures, and data.

Objects which remain inscrutable to all but their creator, are said to be logically encapsulated because from the user's point of view, they are black boxes without internal structure. Objects have specific visible properties, and certain uses, but their inner workings are of little concern. The object-oriented programming paradigm provides a unified view in which a program consists of a network of objects that use one another to perform the desired application (Fig. 1.5).

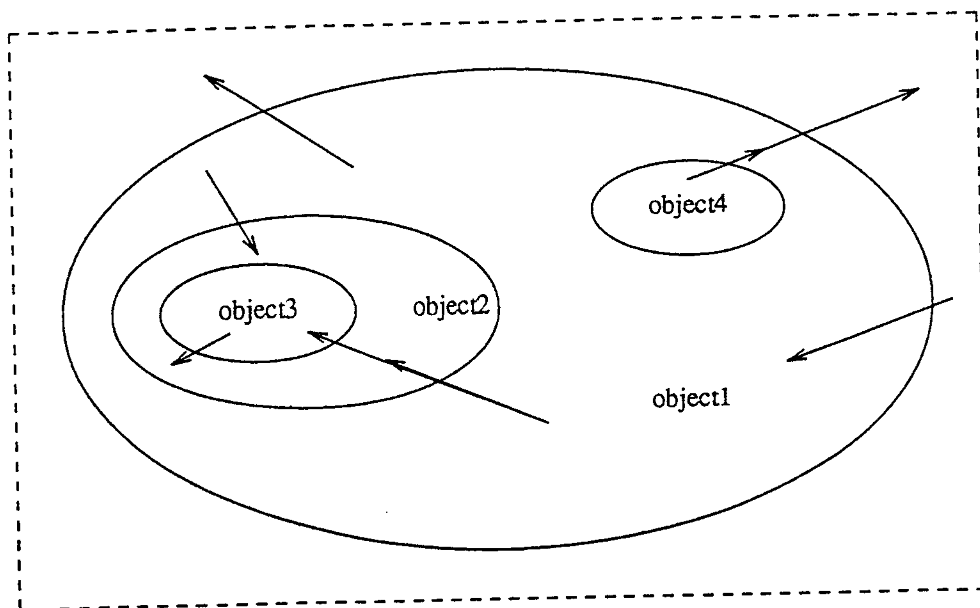

Fig. 1.5 Import/Export transactions of data between objects

Amongst the programming notations which adopted the object-oriented paradigm are **SIMULA** ,**CLU**, and **SMALLTALK.**

An object can also be "classified", "abstracted", "categorised", "made-generic", or "typed". An object therefore belongs to some abstract (semantic) class. Implicitly

associeted with an abstract class is a set of "allowable" operations that may be performed on the object of that given class. We may illustrate the object-oriented approach to programming with a simple example from taxonomy. We specify **horse**, and **dog** as the classes (or types) *EQUUS CABALLUS*, and *CANIS FAMILIARIS* (Fig. 1.6).

type *horse* is *EQUUS CABALLUS*;

type *dog* is *CANIS FAMILIARIS*;

we can define horse and dog objects by

Blackarrow : *horse*;

Lassie : *dog*;

Fig. 1.6 A An informal object-oriented program sample.

Associated with the objects **Blackarrow**, and **Lassie** are certain (implicit) operations. We can mate two objects of the type *EQUUS CABALLUS*, and mate two objects of type *CANNIS FAMILIARIS*. Nonetheless, it is not permitted to perform the operation **MATE (Lightning,Lassie)** because the two objects belong to different classes.

Objects can be elementary such as whole numbers, characters, and logical values, or standard such as stacks, queues, and files which are built of elementary objects, and are context-independent. Finally, application objects are tailor-made according to specific needs, and without the intention of later using this part of software for other applications to come. It is worth mentioning that a procedural program, as a whole is nothing else but an application object. As a result, the object-oriented programming paradigm is mainly procedurally based, where its components communicate information under the control of some defined protocols e.g. message sending in **SMALLTALK** [KP 86].

However, we must underline that object-oriented programming presents some

unsatisfactory features when dealing with interaction. An object is considered to be a black box whose definition is stable. This can lead to severe inflexibility. For instance, if we consider defining a type **shape** for use in a graphics system. We expect for the moment that the system has to support **circles**, **triangles**, and **squares** that will belong to a class **shape**. The problem comes when the programmer wants to define a new shape [Ne85]. At that point, each operation on a shape must be examined, and possibly modified, which involves great skill, and potentially introduces bugs into the code handling other objects. In order to avoid this problem, the programmer is constrained to examine the distinction between the general properties of a shape (it has a colour, it can be drawn, ect.), and the properties of an individual shape (a circle is a shape that has a radius, is drawn by circle drawing functions, etc.).

During the course of a dialogue, the programmer can resume the conceptual model related to her/his application using special purpose modules to monitor the implicit relationships between different objects. This can be difficult to realise, but is feasible. On the other hand, references to the current state of dialogue in to transit towards a new state cannot be adequately expressed in an object-oriented programming environment since each object usually relies on functions for proper setup and operations, which themselves may be linked to other objects and functions. When a suspension of the dialogue occurs, one typically has to stop testing functions until a substantial part of the program has been completed, otherwise the state of the dialogue would have to be resumed to its initial state, and the functions restarted over again.

The most commonly used programming paradigm is based on "*Decide what you want to do, and then write the procedures for doing it*". Procedural programming is mainly oriented towards problem-solving, and has no means to conceptually reconstruct the functional relationships between variables in a dialogue. The functional programming paradigm is based on "*Describe your solution by a mathematical expression, and then evaluate it*". Functional notations emphasise the relationships between variables, but do

not consider the concept of state to which references can be made. The equational programming paradigm is based on "*Here are the equations my solution should satisfy, find it*". Equational notations have a great potential for describing persistent relationships between variables, but present some ambiguity in monitoring constraints i.e. the choice of equations, and constraints is non-deterministic. The object-oriented programming paradigm is based on "*Decide which types you want your solutions to belong to, and then define them with the representation hidden, and all necessary operations provided*". Object-oriented notations provide ways of specifying the persistent relationships e.g. the programmer could write special purpose protocols to be invoked whenever some objects are activated to capture the relation between them. Nevertheless, the semantics of an object might be confusing since it is mainly defined by its operations. An object can be viewed differently from one user to another which makes the interpretation of the state of dialogue depend on the semantics of the provided operations.

Although the aforementioned programming paradigms can be used in an interactive environment, they need to be improved in order to capture in a satisfactory way the state of dialogue. As a result, a simple, and clear programming paradigm based on definitions has been proposed whose primary goal is to suit interactive applications.

§2 The definitive programming paradigm

This chapter explores some theoretical, and practical features of the definitive programming paradigm. Definitive notations were developed from the spreadsheet principle to become a simple, and concise programming language based on a sequence of definitions for interactions [Be85]. Basically, a definitive notation involves an underlying algebra of concrete data types (i.e. they cannot be evaluated further, not to be mistaken with abstract data types) together with a set of related operators. The underlying algebra is chosen to reflect a particular universe of discourse where elements of different types are represented by variables. A "definitive program" would consists of assigning values to variables, either explicitly or implicitly by means of formulae, and referencing objects through their variable names. Semantically, a formula assignement :

a = f (b, c, d, ..., z);

where **f** is a formula over the underlying algebra in the variables **b, c, d**, ..., **z**, states that (until redefinition happens) the value of the variable **a** is to be determined when required by evaluating the formula **f (b, c, d, ..., z)** over the underlying algebra.

Using the definitive programming paradigm, a typical program would be a dialogue between the user, and the computer over some universe of discourse. Definitive notations in that way provide suitable means to restart the dialogue by keeping record of the transient values of variables, and the persistent relationships between them which together constitute the state of dialogue.

Having framed the principal theory behind the definitive programming paradigm, we will discuss in detail some implemented systems which share the same concepts.

2.1 The spreadsheet

A familiar instance of definitive programming is the dynamic spreadsheet e.g. MULTIPLAN, VISICALC, MINDWARE, and many others [Ha84]. A spreadsheet is

a collection of named cells, each containing a numeric or textual value (Fig. 2.1). A cell's value may be simply a literal number, or character string put there explicitly by the user; alternatively, it may be given by a formula (also specified by the user) establishing its value in terms of other cells' values, which may themselves be either literals or formulae.

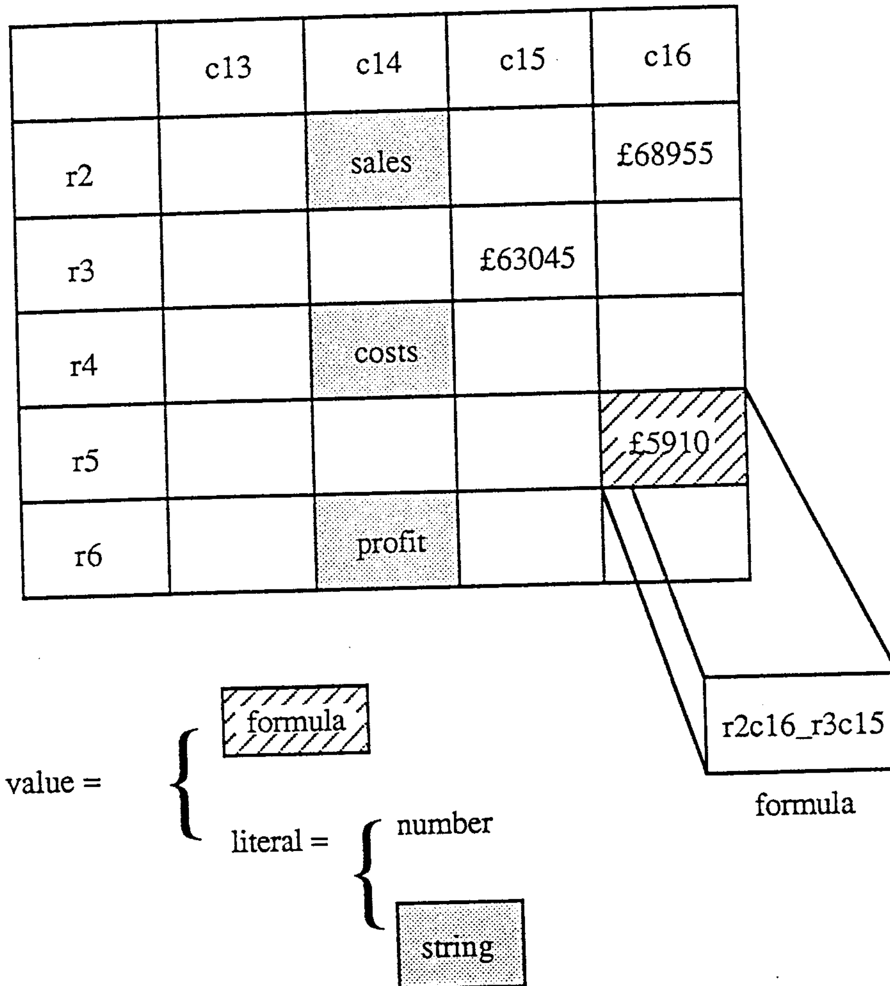

Fig. 2.1 The dynamic spreadsheet

The spreadsheet user is free to concentrate on defining the cells' values and the functional relationships between cells which is definitive programming in a nutshell. All responsibilities for maintaining these relationships are borne by the underlying system. In particular, the storage in which cells, values, and formulae are concretely represented is managed entirely by the system. Whenever any cell's value is altered, the system

deduces which other cells' values need to be recalculated, and carries out the necessary computations in a proper sequence with no direction from the user.

The enormous success of the spreadsheet model is a proof of the appeal of its generalisation, and abstractness, which is so convincing that most spreadsheet users do not even called "programming". Nevertheless, recent work in programming methodology oriented towards interaction, gives strong hints that much more powerful styles of programming can be generated following the spreadsheet principles [Be85]. We believe that definitive programming is a powerful generalisation of the spreadsheet concept oriented towards more complex interactive applications.

2.2 The unconventional desk calculator

The Unconventional Desk Calculator (UDC) is essentially based on the spreadsheet concept. It is a generalisation of the simplest type of desk calculator traditionally implemented on computer systems, in the fact that its primary statements are the assignment and interrogation of variables. Nevertheless, the semantics of primary statements differs greatly from traditional desk calculators in the sense that a UDC statement assigns a formula rather than a value to a particular variable. Thus, the action

x = y + z implicitly defines the value of the variable **x** as being the value of the expression **y+z**, where the values of the variables **y** and **z** are not considered separately i.e. if the value of **y** is not yet defined, it does not imply that **x** is undefined. In UDC, the value of a variable is instantianted at the time of its interrogation unlike conventional desk calculator which instantiate variable values at the moment of their definition. The mode of definition followed by the UDC can be illustrated by considering :

(1) **a = 2**

(2) **b = c+d**

(3) **e = 2*b+f**

(4) **d = 1**

(5) **f = 2*a**

(6) **a = 7**

(7) **?{f}**

Initially, all variables are considered to be undefined. It is also worth mentioning that a formula assignment is not automatically equivalent to a value assignement since dependent variables can be re-defined after the definition of a variable has been made e.g. **a**, and **f**. In the above example, the value of **f** will be **14**, since definition (6) re-defines variable **a**. The variables **b**, and **e** are respectively defined by formulae "**c+d**", and "**2*b+f**", but do not have defined values.

