DATA DEPENDENT PROGRAM GENERATION.

P.QUARENDON

PhD Thesis January 1977.

UNIVERSITY OF NEWCASTLE UPON TYNE.

ACKNOWLEDGEMENTS

My thanks go to Dr. Peter Henderson and Professor Brian Randell for their supervision and help; to past and present members of the IBM Scientific Centre at Peterlee, particularly Ken Hanford, for many interesting discussions; to my wife for her unflagging support and encouragement; to Harry Tuffill for his recent tolerance; to ALGOLW for its reliability under stress and to VM SCRIPT for its help in preparing the manuscript.

I would also like to thank IBM for providing financial support under the Advanced Education Scheme.

ABSTRACT

When information is stored in a computer it can usually be organised in many different ways. If the information is used for a number of different purposes the ideal organisation is not always obvious. It will depend on how often various parts of the data are used, how often they are changed, and the amount of data taking part in each transaction. It may be difficult to predict these parameters in advance, especially in data-base applications where the pattern of use may change as time goes by. Ultimately, one can visualise systems which can automatically choose the optimum representation, or which can substantially assist in the choice. A step in this direction, which could itself find immediate application, is to find a practical way to tailor programs to a particular data organisation. The thesis describes an experimental system which does this for a limited range of programs, and the work which lead up to it. Both data retrieval and simple updates are considered.

One prerequisite is a method of writing the program so that it does not depend on the way that the data is stored. A number of data-base systems achieve this independence by describing the data as a collection of relations. These systems and the background to them are reviewed. The experimental system is loosely based on the use of

relations, but some modifications have been made to make the processing simpler and so that the characteristics of the data organisation can be described. The system incorporates the representation into the program and produces a tailored version which is expressed in abstract, Algol-like code. The result is intended to be similar to code which a human programmer might write in similar circumstances, but as far as possible ignoring the details of any particular implementation.

CONTENTS

1 Introduction 1

1.1 An example 6

1.2 Abstract view of data. 12

1.3 Definitions. .18

1.4 Code generation. 22

1.5 Outline of contents. 27

2 Relational data. 30

2.1 Relational operations. 35

2.2 Relational implementations 40

2.3 Other systems. 48

3 Language F 56

3.1 Problems with Codd's relations 57

3.2 Language description 61

3.3 Data representation. 69

3.4 Background to the implementation 79

4 Compilation. 85

4.1 Pre-processing 87

4.2 Code generation. 95

4.3 Primitive Statements 99

4.4 Conjunction. 103

4.5 Disjunction. 105

4.6 Negation 109

4.7 Projection and related operations. 110

4.8 Choosing a program 120

5 Selective update 132

5.1 Introduction. 133

5.2 Assignments. 141

5.3 Logical expressions. 144

5.4 Other functions. 148

6 An experimental system 161

6.1 Outline description. 161

6.2 Examples 165

7 Conclusions and further work 189

Appendix A 200

Appendix B 206

Appendix C 209

References 211

CHAPTER 1

INTRODUCTION

When designing a computer program, one of the factors which must be considered is the way that the data should be organised. This has a strong influence both on the processing time and on the storage that the resulting program will use.

In a simple program intended for a very specific purpose, the logical organisation of the data is often self-evident. In a larger piece of software, for example a commercial compiler, operating system or data-processing suite, the best organisation is usually much less obvious. A program of this sort will be built from a number of smaller programs which operate on the same data. Considering the sub-programs individually, we may be able to determine the data organisation that each ideally needs, but when they are put together we often find that their requirements conflict. We may then have to decide whether a redundant organisation of the data should be maintained, satisfying the processing requirements at the expense of storage, whether suitable structures should be created temporarily to suit a particular process, or whether some of the component programs should be adapted to work with a structure which is less than ideal. Each of these alternatives might prove the best compromise under some conditions.

The problem is shown up most sharply in the design of a data-base. For example if this were to store information on a number of manufactured assemblies and the parts that each contained, we might ask at different times for:

(i) a complete list of assemblies and their component parts,

(ii) the parts used in a particular assembly,

(iii) the assemblies where a particular part is used,

(iv) whether a part is used in a particular assembly.

In addition new assemblies may be added to the data-base, or an assembly may be modified so that different parts are used. Each of these uses of the data ideally needs a different data-structure. We can choose to store one or more of these to facilitate particular retrievals, but as more redundancy is employed, so the difficulty of modifying the data increases, and so more storage space will be needed.

The aim of the work described here has been to take practical steps toward an understanding of the factors which influence the choice of data structure. There is an emphasis on the application to data-processing, that is retrieving data from, and modifying the data in, an on-line

data-base. This is because these applications often have quite a complex structure, examples come fairly readily to mind and the potential benefits of machine assistance are most obvious in this area. While the design of a systems program like a compiler may be difficult, the job it has to do will remain largely unchanged throughout its lifetime. A data-base on the other hand, models some part of the outside world. The demands made on it may vary from week to week as the interests of its users change, it may grow in directions which were not anticipated by its designers and it may need to reflect organisational changes in the world at large. Ideally one would like to be able to adapt the data structures kept, and the processing methods used, to each change in circumstances.

This thesis describes a processor which will automatically adapt a computation to a given data representation. A practical method of adapting algorithms to work with a particular configuration of stored data could be applied directly in high level data-base systems, where existing data can be used by people with no formal training in programming. For example a supermarket company may have a number of outlets and record the sales at each branch in a central data-base by monitoring the transactions at each check-out. The central data-base is supported by an expert programming team, and performs stock control and provides sales statistics for management at head office. Branch

managers may also have their own individual information needs, but do not have the programming expertise to provide it. This could be overcome by providing a very high level interface to the central data-base and a processor to generate retrieval algorithms automatically, adapting them to the existing data organisation.

In a research environment, for example in studying rock samples, there may be no central programming team. In this case the data-base system may need to take responsibility, not only for generating detailed retrieval algorithms, but also for organising the data representation. A number of systems have been aimed at this broad area, for example the Peterlee IS/1 system (Notley 1972), SQUARE (Boyce 1973) or Woods Lunar Data System (Woods 1972). Stocker and Dearnley in particular have considered the problems of adapting the representation to suit the particular pattern of requests encountered (Stocker and Dearnley 1973). However, to successfully adapt data structures to a pattern of transactions it is clearly necessary to understand how each of the processes depends on the way the data is stored. Developing a system which can reliably adapt programs to a given storage organisation may lead to such an understanding and so to a simple model by which the effect of a change in data organisation can be predicted.

There are also more far-reaching efforts to automate, or to

partially automate, the programming of complete business applications (Goldberg 1974, Krohn 1972). It has been suggested by Darlington, Knuth, Gerhard and others (Darlington and Burstall 1973, Knuth 1974, Gerhard 1974) that a good way to produce a correct, efficient program is to write the algorithms in as clear a way as possible, so making it easy to verify that they are functionally what is needed, and then to transform the program to optimise its performance on the machine. This method is particularly suited to (and indeed may necessitate) automatic assistance. Some of the transformations will be concerned with questions of data organisation, such as whether to store a structure permanently or compute it each time it is needed. Studying ways of adapting programs to a particular data organisation should give insight into a large and interesting class of such program transformations.

The aim therefore has been to consider a program which is written without a knowledge of the actual data organisation, and to investigate how it can be combined with a definition of an actual representation, so that the result executes reasonably efficiently. A system which automatically adapted a program to the representation could be applied directly to high-level data-base systems, and might, in the longer term, assist in the development and maintainance of large applications. The insight gained into the effect of altering the representation might also suggest new methods

of determining the optimum data organisation.

The following sections contain a brief introduction to the work described in the remaining chapters. Section 1.1 gives an example of two programs which are intended to give the same result using different data structures. This shows the type of transformation we are trying to achieve. Then sections 1.2 and 1.3 introduce the abstract view of data which is used, and the way that this reflects the properties of a data representation which are of interest. Some examples are then given to show how the description of the data of interest can be compiled into a program to obtain it. Finally, section 1.5 describes how the remainder of the thesis is organised.

1.1 An example

Suppose that a small factory, manufacturing say electronic equipment, wishes to keep a record of the assembled products it makes and the components used in each. We will assume that this "bill of materials" data is to be kept on a direct access device, so that both sequential and random access can be used.

Each record in the file may contain (amongst others) four fields:

ASSEMBLY - the name or other identification of the finished product.

PART - the name of one of the components used in an assembly.

QTY - the number of components of type PART used in an assembly.

WEEKLY-OUTPUT - the number of assemblies produced per week.

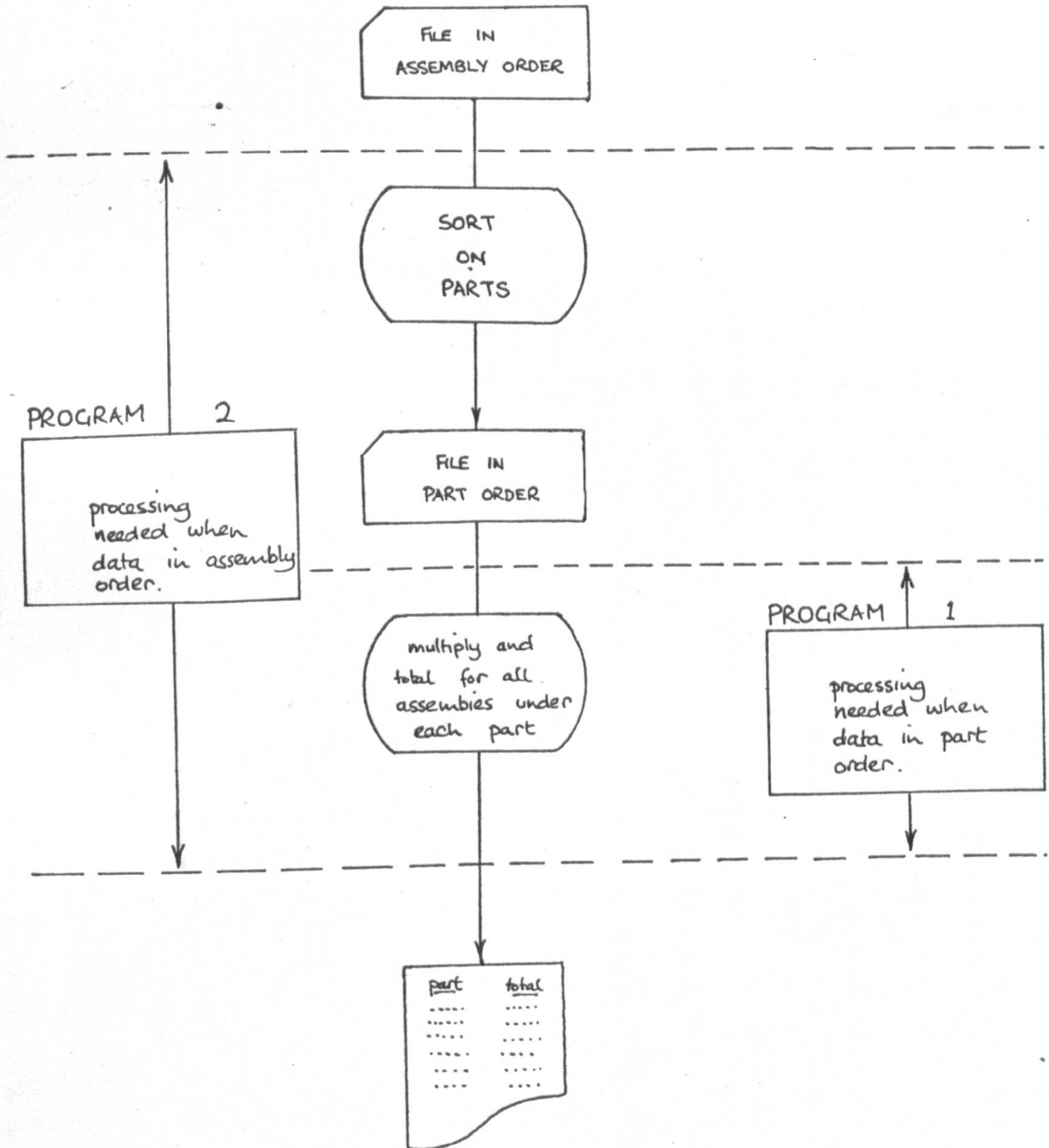
The format is indicated at the top of figure 1.1.

When ordering new supplies of component parts, it might be decided that each should be ordered in just sufficient quantity to cover a single week's requirements. For this purpose, a list is needed showing each part, together with the total number used in all assemblies in one week. Producing this list is quite straight-forward if the bill of materials file were kept in PART order, that is, with all records relevant to one component stored together. It is then possible in one scan of the file, to process the parts sequentially. For each record, the number used in an assembly is multiplied by the number of assemblies produced each week, and the products are summed for the batch of records corresponding to a single component.

However, it is likely that other processing requirements will demand that the records are kept in ASSEMBLY, rather than PART order. There is then a choice between keeping the data redundantly so that both of the orderings are available, or effectively re-sorting the file for this and

ASSEMBLY | PART | QTY | WEEKLY OUTPUT

a) initial data.



b) processing required.

<u>Figure 1.1 Effect of data organisation</u>

similar processes. The two situations are illustrated in the lower half of figure 1.1. When the file is kept only in ASSEMBLY order, to find the weekly usage of all parts the file could first be sorted into PART order, and subsequently processed to sum the quantities used. The processing needed is labelled "program 2". If the data is kept permanently in PART order, only the last step is needed and this is labelled "program 1".

If the application were to be programmed using the Report Program Generator (RPG) (Bowden 1970) and the data is not properly sorted, it will be necessary to perform a sort and then run the program on the newly created file, exactly as illustrated. The file in PART order can be thought of as an intermediate structure needed to produce the result, and this can either be stored, or generated each time it is needed.

If the application were instead programmed in, say, PL/1, more complex internal data-structures can be used. There is no need to perform an initial sort when the data is not correctly ordered and it will be more efficient to omit it. The programming becomes more complicated and to illustrate this figure 1.2 shows the outline of two programs which might be written, the first for sorted data, and the second for unsorted data. (It is not necessary to read these programs in detail.) Program 2 does not obviously perform

```
DCL 1 BM_RECORD,
      2 (ASSEMBLY, PART) CHAR (20),2 (QTY, WEEKLY-OP) FIXED (4);
DCL PNAME CHAR (20), TOTAL FIXED (4);

READ FILE(BM_FILE) INTO (BM_RECORD);

DO WHILE (PART ¬= "ZZZZ");  /*dummy last card*/
   PNAME = PART;  TOTAL = 0;
   DO WHILE (PART = PNAME);
      TOTAL = TOTAL + QTY* WEEKLY-OP;
      READ FILE(BM_FILE) INTO (BM_RECORD);
   END;
   PUT LIST (PNAME, TOTAL);
END;
```

a) Assuming Part order.

```
DCL 1 BM_RECORD,
      2 (ASSEMBLY, PART) CHAR (20),2 (QTY, WEEKLY-OP) FIXED (4);
/* AUXILIARY PART TABLE */
DCL PNAME (N) FIXED (4); TOTAL (N) FIXED (4);
DCL PMAX INITIAL (0); /* GIVES LAST ENTRY USED */

READ FILE(BM_FILE) INTO (BM_RECORD);

DO WHILE (PART ¬= "ZZZZ");
   /* LOOK FOR PART IN TABLE */
   DCL FOUND BIT (1); FOUND = '0'B;
   DO I = 1 TO PMAX WHILE (¬FOUND);
      IF PNAME (I) = PART THEN FOUND = '1'B;
   END;

   IF ¬FOUND THEN
      DO; /* MAKE A NEW TABLE ENTRY */
         I, PMAX = PMAX + 1;
         PNAME (I) = PART; TOTAL (I) = 0;
      END;

   /* ADD PRODUCT INTO APPROPRIATE TOTAL */
   TOTAL (I) = TOTAL (I) + QTY * WEEKLY-OP;
   READ FILE(BM_FILE) INTO (BM_RECORD);
END;

/* PRINT ALL TOTALS */
DO I = 1 TO PMAX;
   PUT LIST (PNAME (I), TOTAL (I) );
END;
```

b) Assuming Assembly order.

Figure 1.2
PL/1 outlines for obtaining weekly-use.

the same function as a sort followed by program 1. A running total is kept for each PART during a single pass down the data and the final values are subsequently printed out. The program shown has been simplified by using a simple linear search to locate part entries in an auxiliary table, and by ignoring any requirement for an alphabetic list of parts. It would be preferable to store the running totals in a binary tree in order to achieve a logarithmic, rather than linear, dependency on the number of different parts appearing. If properly programmed this method will be more efficient than pre-sorting the data.

One of the major concerns of this investigation has been to find a way of reliably achieving optimisations such as this.