The unconventional desk calculator binds variables to definitions, not values i.e. as a definitive programming paradigm. In such a system, there are essentially two types of actions : defining, and evaluating variables where the definitions can be either a formula, or a value assignment. The **UDC** is based on the definitive programming paradigm where the underlying algebra includes only integer numbers. The evaluation consists of returning the value of a variable in the current context, rather than its value in the context in which it was defined. However, it is important to emphasise that the semantics of evaluation places some restrictions on primary statements such as circular definitions of the form **var1 = var1**, or **var1 = var2**, and **var2 =var1** which would lead to an infinitely recursing instantiation when evaluated. In summary, the unconventional desk calculator is a simple abstraction of the spreadsheet principle which can be easily implemented to test the basic features, and problems encountered with definitive programming such as keeping record of the relationships between variables, and changing them whenever is necessary.

2.3 ARCA : a definitive notation for combinatorial diagrams

The definitive programming paradigm approach for interaction was first represented by ARCA, named after Arthur Cayley, to display and manipulate combinatorial diagrams [Be86]. Its underlying algebra consists of three data types namely "diagram", "vertex", and "colour", all built from the basic data type "integer". A diagram of type **diag** is composed of coordinates and incidences respectively described by a vertex array, and a colour list. In other terms, a diagram is a sort whose components are an array of elements of sort **vert**, and a list of elements of sort **col**. In addition, both sorts **vert**, and **col** are themselves arrays of elements of sort **int**. Each integer value has an associated "modulus" which is a non-negative number. In that way, an integer of "modulus" **m** will be called an **m-int** which represents a residue modulo **m**. In general, integer constants are considered as **0-ints** since they are integer in the traditional sense. On the other hand, integers which denote a number of geometrical units of length are regarded as **1-ints**. The ARCA notation includes a wide range of operators that manipulate variables of the algebraic data types. Amongst them are arithmetic, vector, geometric, permutation operators, and operators for the amalgamation of diagrams.

(a) Aritmetic operators

The ARCA notation provides the user with commonly known arithmetic operations such as the addition, substraction, multiplication, and mod-reduction respectively denoted by the prefix binary operator +, -, *, and %.

(b) Vector operators

A vertex is a structure defined as an array of integer elements. for this reason, the vector operations are very similar to the arithmetic ones. Amongst them are the vector addition and inner product, respectively denoted by + : **vert** × **vert** → **vert**, and

< , > : **vert** × **vert** → **int**, and the scalar multiplication of a vector by an integer number denoted by **.** : **int** × **vert** → **vert**.

(c) Permutation operators

A colour sort is a structure defined as a list of integer elements used to give different interpretation of a diagram i.e. it may represent different groups depending on the permutation of its colours. In the class of colour operators we find the composition and exponentiation of permutations, respectively denoted by **.** : **col** × **col** → **col**, and **@** : **col** × **int** → **col**. In addition to these two operators, ARCA presents the programmer with an inverse of a partial permutation operator designated by ¯ : **col** → **col**.

(d) Geometric operators

This class of operators includes the mapping, the rotation, and reflection of a vector in the plane. Scaling of a vector is described by a scalar multiplication of a vector by a rational number, denoted by **.** : **int** × **int** × **vert** → **vert**. The rotation of a vector of a particular angle, around a centre of rotation, is denoted by **rot(, ,)** : **vert** × **int** × **vert**. The reflection of a vertex on some axis is designated by **ref (, ,)** : **vert** × **int** × **vert** → **vert**.

(e) Amalgamation operators

Using existing structures we can define new ones by means of amalgamation operators such as the join of two colours, and the smash of two vertices. The "smash" operator defines a vertex by listing its components explicitly. It is denoted by ** : **vert** × **vert** → **vert**, and describes the new vertex as composed of the components of the first argument follwed by the components of the second argument. The "join" operator defines a new colour in terms of two previously known colours by means of a formula. It is denoted by **::** : **col** × **col** → **col**.

In ARCA, the user has two different ways to reference the same object. In fact, the "moding" facility was first introduced to differentiate between a variable of mode **abstract** whose values is determined by the evaluation of a formula, and a variable of mode **explicit** whose value is defined through its components. For instance, a variable of

sort **vert**, and mode **explicit** is semantically equivalent to an indexed family of scalar variables, whose value can be defined component by component by means of formulae of sort **int**. Fig. 2.2 represents a simple **ARCA** program which defines a combinatorial diagram **D** with two colours **a**, and **b**, together with six vertices. In this case, **D!1** denotes the first vertex of the diagram which is to be specified through its two components. In contrast, the value of the vertex **D!2** can only be assigned a value through a formula (as illustrated in Fig. 2.2). A similar convention applies to the colour variables **a_D** and **b_D**.

```
mode D      ==  'ab'-diag 6;
mode D!1    ==  vert 2;
mode D!2    ==  abst vert;
mode V      ==  abst vert;
mode a_D    ==  abst col;
mode b_D    ==  col 6;
mode I      ==  int 3;
b_D         ==  a_D . a_D;
a_D{1}      ==  2;
a_D{2}      ==  3;
...
D!2         ==  rot(D!1,I-1,V);   †
D!1[1]      ==  D!1[2];
D!1[2]      ==  1;
...
```

Fig. 2.2 An **ARCA** program sample

The above example illustrates the way in which moding governs the abstract representation of values by variables. This important question of handling data abstraction in a definitive notation will be discussed in much more detail in §3.2.2.

The aim of the **ARCA** notation for graphics is not just to display, and manipulate diagrams. In an incremental environment where a diagram can be the subject of a further interactive application, definitive notations present profitable means to capture the conceptual model of each object e.g. the diagram. The **ARCA** underlying algebra enables us to associate relevant semantic information with each variable. A diagram can be a

† *rot is a diagram operator.*

particular instance of a group which encodes the multiplication table, and gives direct visual interpretation to algebraic relations within it. For instance, figures 2.4, and 2.5 are Cayley diagrams respectively representing the cyclic group C_{12}, and the symmetric group S_6. It is important to underline that definitive notations are powerful enough to express all semantic information in a Cayley diagram by choosing an underlying algebra including a class of graphs resembling Cayley diagrams.

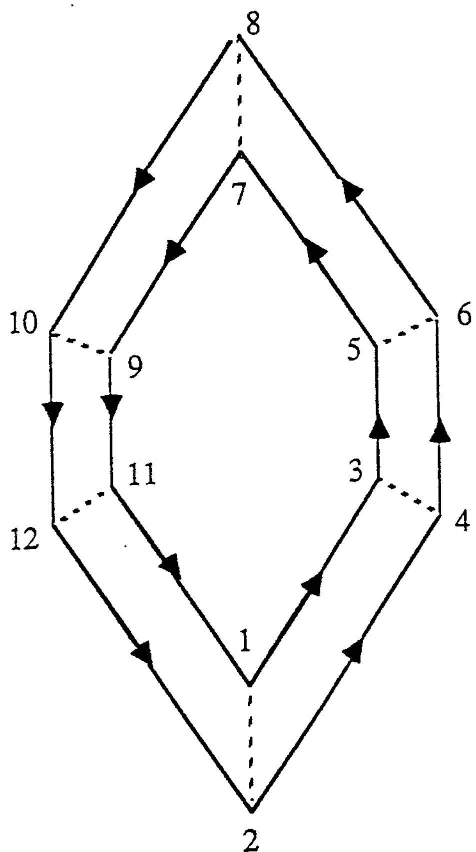

Fig. 2.4 The cyclic group C_{12}

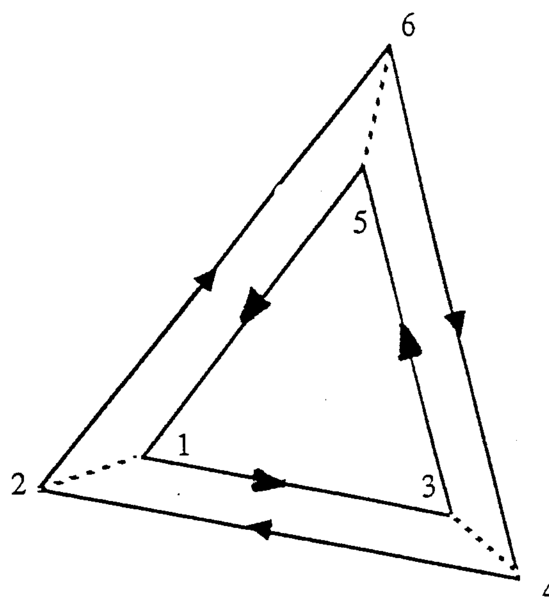

Fig. 2.5 The symmetric group S_6

ARCA aims to support both the depiction of Cayley diagrams and their subsequent group theoretic interpretation. The definitive programming paradigm offers better prospects for intelligent interaction than conventional programming methods.

2.4 DONALD: a definitive notation for line drawings

The definitive programming paradigm is also illustrated by **DONALD** for the description, and interactive manipulation of 2-dimensional line drawings [BABH86]. Its underlying algebra includes five data types namely **real**, **integer**, **point**, **line**, and **shape**. A variable of sort **shape** represents a collection of points (of sort **point**), and lines (of sort **line**). Scalar values are represented by variables of sorts **integer**, and **real**. A point in the plane is a pair of scalars, and a line is defined by an appropriate pair of endpoints. In addition, the **DONALD** underlying algebra includes a panoply of standard operators acting upon shape, line, point, or scalar variables.

(a) Arithmetic operators

DONALD is a definitive notation that provides the user with arithmetic operators acting upon scalar variables. Amongst them are the addition, mutiplication, substraction, division operations respectively denoted by +, -, and /. As in **ARCA**, the programmer is provided with a residue operator (%) which denotes the remainder of the division of one integer by another.

(b) Vector operators

The class of vector operations includes vector addition (+), vector subtraction (-), and multiplication of a vector by a scalar (*). The semantics of these operators are as in **ARCA.**

(c) Point and line operators

Amongst operations that apply on lines are the construction of a line from two endpoints denoted by [,], the intersection of two lines denoted by *, and the inverse of a line which is an operator that swaps the endponts of a line denoted by ~. The point-construction operator is denoted by { , }, and takes as arguments two scalar values, namely the endpoints of the newly formed line.

(d) Geometric operators

As in **ARCA**, the user is given some geometric operators that allow her/him to rotate, and scale geometric objects. A rotation operator is denoted by **rot(x,y,z)** ,where **x** is either a point, line, or shape variable, **y** is a center point of rotation, and **z** is the angle of rotation. In the same way, variables of a complex type such as lines and shapes can be scaled using an appropriate operator. It is denoted by **scale (a,b)**, where **a** is either a line, or shape variable, and **b** is a scaling factor.

(e) Projection and selection operators

A line can be projected on to its initial and final endpoints using the unary operators **.1**, and **.2**. In the same manner, a point can be projected on to its x- and y- coordinates by means of the unary operators **.x** and **.y**. In a particular shape **S**, a point, line, and sub-shape labelled **var-name** are denoted by **S{var-name}**, **S[var-name]**, and **S/var-name**.

(f) Shape operators

It is possible to create new shapes by joining two shapes together using the operator +. The arguments of this operator must be variables of the same type **shape** not constants in order to avoid dealing with a new level of added hierarchy.

On the other hand, the way in which a **shape** variable represents a set of points, lines, and sub-drawings can be difficult to specify. A sort **openshape** is introduced to support explicit definitions of a **shape**. The use of **openshape** has the same role as the "moding" facility of the **ARCA** definitive notation. A variable of sort **openshape** comprises a set of variables of type **point**, a set of variables of type **line**, a set of variables of type **shape**, and a set of variables of type **openshape**. This definition allows an openshape to be evaluated recursively, by looking at the points, and lines of each **openshape** and **shape**, until it generates the entire shape, at which point all the information required for evaluation is available.

The design of **DONALD** constitutes another research into potential applications of

the definitive programming paradigm for interaction. The recursive definition of a **shape**, and the concept of **openshape** is very similar to the notion of a file system directory as a union of files, and sub-directories. On that basis, the user-interface in **DONALD** is based upon introducing a window through which the relationships within each context can be viewed, and manipulated. This interface emphasises the hierarchical structure of a shape, illustrating the usefulness of definitive principles for representing abstraction. The problems encountered in **ARCA** and **DONALD** in finding appropriate ways to reference variables at different levels of abstraction suggest the need for a unifying approach for handling abstraction. One of the principal goals of this research is to formally model data abstraction within the definitive programming framework in order to handle references to objects in a formal way independent of the application in mind (§3.3).

2.5 EDEN: an evaluator of definitive notations

A plausible method of extending the simple definitive programming paradigm described above is illustrated by an Evaluator of Definitive Notations [Yu87]. In the context of an interactive dialogue, **EDEN** allows the user to specify the relationships between objects, and enables her/him to modify them. Furthermore, the evaluator can be programmed to take the initiative in peforming some *actions* on objects which are implied by the relationships themselves. For instance, suppose that the user defines "a book to be on a table", and "a paper to be in the book". In case the user performs an operation on the "book" such as "remove", the evaluator will automatically redefine the status of the "book", and all objects related to it such as the "paper". **EDEN** is a general purpose dependency maintainer which can be used for a specific application with the addition of an appropriate interface. The **EDEN** environment supports three characteristic programming constructs: **definitions, functions,** and **actions.**

2.5.1 Definitions

A definition is used to specify a relationship between two or more objects. **EDEN** includes two kinds of variable, namely a *value variable*, and a *formula variable*. The value of a value variable is always defined explicitly. In the following example,

a = 0;

b = 2;

c = 5;

d = b * c + a;

the variable **a** will be assigned the value **0**, while the variable **d** will be assigned the explicit value **10**. Thus, the semantics of such variables is the same as that of traditional procedural variables.