With examples written in a conventional programming language, the relationship between programs using different data structures to acheive the same result is often not obvious. As Hopcroft (Aho, Hopcroft and Ullman 1975)suggests, the relationship becomes much clearer if we take a more abstract view of the process.

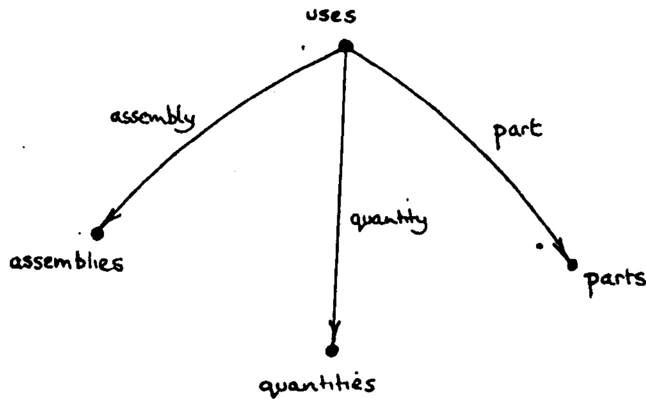
1.2 Abstract view of data

The method of generating programs which we have adopted uses an abstract description of the data, modelling its structure using functions and sets.

In the example, the bill of materials data is held in a sequential file of records. Abstractly we view this file as a set, each member corresponding to a record in the file. We might call this set "uses", as each member represents the use of one part in one assembly.

The records in the bill of materials file contain four fields, of which, for the moment, we will consider only three, ASSEMBLY, PART and QTY. These fields are modelled using functions. For instance, to correspond with ASSEMBLY we use a function "assembly". When applied to a member of the set "uses" it returns the value contained in the ASSEMBLY field of the corresponding bill of materials record.

Very informally, the situation can be shown as follows:



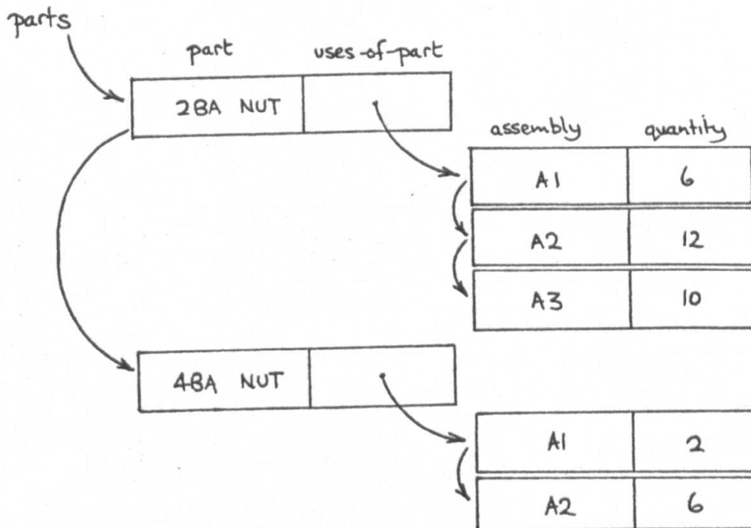
The large dots correspond to sets, and the lines joining them are labelled by functions. For example, the function "part" takes a member of "uses" and produces a member of the set "parts".

A possible physical realisation of the abstract functions is shown in figure 1.3(a). Given a sequential file like that illustrated, we could read through all the records. This is modelled by the ability to sequence through all the members of the abstract set "uses", retrieving all the members in turn. The members of this set correspond to records in the bill of materials file. Having obtained a particular record in the physical file, we can extract the ASSEMBLY, PART and QTY data. This is modelled by the ability to apply to a member of "uses" any of the functions "part", "assembly" and "qty".

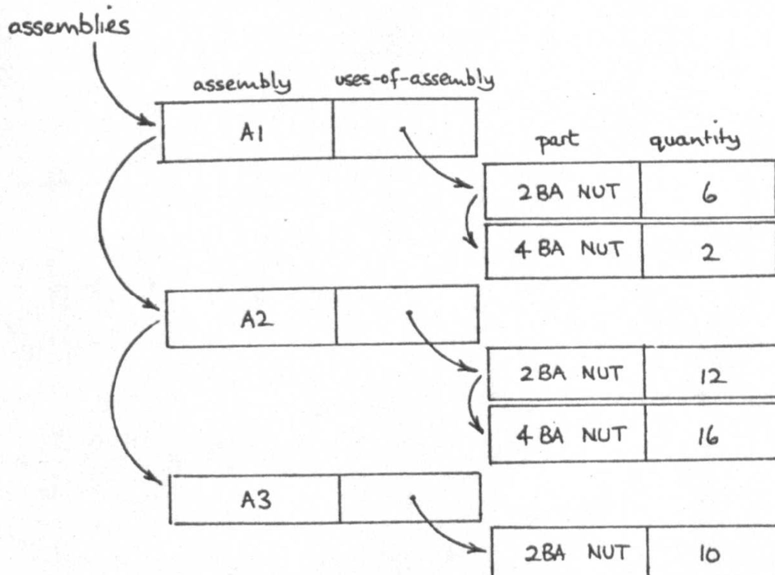
Although we are using functions and sets, throughout this

part	assembly	quantity
2 BA NUT	A1	6
4 BA NUT	A1	2
2 BA NUT	A2	12
2 BA NUT	A3	10
4 BA NUT	A2	16

a) sequential representation.



b) part hierarchy.



c) assembly hierarchy.

Figure 1.3 Alternative representations

thesis they are treated informally. They merely provide a convenient way to ignore unwanted details of a representation. To take an extreme case, the assembly, part and quantity data might be held in three separate arrays, and members of the set "uses" might merely be integer indexes to the arrays. Our abstract description remains unchanged because the same operations can be carried out. In programming terms however, the representation would be considered very different from the sequential file.

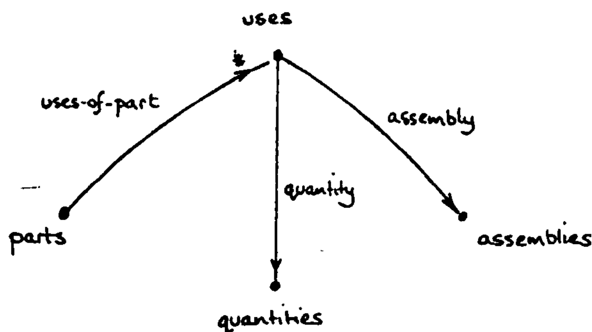
The program in section 1.1 (to find the weekly consumption of all parts) ideally requires that the records relating to each part can be obtained together. The simpler code assumes that the function:

uses-of-part(p) = {b | part(b)=p}

is stored. Given a part p, "uses-of-part(p)" will stand for the set of members of "uses" whose part is p. These correspond to the bill of materials records containing part p. If we stored all the parts as a list of records and in each kept the result of this function, again represented as a sequence, then a hierarchical organisation like that in figure 1.3 (b) would be obtained.

In the figure the functions "uses-of-part","assembly", and "quantity" are represented by the fields USES-OF-PART, ASSEMBLY and QTY. Again the representation is only one of a

number of possible ways of storing the data modelled by:



The function "uses-of-part" produces a subset of (rather than a single member of) the set "uses". The diagram merely shows how the functions and sets are connected and how one can get from one set to another. It does not accurately describe the domains and ranges of the functions.

In a similar way a representation which allowed all the data for each assembly to be obtained together might store the function:

uses-of-assembly(a) = {b | assembly(b)=a}

together with a set "assemblies" and the functions "part" and "quantity". Such an organisation is shown in figure 1.3 (c). Although the sorted files in the earlier example are not quite in this hierarchical form, it would be easy to arrange a subroutine to present them in this light. The first PL/I program, for instance, is structured as if the data were hierarchical.

In summary, we make the assumption that the characteristics of the data organisation which are of interest can be reflected by a model in terms of functions and sets. The details of how these are stored will be ignored.

1.3 Definitions.

We might also record in the data-base information about each part and each assembly. For example we might include the following functions:

cost(p)=n	meaning the cost of part "p" is "n"
number-on-hand(p)=n	meaning there are "n" of part "p" on hand
supplies(m,p)	meaning manufacturer "m" can supply part "p"
weekly-prod(a)=n	meaning "n" assemblies of type "a" are produced per week.

The first two functions, "cost" and "number-on-hand" will be contained in a part file; the function "weekly-prod" belongs in an assembly file. In the original example this data was included in the bill of materials file as a field WEEKLY_OUTPUT, but a problem can arise if the values are stored here. In the bill of materials file there is more than one record associated with each assembly. It is therefore possible, perhaps as a result of an update, that two records could give conflicting information about the number being produced. It is to avoid probems of this sort that Codd, in the relational treatment of data, defines "third-normal-form". Chapter 2 expands on this point.

The function "weekly-prod" does not have this problem, as it

associates only one output with each assembly and so cannot give inconsistent information. However, if it were more convenient to use, the function "weekly-output" can be introduced by defining it in terms of "weekly-prod":

weekly-output(b) = weekly-prod(assembly(b))

By storing this function in a set of fields we get a representation like that used in the original example. But it is now clear that these fields are, in fact, redundant. They depend on other values stored, and in particular are not independent of one another. This is important when considering update because all related fields must be changed consistently.

Further definitions can be made using the available functions, for example:

out-of-stock(p) ≡ (number-on-hand(p) = 0).

This gives a predicate satisfied only by parts which are not in stock.

cost-on-hand(p) = cost(p) * number-on-hand(p).

This logically forms an addition to the parts file, giving the product of two other fields.

rate-used(b) = quantity(b) * weekly-output(b)

defines the number of parts used each week in the production

of an assembly. The part and the assembly are those associated with the bill-of-materials item b.

suppliers(p) = {m|supplies(m,p)}

defines a function corresponding to a field in the parts file containing a set of all suppliers of the part. Using this we can define:

can't-obtain(p) ≡ _not_ _some_ (suppliers(p))

Here _some_ is a function corresponding to the existential quantifier in predicate calculus. Its argument is a set, in this case of the manufacturers supplying part p. _Some_ determines whether there is at least one value contained in the set.

Another operator on sets is _the_. If the set contains a single member it will produce that member, but otherwise is undefined. For example:

use(a,p)=_the_ {r:uses | part(r)=p _and_ assembly(r)=a}

defines a function which, given a part and an assembly, produces the unique member of "uses" which relates that part and that assembly. The function models an index to the "bill-of-materials" file.

The functions or predicates defined by each of these equivalences might be used as part of a program to extract

data of interest from the data-base. The values they represent could be computed each time they are needed, or, to avoid repeated re-calculation they could be stored permanently. The latter choice usually produces an overall representation which stores the data redundantly, with a corresponding update problem.

Given a program to retrieve some data, some of the functions it uses will be supported directly by the representation. Others will not be stored. For these, the definition must be used to relate them to functions which are part of the stored representation. Again returning to the example of section 1.1, the program to find the weekly consumption of parts assumed that it had available a representation modelled by the functions and sets:

parts	the set of all parts.
uses-of-part	giving the bill-of-materials data for a part.
weekly-output	giving the number of assemblies produced.
qty	giving the number of each part used in each assembly.

The first PL/I program shows the straightforward loop needed to do the processing when the data is indeed stored. If "uses-of-part" is not stored directly, a definition must be provided so that its value can be computed. Given only a

sequential file, it must be expressed in terms of the function "part":

uses-of-part(p)= {b | part(b)=p} .

The problem then is to re-organise the computation to make the best of the situation. As the second PL/I program shows, it is often not the best policy to first create real data corresponding to the defined function (performing th preliminary sort), nor to use its definition blindly to compute the uses of each part encountered. This latter course would mean scanning the bill of materials file many times. Instead we would like to achieve the basically sequential program shown, using an auxiliary table.

The next section introduces the way this is done, but taking a less complex example.

1.4 Code generation

Suppose we wished to obtain a list of the assemblies whose production used more than 1000 4BA NUTs each week. The assemblies of interest can be expressed:

{a | rate-used(use(a,"4BA NUT"))>1000} .

Suppose further that "rate-used" were stored in the bill of materials file, and that these records were kept sequentially. It would be necessary to inspect the part and rate-used field of each of the records, looking for the

value "4BA NUT", and a rate greater than 1000. The code needed, when expressed in an Algol-like language, might be:

```
for b in uses do
    if part(b) = "4BA NUT" then
        if rate-used(b) > 1000 then
                        write(assembly(b))
        fi
    fi
od
```

The retrieval could be performed more quickly if the file were inverted on parts. This could be reflected by keeping the field corresponding to:

uses-of-part(p)={b|part(b)=p}

The code could then become:

```
for b in uses-of-part("4BA NUT") do
    if rate-used(b) > 1000 then
                    write(assembly(b))
    fi
od
```

This makes direct use of the function "uses-of-part", giving directly the assemblies containing each part.

If the function rate-used were not stored at all, it would be necessary to refer to the assembly file in order to

compute it, using the definition in the last section. For example:

```
for b in uses-of-part("4BA NUT") do
  let t1 = quantity(b);
    let t2 = weekly-prod(assembly(b));
      let rate-used = t1 * t2;
        if rate-used > 1000 then
          write(assembly(b))
        fi
od.
```

The process by which this code is obtained has a similar goal to the access-path selection carried out in a data-base system. It attempts to produce the required answer at minimum cost. In the example the value of "p" in uses-of-part(p) is given, so that ideally the process should use this function to obtain the assemblies of interest directly. It can do this in the second case. If this function is not stored, then the system must go back to its definition and simulate it by scanning all records and selecting those of interest. The number of iterations needed and so the cost of doing the calculation, will be greater if this is necessary. The method by which the code is produced uses, as far as possible, only structural information to produce an optimum program. In practice direct estimates of the cost of execution must sometimes be used, but are difficult to make.

Update

If the weekly output of an assembly were to be increased, we might record this in the data-base by an assignment, say:

```
weekly-prod ("DVM"):= 60
```

where "DVM" is used to identify an assembled digital-volt-meter.

This logically has the effect of re-setting the field weekly-prod in the record corresponding to "DVM" in the assembly file. Other actions would be needed, however, if the data were stored redundantly. For example if the information were also kept in the bill-of-materials file, a number of fields must be re-set:

```
for b in uses do
    if assembly(b) = "DVM" then
       weekly-output(b):= 60
    fi
od
```

Slightly more work is required if the field rate-used were also held:

```
for b in uses do
    if assembly(b) = "DVM" then
        rate-used(b):=60*quantity(b);
        weekly-output(b):=60
    fi
od
```

Again, many variations of this code are possible, depending on the functions which are available. These pieces of code must be executed when a modification takes place to guarantee that the values stored corresponding to the functions:

```
weekly-output (b) = weekly-prod (assembly(b))
rate-used (b) = weekly-output (b) * quantity (b)
```

remain correct after weekly-prod has been changed. The problem here is essentially to find the fewest changes that must be made so that the resulting data is correct.

In summary, we have investigated a system which aims to adapt programs to a given data organisation. The data organisation is modelled by functions and sets so that unwanted details of the representation can be ignored. Programs are assumed to be written in terms of these functions and sets using a limited set of operations. The operations chosen are similar to those in Codd's relational algebra, but modified to suit the slightly different data

model. The program can assume a different data organisation than that which actually exists, but definitions must then be provided which relate the assumed and actual representations. For retrieval, the system combines the program and these definitions and then searches for a processing algorithm which it considers satisfactory. For update, it has the additional task of manipulating the program to find an efficient way of modifying redundantly held data. The resulting algorithms are expressed in a simple, but informal, Algol-like language.

The system was produced as a step toward understanding the relationship between the processing algorithm and the data organisation. It might, however, have a practical use in data-base environments where it is an advantage to be able to write programs without knowing how the data is stored.

1.5 Outline of contents

The remaining chapters give an account of the work which lead to this approach, and describe the methods used in the experimental system.

There are a number of data-base systems which are based on the use of operations on sets of n-tuples. Chapter 2 reviews some of these systems and the background to them. Codd's work on relational data and the operations to manipulate it were aimed at presenting a view of the data in

an information system which is independent of the way in which it happens to be stored. Relations form the basis of the work described here, but some detailed changes in their treatment were found to be necessary, mainly to give a closer match with predicate logic. The modifications are discussed in Chapter 3. As a preliminary to the description of the code generation process, some further examples are given to show how the relations can be represented in storage.

Chapter 4 describes how a program written in terms of abstract functions and predicates can be compiled to Algol-like code, once the data representation is known. The programs produced are intended to be similar to those which a programmer might write in similar circumstances, but are expressed in an abstract language which has not been implemented. This avoids the need to consider the detailed conventions of an existing language like PL/1 or Cobol, and the output code is used just to suggest the structure of a suitable program. It might be implemented in various ways, for example transliterating to a conventional language, or using an interpreter to form a complete high level data-base sub-system.