On the other hand, a formula variable defines a delayed instantiation of the value of a variable. In this case, the definition determines implicit relationships between the different variables involved. For instance,

a is b + c;

b is 0;

The variables **b**, and **c** do not have to be defined when the definition occurs, **EDEN** initialises them to "undefined", until an explicit value is assigned to them. However, the principal benefit gained by using a formula variable is that **EDEN** will automatically perform user-specified actions whenever components of the formula are changed. The dependencies are constantly checked, so that incoherent values, and definitions are avoided. In the above example, the value of variable **a** depends closely on the values of variables **b**, and **c**, which implies that whenever the values of **b**, or **c** changes the value of **a** has to change as well.

EDEN offers a primitive definitive notation based on an underlying algebra of basic data types such as **integer**, **character**, **string**, **pointer**, and **list** with respective operators.

EDEN is a low-level evaluator that borrows most of its syntax from the **C** programming language. It is worth mentioning that **EDEN** variables will implicitly belong to a certain type depending on the definition assigned to them. In that sense, **EDEN** is a typeless definitive language.

(a) Arithmetic operators

An **EDEN** Integer variable is assigned a value by means of an explicit integer constant, or a formula involving arithmetic operators such as the addition, multiplication, division, modulus, and substraction of other integer variables. The syntax of these definitions is equivalent to the syntax of arithmetic expressions in the programming language **C**, that is +, -, %, and /.

(b) String operators

Unlike the programming language **C**, the "string" data type is built-in the definitive language **EDEN**. It is considered to be an array of characters to which a string value can be assigned. Amongst the operators that can be applied on strings, we cite the length of a string denoted by #: **string** $\times$ **integer**, the concatenation of two strings indicated by // : **string** $\times$ **string** $\rightarrow$ **string**, and a built-in function denoted by **substr**: **string** $\times$ **integer** $\times$ **integer** $\rightarrow$ **string** which returns a sub-string of a certain input string.

(c) Pointer operators

EDEN allows the user to define pointer variables. The prefix operator & returns the address of a variable, and the prefix operator * references the variable pointed by a particular pointer. For instance,

```
i_ptr = &i;
i = *i_ptr;
```

where **i_ptr** is a pointer to variable **i**.

(d) List operators

List data structures are an important component of the **EDEN** definitive language. The user can collect data into a list on which a wide range of operators can apply. Amongst them are the selection of an element of a list characterised by [] : **list** × **sort** → **sort**, the length of a list denoted by # : **list** → **integer**, the **append** operator : **list** × **sort** → **list** which causes datum to be added at the end of a list, the **insert** operator : **list** × **integer** × sort → **list** which allows a certain datum to be inserted at a specified position in the list, the **delete** operator : **list** × **integer** → list which removes a datum from a certain position in the list, and finally the **shift** operator : **list** → **list** that deletes the first element of a list.

2.5.2 Functions

EDEN provides the user with suitable means for defining functions. Unlike the programming language **C**, **EDEN** functions parameters are all passed by value. They are collected in a list named **$** such that **$1** is the first parameter, **$2** is the second parameter, and so on. A function called with appropriate arguments will return a value to the caller. In this manner, formulae can be defined with functional arguments. For instance,

```
func square
{
        auto result;
        result = $1 * $1;
        return result;
}
b = 2;
a = square(b);
c is square(d+e);
```

will assign the value 4 to variable **a**, and define **c** by the formula $(d+e)^2$.

2.5.3 Actions

Recording and maintaining functional relationships between variables is an important aspect of definitive programming. The user is provided with a useful facility to define updating actions in **EDEN**. Similarly to spreadsheet actions (§ 2.1), the **EDEN** actions are invoked whenever certain variables are revised. However, in **EDEN**, the user is able to specify the name and nature of each action, and the variables which prompt its invocation. For example,

```
proc print_var : a
{
        writeln(" var a is ",$1);
}
```

is an action that will print the value of the variable **a** whenever it is redefined. **EDEN** also provides the user with a dependency list associated with each variable of the form

a ~> [print_a]

The **EDEN** definitive language also presents the user with a query operator that retrieves the definition, and dependency information of any variable.

§3 Definitive principles for abstraction

Computer-based methodologies for systems design, and analysis require the user to understand, and manipulate abstractions of the objects of direct interest. The success of such methodologies hangs on human ability to deal with such abstractions, and the degree to which programming languages capture such abstractions. This chapter stresses the need for formal methodologies within a problem-solving framework, and interactive context when abstraction can be modelled. We start by a historical review of different meanings and uses of "abstraction" amongst computing circles. Two examples where abstraction has been formalised are then distinguished: abstract data types, and definitive notations. In short, we introduce a generalisation of the definitive programming paradigm for handling data abstraction based on an underlying context-dependant algebra.

3.1 The concept of abstraction

The term "abstraction" has been used in different ways, and several frameworks in Computer Science without reference to a universal formal model for understanding it. One would talk about an abstract machine, an abstract specification, or an abstract implementation, without a common agreement amongst different applications on the meaning of each individual notion. From the perspective of a person trying to solve a problem these higher level concepts may appear to be too vague to merit deeper consideration. The methodologist however, argues that the additional effort required to understand, and manipulate abstractions more than pays off in the quality of the resulting solutions. For this, we must seek a formal abstraction methodology that is universal to a wide range of applications. Perhaps a solution to the confusing different understanding of the term "abstraction" is to consider the reasons behind it, and some explicit examples.

Programmers, and indeed problem solvers in general, have two basic strategies for handling unknown problems : abstraction and decomposition. These strategies offer two different ways of solving a complex problem by simplifying it in some way.

Decomposition is the strategy embodied by the time-honored Machiavellian dictum to "divide and conquer", solving large problems by dividing them into simpler, smaller ones that can be solved independently. "Abstraction" is the strategy of ignoring certain details about the original problem so as to transform it into a simpler and more general one.

The methodologies described in this section illustrate the two forms of abstraction that have been important in the history of programming: procedural abstraction, and data abstraction.

For example, consider the problem of computing the sum of the squares of two numbers, **3** and **4** (that is, compute 3^2+4^2). We may first simplify the problem by decomposing it into a sequence of three simpler problems : (1) compute the square of **3**, (2) compute the square of **4**, and add the two results together. We assume that the final step of addition is sufficiently fundamental that we need not to consider it further. However, the first two steps can be restated in more simpler terms : "Compute the square of a number". We may now apply the principle of abstraction to simplify the problem further. We notice a similarity about computing the square of **3** and computing the square of **4**. By abstracting away the particular details of the **3** versus the **4**, both sub-problems can be solved by a single more general solution, namely, that of computing the square of **n**, where **n** can represent any number. On the whole, abstracting a function is equivalent ot the creation of a new operator e.g. **square of.**

The use of abstraction is nothing out of the ordinary. It is inherent to all mental activities : we abstract continously without being aware of it. Abstraction is commonly used to describe a complex object or a complex activity with the omission of details that are unimportant at the level of discussion. For example, someone learning to drive can represent a car simply by four properties: the gas, the brake, the clutch pedal, and the steering wheel. The engineer designing the car would use a model that also shows the relationship between the pedals, and the engine. For the driver's point of view, the design of the engine is irrelevant; for the engineer, it is of crucial importance. The intention is

to make the truly relevant aspect more intelligible, and more easily manageable. Often abstraction is used to describe the function of a machine or a procedure without actually specifying any of the details of its construction or implementation. In addition, it is usually very important to distinguish between different *degrees of abstraction*. A detail that is irrelevant at one level of discussion may be of fundamental importance at another level. As has already been said, all human thought and work makes use of abstraction. The fact that we have no hesitation in calling two such entirely different objects as a coffee table and a dinning table as a "table" neatly demonstrates that all concepts are in fact abstractions. With the concept of "table" there are borderline cases where it is not immediately obvious whether an object should be called a "table" or not. However, the concept of abstraction in exact sciences (especially in Mathematics) is sharply defined and abstraction is not any more synonymous with "vague". What part does abstraction play in programming methodology? This question may be looked at by studying the historical progressive use of abstraction in such a domain.

(a) Numbers :

Bit strings are used to represent many different kinds of entities, such as integers for example. This is done in a number of different ways depending on the entity to be represented. In programming, however, we think and reason in terms of the entire set of integers without being aware of the machine representation and restriction introduced by it : we abstract. It is pointless for programming notations to introduce "binary numbers" in addition to "integers". We want to have as little as possible to do with the representation of numbers in the machine.

(b) Assembly languages :

Machine instructions are also coded internally with strings of bits, and the first programmers had to know the machine code for each instruction. Symbolic notations were introduced for instructions and these developed into assembly languages. In this way, it was possible to abstract from the actual coding in the machine . This abstraction

excluded certain once popular possibilities, such as the instance of the same bit string in the memory to represent both an instruction and a piece of data operated on by instructions.

(c) Programming languages :

The late 1950s and 1960s saw the introduction of high-level programming made it possible to write programs without being aware of the instruction set of the machine. The compiler translates the "abstract instructions" of the programming notation into "machine instructions". As a result of this, programs became more and more machine-independent. This "getting away" from the machine led to another example of abstraction where compilers generate extensive tables as a high-level description of the state machine at a particular moment.

(d) Variables :

Another example of a development that leads away from the physical machine is the way one thinks about the memory function. In assembly languages the memory of the machine is considered to be a collection of identical memory locations, which may be referred to by using addresses. However, high-level languages introduced symbolic entities used to store information : these are *variables* in which *values* are stored. Such variables are intended for several structures varying from integers to complex frameworks such as the need to describe a complicated organic molecule. The task of the compiler is to represent the variable in a linearly ordered memory of identical locations which is an appreciable abstraction from the hardware structure.

(e) Procedures :

A computer program describes a complicated operation for performing a desired task in terms of a set of elementary operations specified previously. A comparable situation can be found in plane geometry, where complicated structures are built up from a small number of simple operations with ruler and compasses. To perform the desired task, the computer program specifies a complex interplay of elementary operations that

cannot be taken in at a glance. For this reason, the algorithm is not built up from elementary operations actually available; instead sub-tasks that might serve as intermediate steps are described at different *levels of abstraction*. This is another phenomenum that occurs in plane geometry. Here we find non-elementary activities such as "construct the perpendicular from a given point P to a given line L". The specification of such an activity, which describes the effect obtained by its execution, has a dual function. For the person who uses the activity as a "module" for performing more complex operation, the specification describes the net effect without indicating the way in which that effect is obtained. For the person who has to implement the activity, the specification describes what the effect of the implementation should be, without going into the applications of the activity. The subtasks introduced in designing a program can be explicitly recognised in the program if they have been implemented by means of procedures. The breakdown of a task into activities is the most important feature of orderly software design, in fact, most procedural notations support this form of abstraction.

3.2 Formal abstraction models

The previous examples show that the concept of abstraction has been used in several ways. It is also clear that the word "abstraction" may have a different meaning in two different contexts, since there is no reference to a formal model for abstraction. Nevertheless, some attempts have been made to capture abstraction in a more formal way. Two approaches are considered below: abstract data types, and definitive notations.

3.2.1 Abstract data types

The development of formal models for data types and structures has grown out of two primary concerns: the desire to specify data and its operations in an implementation

independent form, and the desire to use mathematical systems to describe the properties of data and its operations in a rigorous form [No79].

Appendix A illustrates the development of formal models for describing data structures. Some of the models described such as Codd's relational model [Co70], and Rosenberg's graph model [Ro71] provide a basis for much of the current research. However, the important concept that has emerged from these models is that the mathematical basis for specifying data types should include both the set of values represented by the structures, and the operations allowed on the structures [Ye78].

An important meaning of the term *abstraction* has gained a lot of support in the programming languages community, when the original models describing data types in terms of known mathematical structures such as graphs, or relations have gradually been replaced by more flexible mathematical systems called *abstract data types*, and based on *heterogenous algebras*.

This crucial programming concept was introduced in the late 1970s in response to a view that data types should not be characterised by the way in which their values are represented, but by the operations that can be carried out on their values. Data types that are specified in this way are called abstract data types. They are viewed as a collections of *objects* (§ 1.3.4) together with *operations* that might apply on them. We shall emphasise that an abstract data type is supposed to be totally independent of its representation, in the sense that all implementation details are hidden from the user. She/he is offered a catalogue of operations, and she/he only needs to know what they are supposed to do, not how they do it. This essential design methodology for data abstraction is known as *information hiding*. The approach was first proposed by D.L. Parnas in 1972 [Par72] when he suggested that the behaviour of software *modules* should be specified completely in terms of their external effects. He then stated, that a module specification must provide to the intended user all the information that she/he will need to use the program correctly, and nothing more. Such a module hides a "secret", namely,

the representation of the data object that the module manages. As has been said before (§ 1.3.4), each module was considered as providing a number of functions which can cause changes in state and other functions which can only give to a user program the values of the variables hanging up that state. For each function, he specified a set of current values, inititial values, parameters, and most importantly an effect. To the outside user, the module provides a set of access functions that are used to create, alter, and observe instances of the abstract data type. There is no way for anyone or anything but the implementation of the module itself to access functions. This type of module that Parnas first described has come to be known as abstract data type. It is abstract because the details of the concrete representation of the data type are unknown to the user. It has also been called an *encapsulated data type*, since the details of implementation are locked away from the user inside a capsule. The functions that access an abstract data type are commonly referred to as its op erations.

An abstract data type, therefore, presents itself to the user not as a data structure but as a collection of abstractions. These are the operations that allow one to create, observe, and alter objects of the abstract type. The task of implementation of an abstract data type consists of determining a concrete representation of objects of the type and writing the procedural abstractions that operate on objects thus represented.

Starting with **SIMULA-67**, many programming languages, such as **CLU**, **MESA**, **EUCLID**, **MODULA-2**, **SMALLTALK**, and **ADA**, allow the programmer, to a varying degree, to define an abstract data type by providing special language constructs for encapsulating both the representation, and the concrete operations that implement the abstract view.

As an example, it will now be shown how a new data type called **stack** might be introduced. The idea is that **stack** becomes the type for items made up of elements all belonging to an existing type that is called **type**. The following operations are available for making stacks and manipulating them :

CLEAR : Create an empty stack.

PUSH : Add an element to the stack.

TOP : Get the first element of a stack.

POP : Remove the top element of a stack.

EMPTY : Check whether a stack is empty.

The definition of the new data type **stack**, includes a specification of the names of the above operations, the types of their arguments and results, and an axiomatic characterisation of their properties. The specification of the type **stack** could be :

New type : **stack**

Uses : **type, boolean**

Operations :

CLEAR :		→ stack	†
PUSH :	stack × type	→ stack	
POP :	stack	→ stack	
TOP :	stack	→ type	
EMPTY:	stack	→boolean	

Characteristics:

EMPTY (CLEAR) = **true**

EMPTY (PUSH(s,i)) = **false**

TOP (PUSH(s,i)) = i

POP (PUSH(s,i)) = s for each s in type **stack** and i in type **type.**

The possibility of *parametrizing* user-defined abstractions provides a still more flexible and powerful tool for expressing abstractions in programming languages. For example, the abstract data types **integer stack**, and **book stack** may display the

† *An operation is specified as a function type* $_1$, *type* $_2$, ..., and (viz. ×) *type* $_n$ *into* (viz. →) *type* $_\alpha$, *type* $_\beta$, ..., *type* $_\zeta$.

same abstract behaviour independently of type of the items that are in the stack. A natural way of expressing this is by defining a *generic/polymorphic type*, where the type of storable items is a parameter.

Fig. 3.1 represents the description of the abstract data type stack in SIMULA-67. The existence of an encapsulating mechanism to enclose in a textual unit both the structure that represents the operations and the procedures that represent the operations improves the localisation of modifications. A change to the data structure is likely to require a change in the access procedures; but the effect of these changes is confined within the boundaries of the encapsulating mechanism.

```
class     stack_member;

begin     ref(stack_member) procedure top;
          next_member:-none
end


class     stack;

begin     ref(stack_member) first;
          ref(stack_member) procedure top;
          top:-first;
          procedure pop;
          if -> empty then first: - first.next_member;
          procedure push(e); ref (stack_member)e;
          begin if first =/= none
          then e.next_member: - first;
          first:- e;
          end push;
          boolean procedure empty;
          empty := first == none;
          first :- none
end       stack
```

Fig. 3.1 The abstract data type "stack" in SIMULA-67

The encapsulating mechanism is known as the *class* in **SIMULA-67**, and **SMALLTALK**, the *module* in **MODULA-2**, or the *cluster* in **CLU** [Ma86]. Although classes are abstractions themselves, they can also be organised in a

hierarchy, where each class has at most one superclass; because the superclass is exactly as any other class, it can have a superclass, and so on. Thus, the resulting structure can be represented by a tree (Fig. 3.2) where each class inherits all the properties of its superclass(es). Objects of different classes communicate by sending messages to each other. If an object is sent a message it must at run-time provide the operation also called *method* to respond to such message. For example, if one wishes to implement a generic abstract data type **set-of-T** providing operations **insert**, **delete**, and **is-in**, which require the equality test operator to be defined on the components of the parameter type T. In SMALLTALK one needs to to define a class, say, **anyset** implementing the **set-of-t** concept. To implement the methods of **anyset** it is necessary to use the equality operator, for example, the equality operator is a message sent to the parameter of **insert** to compare the value to be inserted with the values already stored in the set. If **complex** is another class implementing the type "complex", then one can create sets of complex numbers by instantiating an object of class **anyset**, say **cset**, and sending it messages like **insert c1**, **insert c2**, ..., where **c1**, **c2** are complex numbers. The equality operator invoked with **insert** will automatically be bound exactly to the equality test method provided by **complex**.

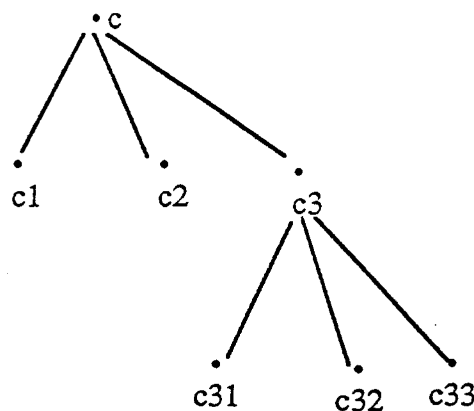

Fig. 3.2 class hierarchy

Though there are certain theoretical aspects still under investigation, the general algebraic approach to the specification, and modelling of abstract data types seems to have gained widespread acceptance [GH78]. Object-oriented languages (§ 1.3.4) such as **SMALLTALK**, enforce a formal methodology for modelling real-world entities by means of systems of classes, and messages. The current concept of data abstraction has evolved from early notions of data structuring, influenced significantly by research in the theoretical foundations of abstract data types, the use of formal systems for providing explicit semantics for programming constructions, and attempts to "package" complex data structures and their operations into components that allow the construction of reliable systems. Algebraic approaches to data abstraction are expected to gain prominence, and are expected to be used in more research on the construction of complex, multi-level abstraction systems.

3.2.2 Definitive notations

The definitive programming paradigm, as illustrated by **ARCA** and **DONALD**, supports data abstraction. In fact, it exploits special-purpose notations based upon an underlying algebra suitable for the application in mind. Every definitive variable which belongs to a particular type of the algebra can be assigned a value either explicitly, or by means of a formula. The way in which the components of a variable are defined and referenced determines the nature and level of its abstraction.

In the unconventional desk calculator, the underlying algebra is the "integer algebra" with its basic arithmetic operators (§2.2). A variable in such a definitive programming environment belongs to a trivial level of abstraction i.e **integer**. In fact, a variable can only be viewed as as an atomic structure that cannot be decomposed, and has to be assigned an atomic integer value. In that sense, the unconventional desk calculator represent a simple environment where abstraction is

an unambiguous issue due to the basic underlying algebra on which it is built.

On the other hand, **ARCA** is a definitive programming notation based on a richer underlying algebra of diagrams, vertices, colours, and integers. As has been said before (§2.3), each **ARCA** data type is built from the basic data type **integer**. A **col**, or a **vert** type is an *array-of* integers, while a **diag** type is a pair comprising a *list-of* vertices, and a *array-of* colours. In that sense, **ARCA** can be viewed as a two-level abstraction system where the top-level algebra is constructed from the **UDC** simple "integer algebra" using the "constructors" *list-of, array-of, set-of, composition*, and *identity*. These will be referred to as transfer operators [Wi76] since they allow us to move from one algebra to another (see Fig. 3.3).

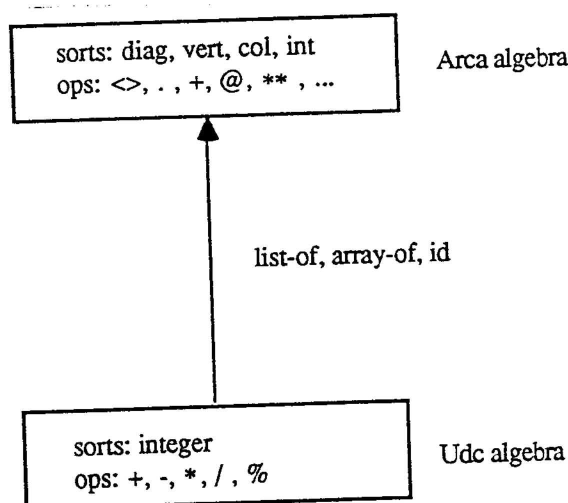

Fig. 3.3 The **UDC** abstraction model

Thus, an **ARCA** data type supports higher level abstractions than the **UDC** data type. The constructors *array-of, list-of, set-of* are used to synthesise complex **ARCA** data types from the **UDC** integer data type. Every **ARCA** operator can similarly be defined in terms of **UDC** operators. In order to control abstraction in **ARCA**, a "moding" facility was introduced such that complex data structures could be viewed as single objects, or decomposable items (§2.3). In that way, the user could define a variable of mode "abstract" which would imply that the considered

variable can only be observed as an individual object whose objects if they exist are not of interest. On the other hand, a variable of mode "explicit", can be defined through its components with possible reference to the first-level algebra i.e the **UDC** algebra. The moding facility adds yet another level of abstraction above the **ARCA** underlying algebra where each typed variable can belong to an "abstract", or an "explicit" mode (see Fig. 3.4).

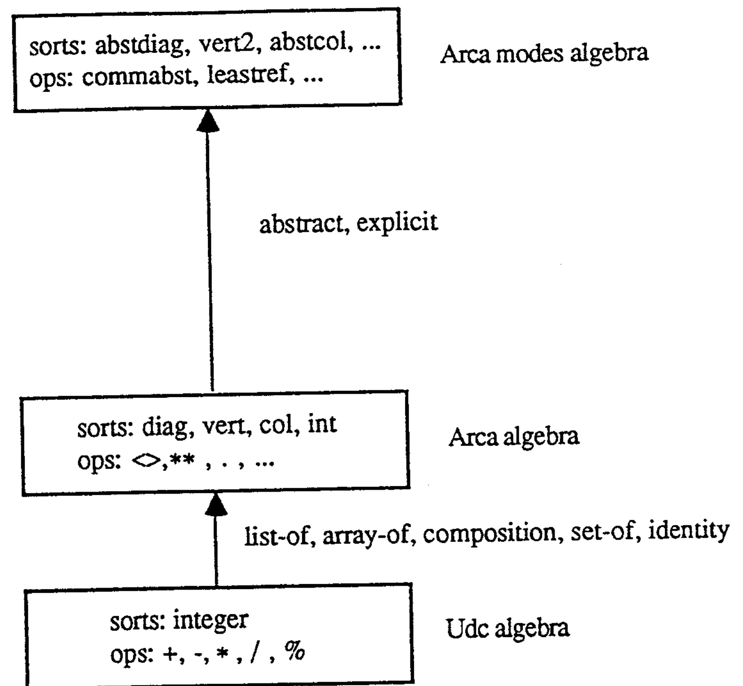

Fig. 3.4 The **ARCA** abstraction model

However, there is no way one could infer the mode of a variable simply from its structure. For instance, the user has to declare a variable of mode **vert 2**, or **col 2** although the two mode define an "explicit" *array-of* integers with two components. Furthermore, the **ARCA** operators refer to the "algebra of modes" in order to determine the mode of a result from its arguments. In this case, the "modes algebra" includes types such as "abstract" **diag**, "explicit" **vert**, etc, together with operators that apply on moding variables. Amongst them are two distinguished operators. The first one will be referred to as the **greatest common abstraction** operator, since it

takes as input two abstraction types and returns a type from which both input arguments can be abstracted (see Fig. 3.5). On the other hand, the second operator which will be referred to as the **least common refinement** operator, for the reason that it defines an abstraction type which both arguments can represent (see Fig. 3.6).

	Abst diag	Explicit diag	Abst vert	Explicit vert	Abst col	Explicit col	Abst int
Abst diag	Abst diag	Abst diag	@	@	@	@	@
Explicit diag	Abst diag	Abst diag / Explicit diag	@	@	@	@	@
Abst vert	@	@	Abst vert	Abst vert	@	@	@
Explicit vert	@	@	Abst vert	Abst vert / Explicit vert	@	@	@
Abst col	@	@	@	@	Abst col	Abst col	@
Explicit col	@	@	@	@	Abst col	Explicit col / Abst col	@
Abst int	@	@	@	@	@	@	Abst Int

Fig. 3.5 The greatest common abstraction operator

The relations over this algebra are used in **ARCA** by complex vertex, colour, and diagram operators to infer the modes of their respective results e.g.

vert n + **vert** n = **vert** n, and **vert** 2 + abst **vert** = abst **vert**;

The fact that the moding of variables in **ARCA** is based on a specific underlying algebra of diagrams, vertices, colours, and integers leads us to a number of redundant cases (see Fig.'s 3.5 and 3.6). In fact, types of the same structure such as **col**, and **vert** should be considered under the same heading **array-of** so that the moding information can be extracted from the structure of the type. A general 3-level abstraction system that would be formed of a "modes algebra" resting on an "algebra of structured types" such as *list-of*, *array-of*, *id-of*, and *composition-of* (see Fig. 3.7).