Chapter 5 shows how code can be produced to perform updates. When the data is stored redundantly, the code must arrange that all the relevant storage has been consistently

modified.

Chapter 6 describes the structure of the experimental system and gives some details about the way it operates. The system was not intended to be a practical tool, nor was it intended to implement a language which would immediately be suitable for interacting with a data-base. It has been used rather as an experimental test-bed to verify that the methods proposed could be sensibly implemented and to uncover any practical problems which might arise. Some examples are given showing the operation of the system and the output it produces.

Finally, in the light of the promise that the system shows, a number of suggestions are made for future work.

CHAPTER 2

RELATIONAL DATA

In a conventional data-base system, whether based on hierarchies like IMS (IMS/360 1972), or on networks like IDS (IDS 1968), an applications program makes direct reference to stored sequences of data items. When the representation is altered the programs may also need to be changed. In the Codasyl proposal (Codasyl 1971), based largely on IDS, some indirection is provided (using "subschemas" which describe a modified view of the data), but the degree of independence obtained is fairly small.

Codd proposed relations as a means of describing data so as to convey only its inherent structure. The motivation was to allow users of the data to be independent of the stored representation. Similar machine independent models were proposed by Childs (Childs 1968), Kuhns (Kuhns 1969) and Grindlay and Stevens (Grindlay and Stevens 1968).

In the relational model of data, (Codd 1970), a number of underlying sets of objects are assumed. These might represent parts, manufacturers, costs, quantities and so on. These sets are called "domains". All information about the objects is held in a collection of time varying relations. A relation on the domains D1, D2 .... Dn is a set of n-tuples, where each n-tuple has its first component drawn

from D1, its second from D2 and so on. The number of components (n) in each tuple is called the "degree" of the relation.

For example, the manufacturers who can supply various parts can be described using a binary relation, that is a set of pairs. In each pair the first component will be a manufacturer (that is drawn from a domain "manufacturers") and the second a part (from a domain "parts"). A tuple in the relation represents the ability of the manufacturer to supply the part:

$$\text{supplies} \subseteq \text{manufacturers} \times \text{parts}.$$

Sample data for the relation is illustrated diagramatically in figure 2.1. As the table is intended to depict a set, all the rows are different. In the table each column is headed by the domain-name from which the elements are taken. The column ordering is significant, as the domain-names need not all be different. A relation giving the nearest equivalent of each part, for example would have two identical domains:

$$\text{nearest-equivalent} \subseteq \text{parts} \times \text{parts}.$$

To avoid dependence on column ordering Codd suggests that the domain-name could be qualified by a "role-name", so that the combination is unique. A user could then deal with "relationships" where the components are not ordered.

Supplies (manufacturer, part)

manufacturer	part
A & Co.	6 BA NUT
A & Co.	6 BA BOLT
B & Co.	6 BA NUT
C Ltd.	6 BA BOLT
C Ltd.	2 BA BOLT
C Ltd.	4 BA BOLT

Figure 2.1 Sample data for Supplies.

Results (student , exam , mark , tutor)

student	exam	mark	tutor
A	1	50	PROF R
A	2	60	PROF R
A	3	70	PROF R
B	1	30	PROF P
B	2	80	PROF P
C	2	80	PROF R
C	3	85	PROF R

Figure 2.2 Sample data for results.

However, operations of the relational algebra are defined only for relations where ordering is used to distinguish the components.

Normalised Relations

In general, the underlying domains of a relation need not consist of elementary values (such as parts or manufacturers) but may themselves be sets of relations. As an example, a collection of examination results might be expressed as a binary relation:

$$\text{results}' \subseteq \text{students} \times \text{mark-lists}$$

associating with each student a relation (from mark-lists) which gives the mark for each examination he took. Here, members of the domain mark-lists are relations:

$$\text{mark-lists} \subseteq (\text{exams} \times \text{marks})^*$$

(* being used to form the set of all subsets of its argument). Codd in (Codd 1972 b), defines a normalisation procedure to systematically eliminate domains of this sort in favour of sets of elementary values. The first-normal-form form of "results" would be a ternary relation:

$$\text{results} \subseteq \text{students} \times \text{exams} \times \text{marks}$$

obtained in the obvious way from results'.

Second and third normal forms are defined, aimed at preventing unexpected behaviour when an update occurs. If a domain "tutors" is added to the relation "results", sample data could occur as in figure 2.2. The "tutors" column gives each student's tutor. A problem occurs if a particular student took no examinations. It would not be possible to record his tutor's name, as no entries occur in the relation for the student. This anomaly can happen as a result of a deletion. To define a third-normal-form relation, one or more of the domains are designated as the "primary key". Any two tuples in the relation must differ in the values of these domains, so that a value of the primary key uniquely identifies a tuple. In a relation "number-on-hand", "parts" would constitute the primary key, while in "supplies", both a value for "parts" and a value for "manufacturers" is needed to guarantee to identify an individual tuple. In a third-normal-form relation, the value in no domain must be uniquely determined by a (proper) subset of the primary key. The restriction is violated by the example, since there is only one tutor for each student, whereas the key (which uniquely identifies a tuple) consists of both the students domain and the exams domain. To obtain the third normal form of the data in this relation it must be split into two, one containing the examination results (whose key is student and examination) and one containing the tutor information (whose key is just student). The normalisation rules are fully discussed by Date (Date

1975).

2.1 Relational operations

The operations on relations are described in some detail because the language to be introduced in the next chapter was developed from them. However we will not need to make use of the definitions which are given. The operations described are taken from (Codd 1972). Variations are possible, for example (Codd 1970) and that used as an intermediate language by IS/1 (Notley 1972), but their structure is similar. They are defined only for normalised relations, those whose domains are simple sets of integers or strings.

Relations may participate in the usual set operations: union, intersection, difference and cartesian product. The first three operations are only defined for a pair of relations which have the same degree (or number of columns) and where corresponding domains in the two relations are either both sets of strings or both sets of integers. The cartesian product is defined so that an expanded product is obtained. Cross multiplying two relations R and S, of degrees m and n, gives a relation of degree m+n. It is formed by concatenating each tuple from the first relation with all tuples in the second. Denoting the concatenation of two tuples r and s by r^s,

$$R \otimes S = \{r^\frown s \mid r \in R \ \& \ s \in S\}$$

The remaining operations apply specifically to relations. The projection operation is used to select or permute the columns of a relation. If r is a tuple from an n-ary relation R, then the elements can be selected and re-ordered by an operation r[v] where v is a vector of indices. (The operation is identical with APL array indexing). For example, ("A" "B" "C")[3 1] = ("C" "A"). The projection of R on v is then defined by:

$$R[v] = \{r[v] \mid r \in R\}$$

Two of the projections of the relation supplies are shown in figure 2.3.

The other important operation is called "join". This concatenates two relations as in a cross-product, but result tuples must also satisfy a test. The test compares a specified column from each relation, so that if θ is one of the comparisons $=, \neq, >, \geqslant, <, \leqslant$ then:

$$R[A \ \theta \ B]S = \{r^\frown s \mid r \in R \ \& \ s \in S \ \& \ (r[A] \ \theta \ s[B])\}$$

Some examples of joins are shown in figure 2.4. When indicates a comparison for equality, two columns in the result will be identical. A projection must be used to remove one of them.

Two other operations are included, both of which can be

manufacturers.
A & Co.
B & Co.
C Ltd.

Supplies [1]

parts.	manufacturers.
6BA NUT	A & Co.
6BA NUT	B & Co.
6BA BOLT	A & Co.
6BA BOLT	C Ltd.
4BA BOLT	C Ltd.
2BA BOLT	C Ltd.

Supplies [2 1]

parts.
6 BA NUT
6 BA BOLT
4 BA BOLT
2 BA BOLT

Supplies [2]

Figure 2.3

Various projections of the relation supplies

from figure 2.1.

parts	quantities
6 BA NUT	1000
4BA NUT	0
2 BA NUT	200
6 BA BOLT	500
4BA BOLT	0
2BA BOLT	200

Number-on-hand

parts	quantities	
4BA NUT	0	0
4BA BOLT	0	0

Numbers-on-hand [2=1] {0}

parts	quantities	
6 BA NUT	1000	500
6 BA BOLT	500	500

Number-on-hand [2>1] {500}

Number-on-hand [1=2] Supplies

parts	quantities	supplier	part
6BA NUT	1000	A & Co.	6 BA NUT
6BA NUT	1000	B & Co.	6 BA NUT
6BA BOLT	500	A & Co.	6 BA BOLT
6BA BOLT	50	C Ltd.	6 BA BOLT
4BA BOLT	0	C Ltd.	4 BA BOLT
2BA BOLT	200	C Ltd.	2 BA BOLT

Figure 2.4 Some examples of join.

defined in terms of the others. The division operator is the counterpart of a universal quantifier. Its definition is omitted because it is quite lengthy and its equivalent is not included in the language to be defined. Restriction subsets a relation on the basis of a comparison of two columns. It is defined by:

$$R[A \ominus B] = \{r \mid r \in R \ \& \ (r[A] \ominus r[B])\}$$

(where $\ominus$ is one of the comparisons above).