	Abst diag	Explicit diag	Abst vert	Explicit vert	Abst col	Explicit col	Abst int
Abst diag	Abst diag	Explicit diag	@	@	@	@	@
Explicit diag	Explicit diag	Explicit diag / @	@	@	@	@	@
Abst vert	@	@	Abst vert	Explicit vert	@	@	@
Explicit vert	@	@	Explicit vert	Explicit vert / @	@	@	@
Abst col	@	@	@	@	Abst col	Explicit col	@
Explicit col	@	@	@	@	Explicit col	Explicit col / @	@
Abst int	@	@	@	@	@	@	Abst Int

Fig. 3.6 The least common abstraction operator

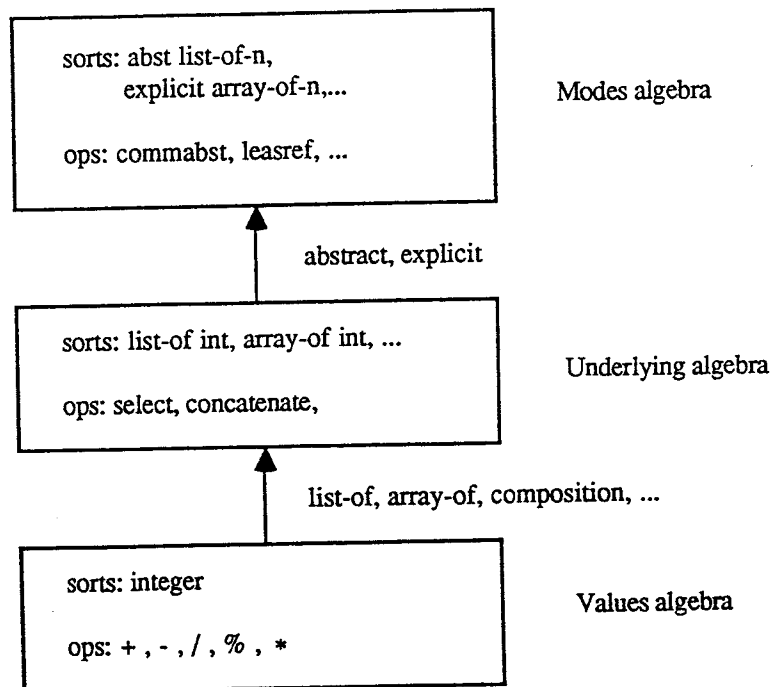

Fig. 3.7 The **ARCA** abstraction model generalised

In **DONALD**, the data abstraction question is handled in a different way. In this case, the underlying algebra comprises **real**, **integer**, **point**, **line**, and **shape** types together with respective operators (§2.4). Scalar values are represented by

real, or **integer** variables, points in the plane by **point** variables, line segments (i.e lines defined by a pair of endpoints) by **line** variables, and line drawings including sets of points, lines, and a set of scalar attributes described by **shape** variables.

The abstraction of data structures in **DONALD** faces a recursive definition of the type **shape** viz.

shape = *set-of* **integer/real** attributes + *set-of* points + *set-of* shapes

The solution adopted is based upon the fact that a **shape** variable resembles a file system directory as a union of files, and subdirectories. In the same manner, a **shape** variable can be declared globally by means of an expression of type **shape**, or denotes a constituent of an **openshape** variable which may itself a variable of type **shape**. This abstraction facility is similar to the "abstract", "explicit" moding of **ARCA** variables. In the same way, an "**openshape** S" identifies an "explicit" **shape** variable which enables the subsequent declaration of attributes, and components of "S". On the other hand, a **shape** variable "S" describes a whole shape whose value must be defined by means of a **shape** expression, and therefore cannot be defined componentwise.

The **DONALD** environment can be described as a 3-level abstraction system, where the lowest algebra includes the **integer**, and **real** types together with their respective operators. Using constructors such as *set-of*, the intermediate algebra of **shape, line, point**, and **scalar** types is built. Finally, the top-level algebra is added to create a higher level of abstraction where **shape** variables can be observed as whole objects, or through their components (see Fig. 3.8).

However unlike **ARCA**, the last layer of abstraction is restricted to **shape** variables. There is no way, the user could choose to specify "abstract", or "explicit" **point**, and **line** variables. There is a default case where **line, and point** variables have to be defined componentwise.

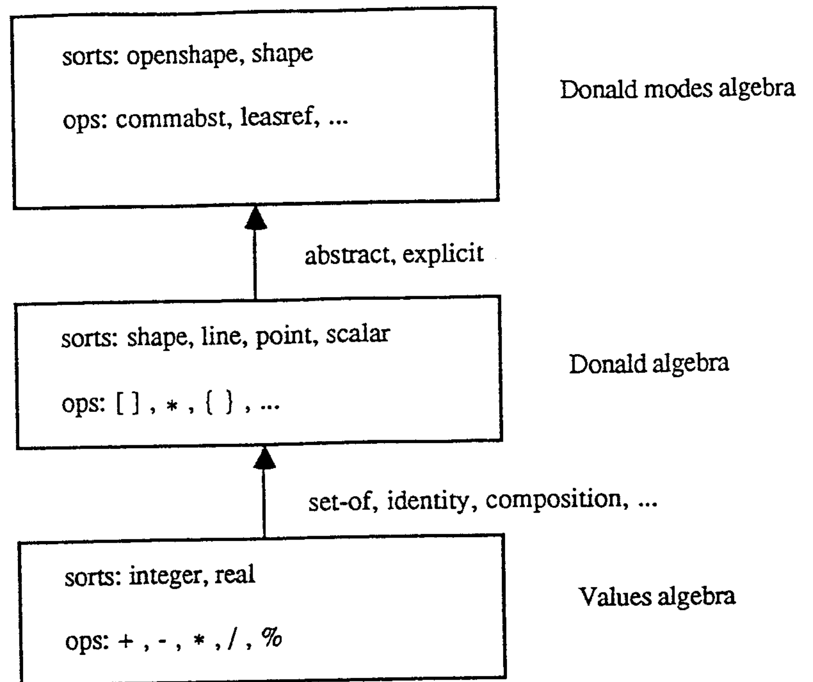

Fig. 3.8 The **DONALD** abstraction model

A better abstraction system in **DONALD** would be a more general one, where types could be replaced by their structures using the constructors *set-of*, *composition-of*, and *identity-of*. In addition, the top level abstraction algebra would offer types such as "abstract" *set-of* **integer**, or "explicit" *set-of* (*set-of* **integer**) so that the mode of every **DONALD** variable could be declared in terms of its basic structure (see Fig. 3.9).

ARCA and **DONALD** are examples where the definitive programing paradigm supports data abstraction. The issues discussed in both cases suggest a formal abstraction model on which a universal definitive programming notation could be based.

3.3 Algebraic abstraction methodology

The central component of the definitive programming paradigm is the underlying algebra from which typed variables can be defined in order to describe

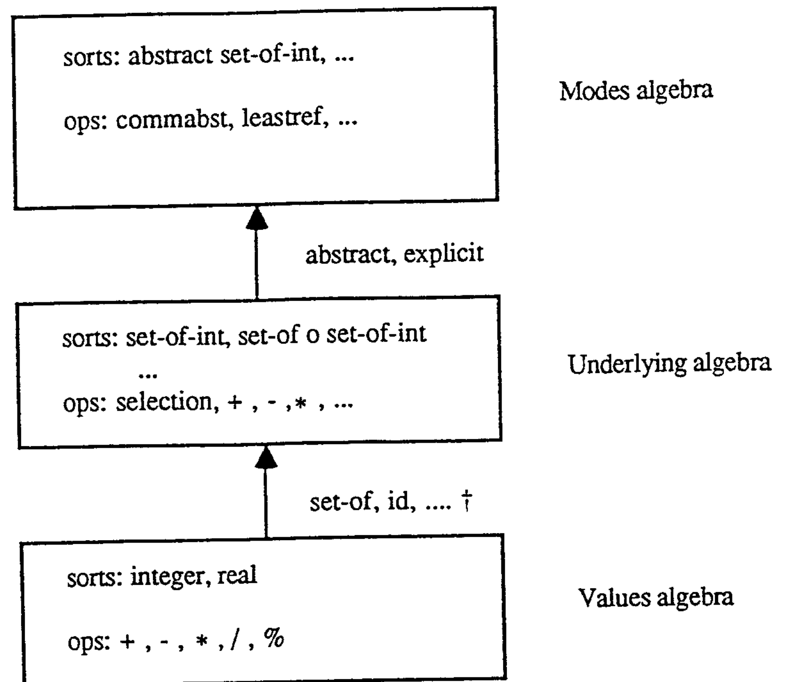

Fig. 3.9 The **DONALD** abstraction model generalised

the state of dialogue. Every variable will be assigned a value either explicitly, or by means of a formula involving some other variables, and algebra operators. The values of variables will be evaluated with reference to some pre-defined values, and operators belonging to a basic algebra from which every definitive underlying algebra is abstracted. For instance, in the context of the **ARCA** underlying algebra, the syntax and semantics of the arithmetic operators and **integer** values are assumed given.

In this context, it is of interest to note that the semantics of the dialogue will depend on the particular interpretation of values in mind. For instance, an **ARCA** diagram may be viewed as a geometric object, as a group, or as an array of coordinate vectors and incidence permutations. This can also apply to the basic data types of the "values algebra". Let's consider the following example over the **UDC**,

a = b+c;

c = 7;

d = 6;

b = c;

?a;

The evaluation of such a state of dialogue over an appropriate "symbolic algebra" will be *value of a is value of b plus value of c* etc. In the context of an "integer algebra", the evaluation of such a definition will be **a = 14**. On the other hand, if we consider an evaluation over the "binary bits algebra", the state of dialogue will be viewed as **a = 1110, b = 0111, c = 0111**, and **d = 0110**. (A definitive statement may be compared with an essay where each single word has a context-independent meaning. However, the semantics of a specific instance of a particular word must in general be inferred from the relationship that the word in question has with other words on the same line, in the same paragraph, or even on the same page. Similarly, a definitive variable is linked with other variables by means of operators over a particular algebra.)

ARCA and **DONALD** illustrate how it is possible to formally describe abstraction within a definitive programming environment. Our objective is to identify generic techniques for describing moding of definitive variables. As has been illustrated above, this is achieved by explicitly describing complex data types such as the **ARCA diagram** and the **DONALD shape** in terms of basic data types. At the lowest level, we must consider an algebra of primitive types such as **integer, real, character**, and **boolean** together with the arithmetic and boolean operators that might apply on them. The explicit data types for any underlying algebra can then be expressed as structured data types over the "primitive algebra" using the constructors. Amongst them are *list-of, array-of, set-of, identity*, and *composition* operators. Furthermore, variables of such structured types can be given values by selecting some chosen components, and defining them by formulae of the

appropriate sorts; in this way the modes of these variables will be formally described.

Finally, the top-level algebra will increment the level of abstraction of each single definitive variable. Any type would be either "abstract", or "explicit". Therefore, a choice would be possible to either decompose, or not a structured variable, and subsequently each one of its components. Moreover, the operators will include the "greatest common abstraction", and the "least common refinement" operators that can define the mode of the result of some operations on definitive variable. In this way, we can in general define a hierarchical 3-level abstraction system composed of a "values algebra", an "underlying algebra", and a "modes algebra" (see Fig. 3.10).

The ultimate result of the application of abstraction will be order, comprehensibility, and correctness, and this will forge abstraction into a weapon against chaos and unreliability.

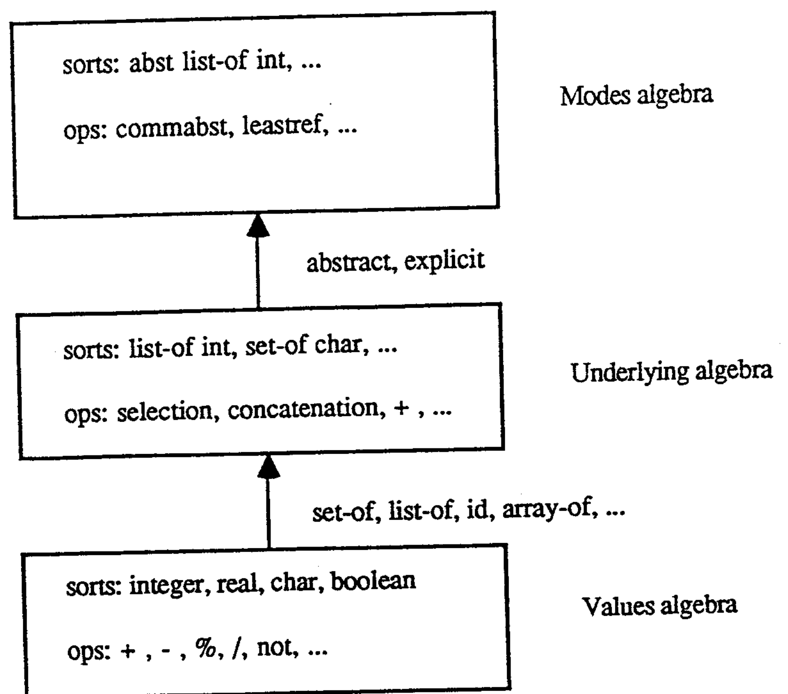

Fig. 3.10 A general definitive abstraction model

§4 The design of DENOTA

Definitive notations are mainly characterised by two important features. Firstly, they provide suitable means to restart a dialogue by recording the transient values of variables, and the persistent relationships between them which constitute the state of the dialogue. Secondly, definitive notations permit data abstraction by supporting many different ways to abstractly define and reference the same object. The aim of the present research is to design, and implement a tool that handles data abstraction within the definitive programming paradigm.

DENOTA is a definitive notation supporting data abstraction illustrating the principles of the general abstraction model described in §3.3. As in **EDEN**, the underlying algebra is based upon *list* data structures. **DENOTA** could be used in two ways. Firstly, definitive notations based on the general abstraction model could be implemented by translation into **DENOTA**. Secondly, **DENOTA** could be used to enhance the *list* based definitive notation of **EDEN**, thereby creating a more powerful version of the evaluator supporting moding of variables.

We have already referred informally to the mode concept (§2.3). It is appropriate at this point to give a more formal account of moding and its significance for the definition and reference of definitive variables. This account will focus on the *list* data structure, but the concepts can be applied more generally by representing complex data types in terms of the *list* data type (cf. §3.3).

A simple way to create a complex data type is to link a set of elements together in a single list. We adopt the traditional definition of the *list* data type, where a *list* is either an atom or a list of *lists* [Wi76]. The *list* we adopt is a dynamic data structure whose atoms are scalars viz. **integer, real, boolean,** or **character** (see Fig. 4.1). A *list* is a particularly flexible structure since it can shrink or grow on demand, and elements can be accessed, inserted, or deleted at any position.

A *list* may be represented by a tree where each internal node is labelled by *list* **n** (where **n** is the number of children of the node), and each leaf node is labelled by an atomic sort.

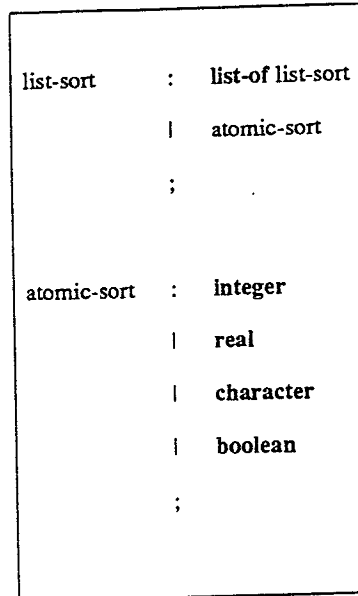

Fig. 4.1 The *list* data type

The purpose of moding is to precisely describe the way in which a definitive *list* variable is to represent a *list* value. For instance, a *list* value **[[a,b,c],d,[e,f,g]]**, can be represented

(1) as the value of a single *list* variable **u**,

(2) as the value of a *list* variable **v** with three components the first and last of type *list*, and the second an atom,

(3) as the value of a *list* variable **w** with three components, where the first is the value of a *list* variable with three components etc.

It is clearly not convenient to enumerate all the possible ways in which a value can be abstractly represented. Instead we attach a mode to a variable, so that for instance **u** is viewed as an "abstract list", **v** as an "explicit list with three components" whose first and last components are "abstract list"s and whose second component is an "atom", and **w** as

an "explicit list" with three components, whose first component is an "explicit list" with three components etc. Formally, when declaring a variable, we associate with it a *list* structure whose form is depicted in Fig. 4.2. Such declaration takes the form of a sequence of simple mode declarations. For instance, the variable **u**, **v**, and **w** would be respectively declared as follows:

(1) **mode u = abst list;**

(2) **mode v = list 3; mode v_1 = abst list; mode v_2 = atom; mode v_3 = abst list;**

(3) **mode w = list 3; mode w_1 = list 3; ...**

Note that partial specification of the mode of a variable is sometimes appropriate; we adopt the convention that the default mode of a component variable is "abstract list".

Specifying the mode for a variable constrains the way in which it can be defined. For instance, the variable **v** above cannot have the following definitions:

v = u;

- **u** might not be a list with three components,

v = [2,4];

- **v** is a list with three components.

v = 7;

- **v** cannot not be an atom.

In order to express constraints on variable definitions formally, we must introduce an ordering relation on modes. This requires a brief digression.

Given two modes **m** and **m'**, we say that **m** is *more abstract than* **m'** if every value that can be represented by a variable of mode **m'** can also be represented by a variable of mode **m**. It will be to write **m < m'** to indicate that the **m** is more abstract than **m'**. (The choice of "<" rather than ">" is just a matter of convention: the reader might read **m** "less concrete than" **m'**.)

Note that, in this ordering there are incomparable modes; for instance:

list 2, **list 3**, and **atom**

are mutually incomparable. Note also that **abst list** is the unique least element with respect to "<". (In generalising these concepts to other definitive notations, there will not necessarily a least element. In **ARCA** for instance, there can be no variable that represents both a vertex and a colour.)

In any ordering, there is a possibility that lattice operations of least upper bound and greatest lower bound are defined. In this context, it turns out that two modes with a common abstraction have a greatest common abstraction, and two modes with a common refinement have a least common refinement. For instance:

list 2 and **list 3**

have no common refinement, but share one common abstraction viz. **abst list.**

The semantic rules governing definitions can be formulated in terms of the abstraction ordering. For a definition to be valid, the mode of the left-hand side must be at least as abstract as that of the right-hand side. The mode of the right-hand side is determined by the nature of the operators involved and the modes of the arguments. This will be illustrated in more detail below.

4.1 Abstraction in DENOTA

The definitive notation **DENOTA** has been designed to support moding as described above. The first function of **DENOTA** is to allow the user to describe the mode of abstraction of any definitive variable over a general underlying algebra of *list* data structures. Its second function is to provide the user with suitable means to assign values to a certain variable respecting its mode. Finally, the interpreter returns the status (i.e. mode and value) of each variable whenever the user requires it.

As explained above, there are three characteristic statements in **DENOTA** to support abstraction: the **mode declaration**, the **value definition** , and the **variable**

interrogation .

Moding of variables was first introduced to differentiate between a variable of mode "abstract" whose value is assigned to it directly, and a variable of mode "explicit" whose value is defined by its component values. Basically, the mode of a typed variable specifies the way in which the variable can be re-defined. The aim of this tool is to present a formal method for specifying the mode of *list* definitive variables. The value of a particular variable is usually defined by an expression of a certain type whose mode will be examined by the interpreter in order to check that the semantic rules described above apply.

The underlying algebra adopted by **DENOTA** is based on *list* data structures. The possible modes of variables over the algebra can be inferred from the BNF definition of the *list* sort. It is a structure composed of one or more elements where each component could either be a *list* data sort, or an *atomic* data sort such as **integer**, **real**, **character**, and **boolean.**

By convention, a variable of mode list **n** could be viewed as a *list* data structure that can be decomposed into **n** elements. On the other hand, a variable of mode *abst list* is to be specified in its entirety as an object of type *list*. Finally, a list variable could belong to an *atomic* sort from the "values algebra" defined in §3.3.

Fig. 4.2 illustrates the moding of *list* definitive variables as a tool for handling data abstraction.

4.1.1 Atomic moding

Every definitive variable over the underlying algebra will be assigned a value at some stage of the dialogue. All values will be expressed in terms of an *atomic* sort whose values are pre-defined in the "values algebra". A variable of mode *atom* is declared to be of a certain primitive type. For instance, the definitive statement "mode **x** = *atom*;" declares a new variable **x** to be a single object of sort *list* that cannot be decomposed, and

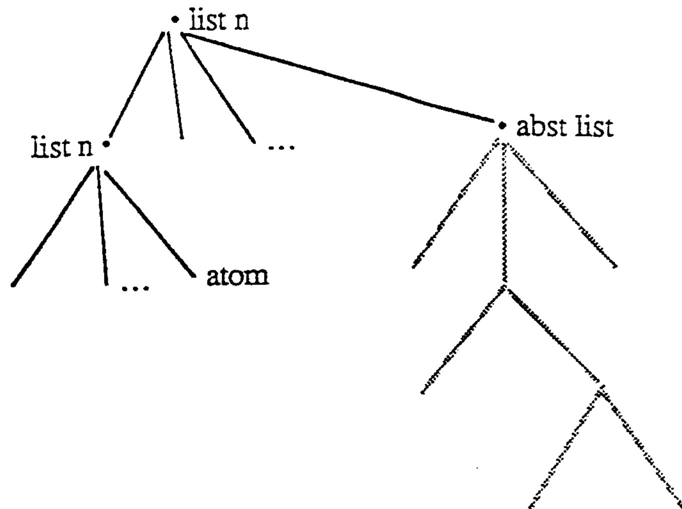

Fig. 4.2 Modes of *list* data structures

whose value will have to be an *atomic* value e.g. **x** = 1;

4.1.2 Abstract moding

There is no way the user could assign values to particular components of a variable of mode "abst list" since they are inspecified in the mode declaration. (Note that the attribution of *atomic* values to *abstract list* variables is permitted.) For instance, "mode **x** = *abst list*"; declares to be a *list* variable with no information about any of its potential components. The user might then assign a value to the variable **x** taking in account the syntax of a *list* value e.g. **x** = [1,2];

4.1.3 Explicit moding

The role of the "constructor" **list-of** is to build a new structured sort, and corresponding operators, in order to enhance the "values algebra". It is not always sufficient to collect new sets of values that are only accesssible as one whole sort. We should be able to choose, consult, and alter any component of the structure. For this, a specific class of operators is defined to allow access to the ingredients of a synthesised sort. Such operators are called "selectors", and are the inverse of the "constructors" (§3.2.2). While data structures are built by means of "constructors", they can be

dismantled using "selectors". Amongst them is the selector

_ : list-sort × integer → list-sort

for *list* data structures.

The mode of every component of an "explicit list" variable can be either an *atom*, an *abst list*, or a *list* n. By convention, the default mode of every component of a variable of mode *list* n will be an *abstract list*. Whenever a value is assigned to a variable of mode *list* **n**, the interpreter creates **n** expected components whose values may have to be infered from the parent's value. For instance:

mode x = list 2;

declares a new definitive variable as a list of **2** elements where **x_1** is the first component, and **x_2** is the second one. The user must then give a value to variable **x** that reflects the declared mode of variable **x**. Thus, one would not be allowed to attribute an *atomic*, or a *list* value of a different dimension to **x**.

4.2 Value definitions

A mode declaration statement specifies the structure of a definitive variable so that a certain value could be assigned to it. Depending on its mode, a variable would have either an *atomic*, or a *list* value.

A variable of mode *atom* is given a value either explicitly, or by means of a formula involving primitive operators and atomic variables/values. The primitive operators are pre-defined in the "values algebra", and include arithmetic, or boolean operators. For instance:

mode a = atom;

mode b = atom;

a = b + 10;

where **a** is a variable of mode *atom* whose value is defined by an expression involving the variable **b** and the arithmetic addition operator (+). In the same manner, a variable of

mode *list* **n**, or *abst list* could be attributed a value either explicitly, or through a formula comprising *list* operators, and list variables.

The arguments of list operators are built from the "values algebra" using the constructor **list-of** (see §3.2.2), and some primitive sorts. The semantics of such operators is described in terms of the list selector, and some primitive operators. Thus, the interpreter selects particular components of the arguments, and applies a pre-defined operator to them e.g. arithmetic addition or perhaps simply the identy function.

(a) The concatenation list operator

DENOTA provides the user with a structured operator that yields to a new list variable constructed from the concatenation of the two input arguments. For instance,

mode a = abst list;

mode b = list 2;

a = [1,2,3];

b = [4,5];

c = a ++ b;

defines the value of variable **c** to be the result of concatenating the list obtained by evaluating the variables **a** and **b**. The concatenation operator permits the user to group together expressions of any mode. The result will be an "explicit list" of the appropriate size if all the arguments are "explicit list"s, and an "abstract list" otherwise.

(b) The head list operator

The user is given the possibility to select the **head** of a list variable. The operator takes a list variable, or a list value as input and produces the first components of that particular list. For example:

mode a = list 3;

a = [x,y,12];

mode b = atom;

b = head(a);

defines the value of variable **b** to be equal to the value of variable **x**. However, the **head** operator checks that the resulting value respects the mode declaration of the variable to which it is assigned, e.g. variable **x** should be of the same mode as variable **b** (i.e. **atom**).

(c) The tail operator

DENOTA offers a **tail** operator that produces a new list variable by removing the first of its components. Thus, the argument has to be of mode *list* **n** so that the result could be extracted. The interpreter expect the argument of the **tail** to be either a *list* value or a *list* variable of mode **list n**. For instance,

mode a = list 2;

mode c = abst list;

a = [b,3,4];

c = tail(a);

define the value of variable **c** to be **[3,4]**. The result of the **tail** operator is a *list* variable/value whose mode is either an "abstract list", an "explicit list", or "undefined". For instance:

tail (abst list) = abst list;

tail (list n) = list n-1;

tail (atom) = @;

(d) List macro-operations

The way in which a list variable is defined implies that its value will be expressed in terms of some pre-defined values. **DENOTA** provides the user with some macro-operators for list variables based on the primitive operators of the "values algebra". In this way, The user can define complex operators on *list* variables which can perform

primitive operations on particular elements of these structures using the *list* selector. Amongst them are the arithmetic *list* operators, and boolean *list* operators. For instance,

mode a = abst list;

a = [1,2];

mode b = list 2;

b = [3,4];

mode c = abst list;

c = a + b;

define the value of variable **c** to be the *list* addition of the values of both *list* variables **a** and b. The interpreter checks that the semantic rules governing definitions are maintained. In this case, the mode of variable **c** is defined by the **greatest common abstraction** operator applied to the mode of a, and the mode of **b**, i.e. **abst list.** The value of variable **c** will be evaluated to **[4,6]**.