Restriction can be defined in terms of join, or join can be defined by a restriction of a cross-product:

$$R[A \ominus B]S = (R \otimes S)[A \ominus B']$$

where B' is B increased by the degree of R.

Codd also defines a relational calculus in which queries can be expressed. The variables in the calculus have tuple values, and unary predicates are used to correspond with stored relations. An algorithm is given to form a relational expression from each expression in the calculus. The resulting expression is not intended to be efficient, but demonstrates that the conversion is possible. Less inefficient conversions are discussed by Palermo (Palermo 1972) and Longstaff and Poole (Longstaff and Poole 1974).

2.2 Relational implementations

Perhaps the implementation most closely following the relational algebra is the Peterlee IS/1 system (now referred to as the Peterlee Relational Test Vehicle, PRTV) (Todd 1975). It provides the four set operations, union, intersection, difference and cartesian product (called rather confusingly "join"), together with the project and restrict operators. The restrict operation is generalised to allow an arbitrary test on the contents of a tuple and called "select".

The implementation is based on sorted, largely sequential, files. The tuples in a relation are sorted in the natural way, the first component of a tuple being the most significant. A compression technique is used so that if a tuple differs from the previous one only in its low-order fields, the unchanging high-order fields are omitted. The remaining fields are also compressed. The complete relation may occupy a number of physical blocks in the file and an index is kept, showing the range of tuple values in each block.

The set operations are implemented by variations on the symmetric merge (Knuth 1973). For union, tuples occuring in either operand are produced, reading the files so that the result is properly ordered. For intersection, only tuples

occurring in both operands are produced. When evaluating a compound expression composed of these operations, a co-routine evaluation technique is used so that the complete result of a sub- expression need not be stored. A tree of co-routines is constructed corresponding to the form of the expression. Execution is initiated at the root of the tree and each operation calls on its operand routines whenever the next block of tuples is needed.

Join operations and most projections cannot participate in the coroutine tree. In a join operation, each member of the first set must effectively be compared with every element of the second set, so that the appropriate members of the cross-product can be formed. Its operands must therefore be evaluated completely and stored so that they can be re-scanned. A complete re-scan can be avoided if the operands are already sorted on the fields mentioned in the selection criterion. In other cases either the operands must be sorted first, or a complete cross-product must be generated. No details are available on how the choice of method is made.

A projection is implemented by appropriately extracting the result fields. Unless only the lowest order fields are removed by the projection, this will produce unsorted output, so that again a complete result must be stored and then re-sorted.

Titman has reported on a system using very similar techniques (Titman 1974). The processing uses collate and sort operations on sequential files. By storing only binary relations in compressed form, even relations with a large cardinality can be retrieved with a single movement of a disk arm, so giving good performance on small batches of input. Titman regards his use of relations as a data-base implementation method, and not primarily as a way of disguising the representation.

The McAims system at MIT (Strnad 1971) was one of the first systems to use n-ary relations. It manipulates them with operations from the relational algebra. The implementation, however, allows for arbitrary storage techniques, by defining just the operations that the representation must support to interface with the remainder of the system. Again although the basic processing technique is collation, the general nature of the system allows for other methods.

The SEQUEL language is a special purpose query language, but closely corresponding to the relational algebra (Chamberlin 1974). Its implementation makes much greater use of random access than PRTV or Titmans system (Astrahan 1972). It utilises a storage system called XRM (Extended n-ary Relational Memory) (Lorie 1974), in turn based on a binary relational storage system. Each tuple of an n-ary relation is stored in a random-access memory and can be located by a

unique "tuple identifier". Access to the tuples is provided by keeping one or more binary relations as indexes for frequently used domains. A binary relation is used to associate with a value the set of the identifiers of the tuples where the value occurs. A mixture of hash-coding and ordered lists is used to give fairly rapid retrieval of all elements in the second column of a relation, given a value in the first column.

The representation for the relation "supplies" might approximate to figure 2.5. The tuples are stored as data in random access storage. Access is provided by what is logically a binary relation, in this case giving tuples with a particular value in the "parts" field. The set of pointers to the tuples (tuple identifiers) for, say, a "6BA NUT" are chained together, and the head of the chain is accessed by hashing the string value "6BA NUT".

Optimisation

The PRTV system optimises relational expressions in approximately source form. The intent is to move restriction (or selection) operations as early as possible in the computation, and project operators as late as possible. It is obviously advantageous to perform restrictions at an early stage, as these produce subsets of the data. Such a test is therefore applied as soon as the

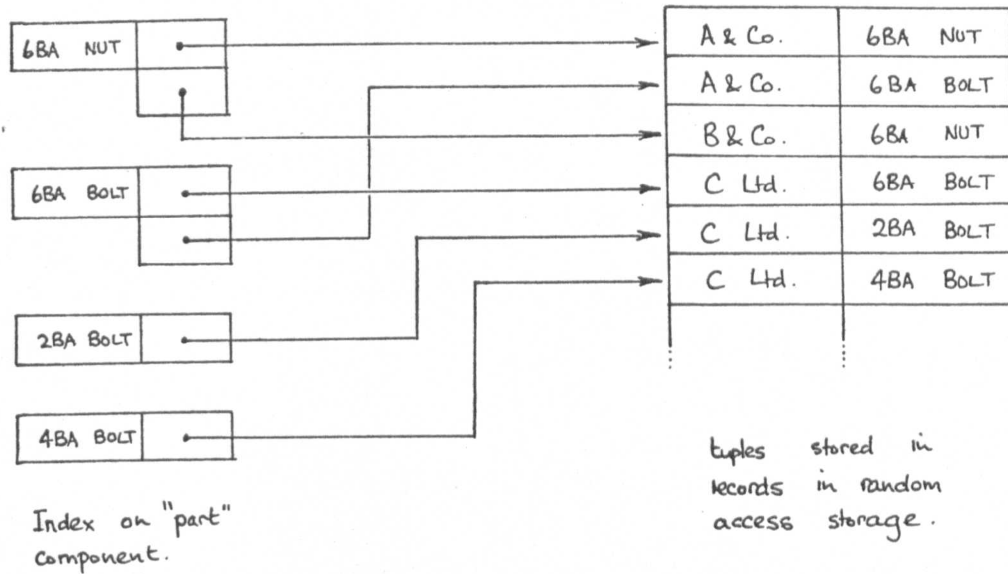


Figure 2.5

Schematic representation of Supplies in XRM

data it mentions is brought together by a join. Project operations usually imply a sort and if they do, their execution is delayed. Two or more projections may then be combined into a single projection. This optimisation should reduce the amount of sorting needed, by sorting when the data is smallest and combining sorts where it is possible. The principles are given by Hall and Todd in (Hall and Todd 1974) but their effectiveness has not been reported. Hall also describes a method for common sub-expression elimination (Hall 1974).

The optimisation algorithm used in the implementation of SEQUEL is given in some detail. This is intended to make good use of the available binary indexes in obtaining the required subsets of a relation. In a paged multi-user system retrieval costs cannot be accurately predicted, so the algorithm attempts to minimise the number of tuples retrieved rather than minimising the overall cost. The description given is quite complex, but the outcome is that where a test on a relation takes the form "column-name=value" and an inversion on the column-name exists (as a binary relation), the inversion is used to immediately locate the subset of tuples which satisfy the test. When simple tests are combined by "and", "or", etc., these operations can sometimes be implemented by merging the sets of tuple identifiers obtained from the inversions.

Bracchi, Fedeli and Paolini (Bracchi 1974) describe a scheme which provides n-ary relations at the user level, but which uses binary relations at the system level. N-ary relations (and also hierarchical structures) can be defined in terms of combinations of binary relations. They outline an optimisation procedure not dissimilar to that used in SEQUEL. Another optimisation procedure for queries, this time based on the entity set model of data (Senko 1973) is described by Ghosh and Astrahan (Ghosh 1974). The different nature of the model makes it difficult to relate this to the previous discussion, but basically it uses a cost function to choose the preferred access-path.

Of these optimisation methods, the simplest is that described by Hall and Todd. It has the great advantage of working entirely in terms of the source program. The other algorithms tend to be more difficult to appreciate because they introduce (and consequently can take advantage of) more details of the representations used.

Update

Relational systems appear to find update difficult. The Peterlee PRTV system implements an assignment statement, so that one or more complete tuples can be added to a relation R by executing the assignment: R=R+Newtuples. (The + sign is used for union). Tuples can be deleted in much the same

way using a set difference operation. A more traditional selective update of a tuple, for example to alter the number-on-hand field of a particular part, can only be done by a complex relational expression, or by the addition of a subroutine written in PL/1. Todd (Todd 1975) remarks that the update of relations is not sufficiently well understood to allow a general update facility.

Titman's system keeps changes in separate files showing the additions and deletions to the master relations. The master relation and the changes are merged at each retrieval, and periodically the data-base must be re-organised so that accumulated changes can be incorporated in the master files.

Other systems concentrate entirely on retrieval, or make only passing reference to the possibility of update. A reason for the difficulty may be related to that suggested by Heath (Heath 1972). Arbitrary changes to components of a tuple can cause side effects. For example if the parts field of a tuple in the relation "number-on-hand" were altered, it might create two numbers on hand for the same part. It will certainly upset the sorting order of the tuples and might also create two identical tuples one of which should be deleted. To prevent such effects, the relation should be in third-normal-form and no assignment can be permitted to fields which are part of the key. None

of the relational systems mentioned restrict relations sufficiently to detect this.

2.3 Other systems

In "Data Semantics" (Abrial 1974) Abrial describes a somewhat different data model which is relevant in this context. With the relational model, the existence of an object is essentially implied because data is stored about it. For example the existence of a 4BA NUT is implied by the occurrence in the relation Number-on-hand of a tuple with "4BA NUT" as a part component. The values from a domain such as "parts" which are actually mentioned in a relation are called by Codd the "active domain". Abrial makes the existence of an object such as a part explicit by including it in a "category" (a set) of parts. An object in a category does not necessarily have an external identification and is just known to be distinct from other objects.

Properties are attached to objects using binary relations. For example two binary relations could relate parts to their descriptions (the description being a string such as "4BA NUT") and their numbers-on-hand. Both these relations will in fact be functions, one description and one number-on-hand being associated with each part. Diagramatically, this is represented in figure 2.6. The categories are shown as large dots and the relations as lines between them.

The labelled arrows represent "access-functions" and two are associated with each binary relation. That labelled "number-on-hand" gives the quantity associated with each part and the inverse "parts-in-quantity" gives all parts with a given number-on-hand. An access function for a binary relation maps one set to a powerset of the other. In Abrial's model the access functions can be constrained to produce result sets with given maximum and minimum cardinalities.

In the example, each part will have only one number-on-hand. The maximum cardinality of the access-function will therefore be unity. The inverse will be unconstrained. Both the access-functions "description" and "part-with-description" might be expected to have maximum and minimum cardinalities of unity, all parts having a unique description. (However, when the check for minimum cardinality can be made is not clear).

A ternary relation such as:

$$\text{delivery-delay} \subseteq \text{manufacturers} \times \text{parts} \times \text{delays}$$

probably needs to be broken down as shown in figure 2.7, inventing a new category called "supply". An object in the supply category with a manufacturer m and part p represents the ability of manufacturer m to supply part p. Each of the

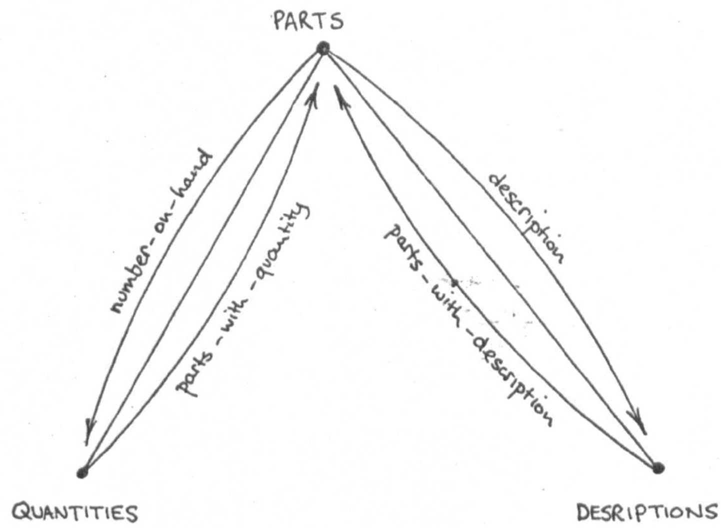


Figure 2.6 Relations between

parts,quantities and descriptions.

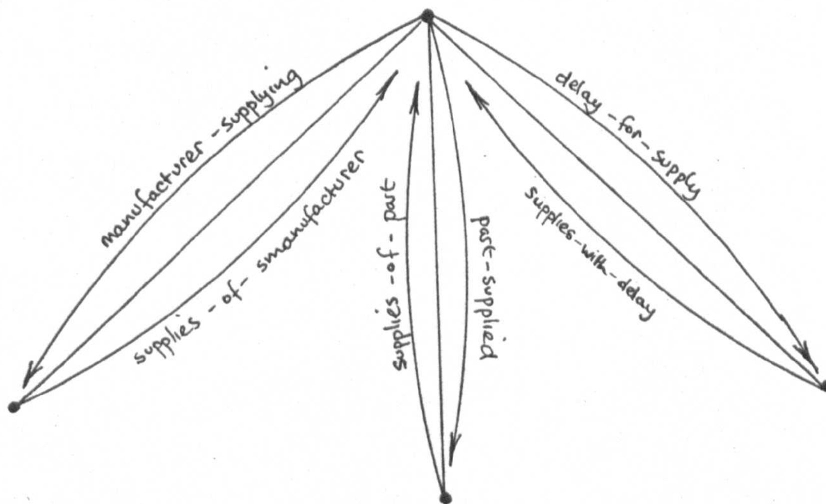


Figure 2.7

Possible breakdown of "delivery-delay"

access-functions "manufacturer-supplying", "part-supplied" and "delay-for-supply" will produce at most one value.

As pointed out by Sharman (Sharman 1975) there will be a close correspondence between the objects in a category (such as supply and part) and the tuples in the corresponding n-ary relations of a third-normal-form description. The outgoing access functions will correspond approximately to domain-names in the relation. The inverse access-functions do not have a relational equivalent but correspond to possible indexes in SEQUEL for example. An advantage of Abrial's model is that it caters more naturally for objects which have more than one identification (personnel numbers and national insurance numbers for employees for example). On the other hand, the restriction to binary relations means that the constraint that given a manufacturer and a part, only one delivery delay is possible, is not directly shown.

To express the operations on the data, Abrial uses a PLANNER-like language (Hewitt 1969), with a very powerful evaluation mechanism. This makes it easy to express the processing when modelling a data-base system, but there is a considerable gap between the facilities provided by PLANNER and those of, say, Cobol, which might be used to implement a final working version.

Predicate Calculus systems

The author is not aware of any systems which implement the relational calculus directly. (Although a number of algorithms were mentioned earlier for converting the calculus to algebraic expressions, and MORIS (Bracchi 1972) does use a calculus-like language). There are, however, some systems based on the predicate calculus. These were not intended for data-base work and so may seem somewhat out of context here. They are described briefly because they have certain features in common with relational systems and the implementation of ABSET in particular strongly influenced the modified relational language defined later.

The ABSYS system (Foster 1968) was designed for experiments in problem solving and evaluates a subset of the predicate calculus. It does this by setting up a collection of "states". Each state gives values to variables so that they satisfy a given predicate expression. For example the expression: x+y=6 <u>and</u> y=5 is only satisfied in a state where x=1, y=5. There is an obvious correspondence between a state and a tuple in a relation, and between a set of states and the relation itself.

The processing method reflects the fact that the evaluation order is not specified. The system attempts to satisfy each conjunct in turn. If the conjunct cannot be satisfied

immediately, it is attached to each variable it uses which currently does not have a value. In the example, for instance, it might try first to satisfy x+y=6, but as there is an infinity of possible pairs of values, the expression is attached to x and y and another conjunct tried. The expression y=5 can be satisfied immediately by setting y to 5. This causes the expression x+y=6 (attached to y) to be re-examined and when y=5 there is only the single possibility that x=1. The effect of this "sequencing" process is to sort the conjuncts into an order so that the final set of states is built reasonably efficiently. It performs much the same function as a conversion algorithm between the relational calculus and the relational algebra.