4.3 Variable interrogations

The state of dialogue is defined in terms of the transient values, and persistent realtionships between different variables over the underlying algebra. The status of defintive variable comprises a "mode", and a "value" information. DENOTA provides the user with interrogative statements that extract either the "mode", and/or "value" definition of a certain variable. In this way the user could examine the level of abstraction, value, and relationships of a variable with others. For instance,

mode a = abst list;

a = [1,2];

?mode (a);

?val (a);

?a

define the mode a variable **a** to *abst list*, and its value to be **[1,2]**. The user then inquires

about the "mode", "value", and both "value" and "mode" of variable **a**.

§5 The implementation of DENOTA

DENOTA is a tool for definitive notations encoded in the programming language C. It translates definitive statements into **EDEN** statements in order to automatically handle relationships between variables. As has been said before (§2.5), **EDEN** provides the user with the possibility to define a "formula variable". The value of such variable is expressed in terms of an expression with implicit relationships between the components of the expression, and the variable to which it is assigned. This facility is used by **DENOTA** to capture the hierarchical relationships that exist between a list definitive variable, and its components. It is clear that the value of a particular list variable is dependent upon the values of its components. By using **EDEN** statements, **DENOTA** makes sure that necessary updating of the value of a variable is made whenever the values of its components are modified. In addition, **DENOTA** handles data abstraction of *list* variables, by using mode declarations (§4.1). The value of a list variable is closely dependent upon its mode. For this, the interpreter needs to redefine the value of a variable whenever its mode is re-declared using **EDEN** actions (§2.5.3). On the other hand, despite these outstanding features of **EDEN**, the implementation of **DENOTA** still posed some problems.

In summary, the implementation of **DENOTA** makes use of a generalised definitive programming paradigm based upon definitions, functions, and actions (see [BY87] for an analogous use of **EDEN** for the implementation of **DONALD**).

5.1 EDEN constraints

The translation of definitive statements into **EDEN** was a good opportunity to investigate the restrictions of **EDEN**. As mentioned before (§2.5), **EDEN** introduces two types of variables viz. "value variable" and "formula variable". These define when the value of a variable is to be evaluated. By this, **EDEN** restrains the user to declare either a "formula" or a "value variable". There is no way in which one could define the value of a definitive variable by partially evaluating a formula. For instance, the following

statements:

x = b + c;

b is d;

declare a "value variable" **x**, and a "formula variable" **b**. **EDEN** will not allow the user to define the value of variable **x** by the instant evaluation of variable **b** for example. For this, **DENOTA** provides the user with a suitable operator which instantly evaluate a definitive variable (denoted by | |). For instance, the following statements:

c = a;

b = 10;

x = |b| + c;

will define the value of variable **x** to be **10 + c**.

DENOTA relies on the use of **EDEN** actions to control relationships between variable, and automatically prompt updating functions whenever necessary. **EDEN** defines an action by its name, body, and variables upon which it is prompted. Nevertheless, there is no way the user could define parametrised actions, so that the same action with a parametrised body is invoked with different variables. For instance, the user might wish to declare

proc eval-square ( ): $var_1,var_2,...,var_n$

{

square ( );

}

instead of

proc eval-square-a: $var_1,var_2,...,var_n$

{

square(a);

}

and,

```
proc eval-square-b: var1,var2,...,varn
{
        square(b);
}
```

etc.

The fact that actions are not invoked with arguments means that variables cannot share common actions, every variable must declare its own action which leads to many redundancies.

The order in which formulae are evaluated and actions invoked also led to problems. For instance, an action may be triggered whenever the value of a variable is changed. As soon as a new value definition is made, the interpreter faces a non-deterministic choice about whether to evaluate the new value, or invoke the action, or vice versa. These issues have serious implications when several actions are linked together via close relationships between different variables. (Some of the difficulties associated with synchronising evaluation and action have been remedied in revised versions of **EDEN**.)

5.2 Implementing variable modes

A definitive variable is characterised by a mode, and a value. Both are closely related and constitute the status of the variable. For this, **DENOTA** translates every definitive variable into an **EDEN** "formula variable" of type list with two elements, namely a mode and a value. For instance, the definitive variable **x** declared above will be translated into the following **EDEN** definition:

x is [mode_of_x, val_of_x];

in order to express the relation between the variable **x**, its mode (**mode_of_x**), and its value (**val_of_x**). We will refer to the EDEN variables **mode_of_x** and **val_of_x** as a

moding variable and a valuing variable {for **x**} respectively. Note that both moding and valuing variable may be defined by formulae, and are therefore declared as **EDEN** "formula variables". This ensures that regular automatic updating can be made by **EDEN** whenever the mode, or the value is changed.

This section covers the translation of mode declaration statements into **EDEN** definitions where the mode of each variable is described by an **EDEN** variable in order to illustrate the relationships between the mode of variable and the mode of its components (if they exist). A definitive variable of mode "abstract list" will be translated into an **EDEN** moding variable which is assigned an "explicit value", namely "*", in order to express the fact that a variable of mode "abstract list" is of type *list*, and is not dependent on the mode of any of its components. For example, the following statement:

mode a = abst list;

is translated into

mode_of_a is "*";

the value of such moding variable (**mode_of_a**) describes a list definitive variable of mode "abstract". In a similar way, an "atomic mode" declaration is translated into an **EDEN** moding variable which is assigned an "explicit value", namely "%". For instance, the following statement:

mode a = atom;

is translated into

mode_of_a is " %";

in order to state that variable **a** is an undecomposable atomic variable.

On the other hand, the mode of a definitive variable declared as an "explicit list" is most conveniently represented in terms of the modes of its components. To this end, **DENOTA** translates an "explicit mode" declaration statement into a definitive **EDEN** statement where the value of a moding variable is defined by a list value of the respective moding component variables. For instance, the following statement:

mode a = list 2;

is translated into

mode_of_a is [mode_of_a_1,mode_of_a_2];

in order to formalise the relationship between the mode of variable **a**, and the mode of its components **a_1**, and a_2. It is clear that whenever the mode of either **a_1** or a_2 is updated, the mode of the parent variable **a** has to be re-evaluated. This is motivates the use of an "formula variable" to represent a moding variable. In this way, the mode of a definitive variable is represented hierarchically so that at the base level the mode of every component is either "abstract list", or "atom".

5.3 Implementing variable values

The mode facility is introduced to express the way in which a value could be assigned to a variable. The translation of a value definition in **DENOTA** uses very similar principles to those explained for mode declarations. The translator uses **EDEN** "formula variables" - the valuing variables - to describe the relationships between the value of a variable, and the values of its components.

In **DENOTA** a variable of an "atomic mode" is expected to receive an "atomic value". For this pupose, the corresponding valuing variable is initialised to "undefined" whenever an "abstract mode" declaration is made. For instance,

mode a = atom;

will automatically generate the following variable initialisation:

val_of_a is @;

Similarly, a variable of an "abstract list" mode is expected to receive an "explicit value". For this, the valuing variable is initialised to [] whenever an "abstract list" mode declaration is made. For example, the following statement:

mode a = abst list;

automatically generates the statement **val_of_a is [] ;** in **EDEN**.

On the other hand, the value of a definitive variable of an "explicit list" mode will be dependent upon the values of its components. Thus, the translator automatically generates a definitive statement of the form:

val_of_a is [val_of_a_1,val_of_a_2];

whenever a mode declaration **mode a = list 2** is made. In this manner, the value of a variable is expressed by a *list* formula involving the values of its components.

EDEN actions are used to maintain the relationship between the mode and the value of every definitive variable. It is important that the representation of the value of a **DENOTA** variable is altered whenever its mode is updated. For this, **DENOTA** generates actions for every definitive variable that are activated by the mode of the variable. These actions check that the new mode matches the value of the variable defined previously. For instance, the following action:

```
restore_value_a : mode_of_a
{
        /* checks the mode and the value of a */
}
```

will be activated whenever the mode of variable **a** is updated. It is clear that the checking function will be the same used by all variables. However, we had to declare as many actions as variables since there is no way we could have parametrised actions.

Finally, the interrogation of a definitive variable is simply translated into the query of the corresponding **EDEN** variable. For instance, the following statement:

?mode(a);

will be translated into the following similar query statement in **EDEN**:

?mode_of_a;

that enquires about the definition, value, and relationship of the moding variable **mode_of_a** with other moding variables.

In the same manner, the user enquiry about the value of a certain definitive variable is translated into an interrogative **EDEN** statement of the form **?val_of_a**. On the other hand, the user could enquire about the definition of both mode, and value of a definitive variable by a statement of the form:

?a;

that will be translated to an identical statement in **EDEN**.

§6 Conclusion

The abstraction model outlined in this thesis could be generalised to other structured data types such as arrays, cartesian products, and discriminated unions [BMx]. The concepts and techniques adopted in this work could be used to implement a general abstraction model (§3.3) based on the *list* data type.

As explained in §4.1, the moding of definitive variables can only be specified with reference to an "explicit" model viz. values algebra. The "constructors" are used to build complex data structures, and operators. However, difficulties arise when inferring the behaviour of structured operators on "modes". It is not always possible for a structured operator to determine the mode of its result from an action on values. For instance, it is possible to determine the **mode of an addition of vertices**, but not feasible to decide on the **mode of the permutation of colours**, even if both variables are abstractly defined.

Definitive notations could be augmented to support information hiding. For this, two facilities could be used, namely a *reference moding*, and a *definition moding* of variables. The former would describe the way in which variables are references and defined. The former could describe the way definitive variables could be viewed. For instance, A variable of *reference mode* **abst list** could have hidden components. This concepts would also be useful to express semantic rules such as: the **tail** could apply on **abst list** variables. This potential facilities would provide stronger typing of expression analogous to Abstract data types, and Object-oriented programming paradigms.

Finally, Notwithstanding its limits, we believe that **EDEN** provides an outstanding environment to implement definitive notations based on **definitions, functions,** and **actions.** The concepts mastered by **EDEN** such as dynamic re-configuration of variable values using actions, would be very difficult to implement in other programming paradigms.

Bibliography:

[Am86] P. AMERICA, *Object-oriented programming : a theoretician's approach*, EATCS Bulletin, vol.29, 69-84, (June 1986).

[Ar82] F.E.J.K. ARETZ, *Abstraction*, The Philips technical review, vol.40, no.8/9, pp.225-229, (1982).

[ABCCM83] M.P. ATKINSON, P.J. BAILEY, K.J. CHISHOLM, P.W. COCKSHOTT, and R. MORRISON, *An approach to persistent programming*, Computer Journal, vol.26, no.4, pp.360-365, (1983).

[Ba78] J. BACKUS, *Can programming be liberated from the Von Neumann style ?*, C.A.C.M., vol.21, pp.613-641, (August 1978).

[Ba67] R. BALZER, *Dataless programming*, FJCC-67, pp.535-544, (1967).

[Ber75] A.T. BERZTISS, *Data structures:Theory and practice*, Computer Science and Applied Mathematics, Academic press, (1975).

[Bey85] W.M. BEYNON, *Definitive notations for interaction*, Proc.hci'85 : "People and Computers : Designing the interface", Cambridge University Press, (1985).

[Bey86] W.M. BEYNON, *ARCA - a notation for displaying and manipulating combinatorial diagrams*, TC Report No.78, The University of Warwick, (1986).

[Bey87] W.M. BEYNON, *Definitive principles for interactive graphics*, The University of Warwick Computer Science Research Report No.RR93, (January 1987).

[Bo83] S.R. BOURNE, *The UNIX System*, Addison-Wesley Publishing Company, (1983).

[Br7x] D. BRAND, *A note on data abstractions*, SIGPLAN notices, vol.13, no.1, pp.21-24, (197x).

[BABH86] W. M. BEYNON, D. ANGIER, T. BISSELL, and S. HUNT, *The DoNaLD*

report, The University of Warwick, (July 1986).

[BFS86] C.C. BROWN, J.L. FALK, and R.D. SPERLINE, *Preparing documents with UNIX*, Prenctice-Hall, Englewood Cliffs, N.J., (1986).

[BMx] W. M. BEYNON and K. MURRAY, *The revised ARCA notation*, Computer Science Dept., The University of Warwick, (report being processed).

[BY87] W. M. BEYNON and E. YUNG, *Implementing a definitive notation for interactive graphics*, The University of Warwick Research Report No.111, (November 1987).

[Cl86] J.C. CLEAVELAND, *An introduction to Data Types*, Addison Wesley Publishing Company, (1986).

[Ch68] D. CHILDS, *Description of a set-theoric data structure*, FJCC-68, pp.557-564, (1967).

[Co70] E. CODD, *A relational model for large shared data banks*, C.A.C.M., vol.13, no.6, pp.377-387, (1970).

[CCW85] B.G. CLAYBROOK, A. CLAYBROOK, and J. WILLIAMS, *Defining database views as data abstractions*, I.E.E.E. Trans. Software Eng., vol.11, no.1, (January 1985).

[Da85] J. DAIN, *Error recovery for Yacc Parsers*, The University of Warwick, Report No.73, (October 1985).

[De84] M. DE PACE, *Working with Dbase* II, Granada Publishing Ltd, (1984).

[De85] N. DERSHOWITZ, *Program abstraction and instantiation*, A.C.M. Trans. Prog. Lang. Syst., vol.7, no.3, pp.446-477, (July 1985).

[Di76] E.W. DIJKSTRA, *A discipline of programming*, Prenctice-Hall, Englewood Cliffs, NJ, (1976).

[Do74] A. DONAHUE, *Three approaches to reliable software: language design, dyadic specification, complementary seamntics*, University of Toronto, CSRG 45,

(december 1974).

[Du86] C.B. DUFF, *Designing an efficient language*, Byte, vol.8, pp.211-224, (August 1986).

[DDH72] O.-J. DAHL, E.W. DIJKSTRA, C.A.R. HOARE, *Structured programming*, Academic press, pp.83-149, (1972).

[DFLLRW84] P. DEUTSH, S. FELDMAN, B. LAMPSON, B. LISKOV, L.A. ROWE, and T. WINOGRAD, *Programming languages issues for the 1980's*, SIGPLAN Notices, vol.19, no.8, (August 1984).

[Ea71] J. EARLEY, *Towards an understanding of data structures*, C.A.C.M., vol.14, no.10, pp.617-627, (1972).

[Ea73] J. EARLEY, *Relational level data structures for programming languages*, Acta Informatica, vol.2, pp. 293-309, (1973).

[EB77] H. EHRICH, and J. BERGSTRA, *An axiomatic of the rational dat objects*, F.C.T. Conf., pp.33-38, (1977).

[Fl71] A. FLECK, *Recent developments in the theory of data structures*, Comp. Lang., vol.3, no. 1, pp. 37-52, (1971).

[Fos86] E. FOSTER, *Building simple spreadsheets*, Unisphere, vol.5, no.2, pp.32-36, (1986).

[Foy87] J. FOLEY, *Models and tools for the designing of user-interfaces*, Proc NATO ASI: Theoretical foundations of Computer graphics and CAD, Il Ciocco, (July 1987).

[Ga75] J. GANNON, *Language design to enhance programming reliability*, University of Toronto, Ph.D. Diss., (1975).

[Go75] J. GOGUEN, *introduction to categories, algebraic theories, and algebra*, IBM Tech. Rept. RC 5369, (April 1975).

[Gu75] J. GUTTAG, *Specification and application to programming of abstract data types*, University of Toronto, Tech. Rept. CSRG-59, Ph.D. Diss, (1975).

[Gu77] J. GUTTAG, *Abstract data types and the development of data structures*, C.A.C.M., vol.20, no.6, pp.396-404, (1977).

[GGM76] V. GIARRATANA, F. GIMONA, and U. MONTANARI, *Observability concepts in abstract data type specification*, Lecture notes in Computer Science, vol.45, Springler-verlag, pp.576-587, (1976).

[GHT84] H. GLASER, C. HANKIN, and D. TILL, *Principles of functional programming*, Prenctice-Hall, Englewood Cliffs, (1984).

[GJM86] J. I. GLASGOW, M.A. JENKINS, and C. D. MCCROSKY, *Programming styles in NIAL*, I.E.E.E. Soft., vol.2, no.1, pp.46-55, (January 1986).

[GG78] C.C. GOTLIEB, and L.R. GOTLIEB, *Data types and structures*, Prenctice-Hall, Englewood Cliffs, (1978).

[GG77] D. GRIES and N. GEHANI, *Some ideas on data types in high-level languages*, C. A.C.M., vol.20, no.6, (1977).

[GH78] J. GUTTAG and J. HORNING, *Algebraic specification of abstract data structures*, Acta Informatica, vol. 10, no.1, pp. 1048-1064, (1978).

[GT76] J. GOGUEN, and J. TATCHER, *Initial algebra approach to specification, correctness, and implementation of abstract data types*, IBM Tech. Rept. RC 6487, (October 1976).

[GT78] J. GOGUEN, J.THATCHER and E. WARNER, *An initial algebra approach to the specification, correctness, and implementation of abstract data types*, [Ye78], (1978).

[He70] J. B. HELLER, *A graph theoretic model of data structures*, SIGIR FORUM, vol. VII, no. 4, pp. 36-44, (1970).

[Ho76] E. HOROWITZ, *Fundamentals of data structures*, Computer Science Press, (1976).

[Ho86] C.A.R. HOARE, *Mathematics of programming*, Byte, vol.8, pp.115-149, (August 1986).

[Ho84] E. HOROWITZ, *Fundamentals of programming languages*, second edition, Computer Science Press, (1984).

[HM85] E.C. HAREL, and E.R. MCLEAN, *The effects of using a non procedural computer language on programmer productivity*, Management Information Systems Q., vol.9, no.2, pp.101-120, (June 1985).

[HO82] G.M. HOFFMAN, and M.J. O'DONNELL, *Programming with equations*, A.C.M. transactions on programming languages and systems, vol.4, no.1, pp.83-112, (January 1982).

[Im69] M. IMPERIO, *Data structures and their representation in storage*, Annual review in automatic programming, vol. 5, pp. 1-75, (1969).

[Jo75] S.C. JOHNSON, *YACC : Yet another compiler-compiler*, CSTR No.32, Bell Labs, Murray Hill, New Jersey, (1975).

[Ka82] K.C. KAHN, *Object-oriented languages tackle massive programming headaches*, Electronics, November, (1982).

[KP86] T. KAEHLER, and D. PATTERSON, *A small taste of SMALLTALK*, Byte, vol.8, pp.145-159, (August 1986).

[KT84] A.L. KELLEY, and I. TOHL, *A Book on C*, Bengamin Cummings Publishing Company, (1984).

[Ma86] R. MARTY, *Object-oriented programming*, Computer Physics Comm., vol.38, no.2, pp.181-190, (1986).

[Me67] G. MEALY, *Another look at data*, FJCC-67, pp.525-534, (1967).

[Mi85] H. MILLER, *Introduction to Spreadsheets*, PC World, pp.249-256, (May 1985).

[Mo73] J. MORRIS, *Types are not sets*, A.C.M., POPL-73, pp. 120-124, (1973).

[Mo86] L. MOSKOWITZ, *Rule-based programming*, Byte, vol.11, pp.217-224, (November 1986).

[MGM78] B. MAYOH, B. CHAUDHARY, and W. MERZENICH, *Suggestions about a specification technique*, SIGPLAN, vol. 13, no. 12, pp. 25-28, (1978).

[Ne85] G. NELSON, *JUNO, a constrained-based graphics system*, SIGGRAPH, vol.19, no. 3, San francisco, (1985).

[No75] B. NORDSTROM, *Concepts of very high level language without language without explicit pointers*, University of Umea, Sweden, Ph.D. Diss., (1975).

[No79] G.J. NOLAN, *DASIM1 : A practical exercise in data abstraction*, Proceedings of the symposium on language design and programming methodology, Sydney, (September 1979).

[Od85] J.T. O'DONNELL, *Dialogues : A basis for constructing programming environments*, Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments, Seattle, Washington, (25-28 June 1985).

[Pa86] G.A. PASCOE, *Elements of object-oriented programming*, Byte, vol.8, pp.139-144, (August 1986).

Re75] J. REYNOLDS, *User defined types and procedural data structures as complementary approaches to data abstraction*, New directions in algorithmic languages-75, I.F.I.P. working group 2.1 on ALGOL, S. Shuman-editor, pp.157-168, (1975).

[Ri85] H. RICHARDS, *Applicative programming*, Systems research, vol.2, no.4, pp.299-306, (1985).

[Ro71] A. ROSENBERG, *Data graphs and addressing schemes, JCSS, vol.5, pp. 193-238, (1971).*

[Sh73] B. SHNEIDERMAN, *Data structures: description manipulation, and evaluation*, SONY-Stoney Brook, Ph.D. Diss., (1973).

[Si84] G.F. SIMONS, *Data abstraction*, Byte, vol.10, pp.130-434, (October 1984).

[St73] T. STANDISH, *Data structures: an axiomatic approach*, BBN Rept. 2639, (August 1973).

[SSS77] D.SCOTT, J. STOY, and L. SANDISH, *Denotational semantics: the Scott-Strauchey approach to programming language theory, M.I.T. Press, (1977).*

[ST76] D. SCOTT and R. TENNENT *Data types as lattices*, SIAM Journal of Computing, vol.5, no.3, pp.522-587, (1976).

[Te86] L. TESLER, *Programming experiences*, Byte, vol.8, pp.195-206, (August 1986).

[Th86] S. THOMPSON, *Writing interactive programs in Miranda*, UKC Computing Laboratory Report no.40, University of Kent at Canterbury,(July 1986).

[Tu71] V. TURSKI, *Model for data structures and its application*, Acta Informatica, vol. 1, n0. 28-34, pp. 282-289, (1971).

[TRL77] R. TENNENT, J. REYNOLDS, and D.LEHMAN, *Data types*, Conf. Found. Com. Sc., pp.7-12, (1977).

[TT83] T. TAKALA, and P.J.W. TEN HAGEN (ed.), *Standard graphics as a geometric modelling device*, Eurographics'83, Proceedings of the international conference and exhibition, University of Zagreb, Faculty of Electrical Engineering, Yuguslavia, (31 August - 2 September 1983).

[Wir73] N. WIRTH, *Systematic programming : An introduction*, Englewood Cliffs, N.J., Prenctice-Hall, (1973).

[Wir76] N. WIRTH, *Algorithms + Data structures = Programs*, Prenctice-Hall, pp.1-55, (1976).

[Wir84] N. WIRTH, *History and goals of MODULA-2*, Byte, vol.9, pp.145-151, (August 1984).

[Wis86] D.S. WISE, *The applicative style of programming*, Abacus, vol.2, no.2, pp.20-32, (Winter 1985).

[WY84] D.L. WELLER, and B.W. YORK, *A relational representation of an abstract type system*, IEEE Trans. on Soft. Eng., vol. SE-10, no.3, (May 1984).

[Ye78] R. YEH, *Current trents in programming methodology*, vol.IV, Data structuring, Prenctice-Hall, (1978).

[Yu87] E. YUNG, *An evaluator of definitive notations*, final year project, Computer Science, University of Warwick, (1987).

[Zi75] S. ZILLES, *Data algebra: specifiaction techniques for data structures*, M.I.T., Ph. D. Diss., (1975).

APPENDIX A

[Ba67]
[Me67]
[Ch68]
[Im69]
(early data models)

[Co70]
[He70]
[Ea71]
[Tu71]
[Fl71]
[Ro71]
(data models)

[St73] (an axiomatic model)
[Mo73]
[Sh73] (unified structures and operations)
[Do74]
[Ga75]
[Cl77]
[Ma77]
[Ba78]
[Sc78]

(early algebraic models)

[Zi75]
[No75]

ADT Group

[Gu75]
[Ho76]
[Gu77]
[GH78]

ADJ Group

[Go75]
[GT76]
[Go77]
[GT78]

SS-DS Group

[ST76]
[SSS77]

(connection between algebraic and lattice-theoretic)

[Re75]
[TRL77]

(other algebraic models)

[EB77]
[MCM78]

APPENDIX B

Mode declarations section

BNF definitions:

mode_declaration	:	*mode* var_name = mode_expr;
	|	*mode* var_name = mode_val;
	;	
mode_expr	:	*list* num
	|	*abst list*
	|	*atom*
	;	
mode_val	:	*mode* var_name >< mode_val /*least refinement*/
	|	*mode* var_name <> mode_val /*common abstraction*/
	|	*mode* var_name
	;	
var_name	:	head_name
	|	head_name _ indirections
	;	
head_name	:	letter (letter | digit)*
	;	
letter	:	**a | b | ... | z | A | B | ... | Z**
	;	
digit	:	**0 | 1 | ... | 9**
	;	
indirections	:	num
	|	indirections _ num
	;	
num	:	digit digit*
	;	

Statements samples:

mode a = list 2;

mode b = *mode* a;

mode c = abst list;

mode d = *mode* b >< mode a;

mode c_1 = list 2;

mode x = *atom*;

mode y = *mode* a <> mode b;

mode a = mode b;

Value definitions section

BNF definitions:

```
value_definition    :   var_name = list_val
                    ;
list_val            :   [ list_body ]
                    |   val_expr;
                    ;
list_body           :   list_body , list_val
                    |   list_val
                    ;
val_expr            :   val_expr + val_expr     /* addition of list elements */
                    |   val_expr - val_expr     /* substraction of list elements */
                    |   val_expr * val_expr     /* multiplication of list elements */
                    |   val_expr / val_expr     /* division of list elements */
                    |   val_expr % val_expr     /* modulus of list elements */
                    |   val_expr ++ val_expr    /* concatenation of lists */
                    |   || val_expr             /* boolean OR on list elements */
                    |   && val_expr             /* boolean AND on list elements */
                    |   - val_expr              /* unary minus on list elements */
                    |   head (val_expr)         /* head of a list */
                    |   tail (val_expr)         /* tail of a list */
                    /   (val_expr)
                    |   | val_expr|             /* evaluation of the value of list */
                    |   var_name
                    |   num
                    ;
```

Statements samples:

```
b    =  [2,3];
a    =  [b,[14,c]];
c    =  [a*b,18,[p,36]];
x    =  a ++ c;
l    =  | a |;
s    =  head (x);
```

Variable interrogations section

BNF definitions:

```
var_interrogation   :   ? var_name           /* a variable status */
                    |   ? mode (var_name)    /* a mode status */
                    |   ? val (var_name)     /* a value status */
                    ;
```

Statement samples:

```
?a;
?mode(a);
?a_1;
?c;
?val(c);
```