The appearance of "or" causes two states to be created. For example in x+y=6 or x+y=7 one state is used to keep the values satisfying x+y=6 the other for values satisfying x+y=7. Recursion can be used to set up a larger number. For example, the expression (slightly paraphrased):

```
mem([1,2,3],x)
    where mem(l,x) =  (hd(l)=x or mem(tl(l),x))
```

would cause three states to be set up, with x taking on respectively the values 1,2,3. Values from the states which satisfy an expression can be used to form a set.

The overall effect is therefore very like an evaluator for

the relational calculus, although reflecting the greater freedom allowed in forming expressions. In particular identical processing need not be performed on all states and the presence of recursion means that all states have to be processed more or less in parallel to avoid infinite loops. Execution tends to be breadth-first, while relational evaluators can work depth first to minimise the amount of intermediate storage used.

The later ABSET system (Elcock et al.1971) is linguistically smoother, but does not include the facility for generating multiple states. The authors report that the evaluation order is critical in determining the number of intermediate states which are created, and it was found difficult to control this.

That predicate logic can be treated as a programming language is shown by Kowalski (Kowalski 1974). In principle his system is similar to ABSYS, but using a much more meagre syntax. As well as satisfying assertions, it uses a resolution method (called SL resolution) to produce counter-examples showing that a set of assertions are unsatisfiable. At an abstract level, the executions are very similar to those of ABSYS and Kowalski also comments on the need for a separate means to indicate the execution order.

Ideas from these direct implementations of predicate logic, particularly the analogy between a relation and a set of states, have been drawn on to overcome a number of the difficulties encountered with standard relations.

CHAPTER 3

LANGUAGE F

Earlier we considered some program transformations connected with the choice between keeping items of data in storage and computing them as the need arises. This chapter describes a language based on the use of relational operations and introduces the way that the transformations can be accomplished. For this purpose the standard relational algebra turns out to be less than ideal and in the first section we will look at some of its drawbacks. Then, in section 3.2, the alternative language (designated Language F largely for historical reasons) is introduced. Although this is very similar in principle, it was developed to overcome some of the implementation problems. The new language is close to conventional predicate logic, effectively using variables to name the columns in a relation instead of domain numbers. This has the advantage that the program can be manipulated using the properties of the logical operators rather than their more complex relational equivalents. Expressions are allowed which contain both relations and arbitrary functions or arrays. This means that the language can be used to describe common storage organisations, something outside the scope of a purely relational structure. Section 3.3 gives some examples to show how this is done and how the descriptions

can correspond to possible physical realisations. The discussion shows that, if relations of the type used by Abrial are allowed, we can produce fairly simple definitions of various representations. Section 3.4 then introduces the method of processing and explains the reasons for choosing to implement a compiler rather than a complete data retrieval system.

3.1 Problems with Codd's relations

If we express programs in terms of operations on relations, the program transformations become transformations on relational expressions. The first problem we encounter is that the relational algebra is not very easy to manipulate. In large measure, this stems from its use of ordinal numbers to identify the columns in a relation. For example we find that the relational join operator does not commute. In general R[A=B]S will be different from S[B=A]R. The result of the first expression is a subset of R ⊗ S, and each tuple has components from R occuring first, followed by the components from S. In the result of the second expression the components of S occur first. The two result sets contain identical data, but their columns are in a different order. We can obtain an equivalence between the two expressions by adding a project operator to re-order the columns, but the form of the resulting equivalence is far from simple.

For exactly the same reason the join operator is not associative and again rather involved projections must be introduced to maintain the correct column ordering. The complexity of the transformations need not itself be insuperable in a mechanical processor, but the problem is made worse because projections are introduced and the project operator does not have a very efficient implementation. In general the result of a projection will have fewer members than the original relation (figure 2.3 showing some examples). In view of this, consider how the projection Supplies[2] might be implemented. The result is the set of parts supplied by at least one manufacturer. First we might scan the members of Supplies, creating from each tuple a tuple for the result. This removes the manufacturer column, leaving the list of entries in column 2. As figure 2.1 shows, the list may contain duplicates which have to be removed. Probably the best way to do this is to sort the list, so that when duplicate members are brought together they can be eliminated. The complete process is rather a lengthy one and should not be performed unnecessarily. In particular there is no need to carry out duplicate elimination if it can be guaranteed that the original relation and the result of the projection always have the same number of members. This special case clearly applies if the projection only serves to rearrange the columns as it does in the projections introduced during transformation.

Certain other projections can also be dealt with in this simplified way. For instance a common type of join is illustrated by:

Number-on-hand[1=2]Supplies.

The result (shown in figure 2.4) necessarily has two identical columns. Presumably, later on in the processing, one of these columns will be removed by a projection. While this operation will actually subset the columns, it cannot change the number of elements in the relation, so again there will be no need to remove duplicates from the result. As a further example, the projection Number-on-hand[1] must have the same cardinality as Number-on-hand itself because the original relation is always, in fact, a function.

Consequently, if unnecessary sorts are not to be introduced, we must detect and keep track of these special case projections. This was found to be a more difficult task than eliminating the root of the problem, particularly the dependence on column ordering.

A rather less tangible reason for modifying the language is that relations were originally designed to provide a standard description of the data, deliberately avoiding any suggestion of a particular storage structure. In spite of this, the tacit assumption is often made (referred to, for example, by Bracchi (Bracchi 1974)) that a collection of

relations in third normal form also gives its representation in storage. For example the relation:

part-data (part-number, cost, number-on-hand)

could well be represented by a file with three fields part-number, cost and number-on-hand, with an index on the key field part-number (which has been underlined). This is obviously one possibility, but by no means the only one. There are many other file organisations using secondary indexes, heirarchies and networks which cannot be related to the abstract relational description in such a direct way. To cover these possibilities it seems essential to have the notion of an array or function in addition to that of a relation.

Another related aspect is that of update. It was mentioned earlier that relational systems have some difficulty with this. The only type of assignment which is naturally provided on relations is the addition and deletion of their members. Many updates, which typically alter the value of some data item, do not fall into this category and we need arrays to express them conveniently.

3.2 Language description

The input language we will consider in the following chapters is shown in figure 3.0. Although it has the form of a functional calculus and appears to be very different from the relational algebra, the interpretation it will be given is quite similar. Like the relational algebra it is not intended for direct use in a high-level data-base system. For this purpose it would probably be necessary to provide a rather heavier syntactic disguise, such as the very English-like syntax used by Bracchi (Bracchi 1973). The syntax shown is intended to convey the style of the language only. It contains only a few of the large number of possible operations and for simplicity does not indicate their priorities. The conventional precedences will be observed to resolve the resulting ambiguity.

The meta-syntax used in figure 3.0 is described in appendix A. The only features which need comment are the use of [ ] to enclose an optional phrase and the symbol $\underline{|}$ which stands for the terminal symbol |.

The language evolved from a simple predicate logic containing the connectives "and", "or" and "not". Together with an existential quantifier and some built-in predicates like "greater-than", expressions can be constructed which are equivalent to those in the relational algebra. For

```
definition  ::= function  [parameters]  ≡ valued-expression |
                predicate [parameters]  ≡ logical-expression
result-to-print  ::= set-expression
set-expression  ::=  {parameters | logical-expression }   |
                     function   [arguments]          |
logical-expression  ::= predicate  [arguments]  |
                        arguments in set-expression  |
                        not logical-expression       |
                        logical-expression and logical-expression  |
                        logical-expression or  logical-expression  |
                        valued-expression comparison valued-expression|
                        some   set-expression
comparison  ::=    =|>|≥
valued-expression  ::= constant  | variable  |
                       function [arguments]  |
                       the  set-expression     |
                       number  set-expression  |
                       sum(set-expression, function-name)
parameters  ::= parameter | (parameter-list)
parameter-list ::= parameter | parameter,parameter-list
parameter ::= variable  |  variable:set-expression
arguments   ::= valued-expression | (valued-expression-list)
valued-expression-list::= valued-expression |
                          valued-expression, valued-expression-list
```

Figure 3.0 Syntax for language F

example the relational expression:

supplies [2=1] out-of-stock

can be transliterated very approximately as:

Supplies(m,p) <u>and</u> Out-of-stock(p).

The result of the relational expression is shown in figure 2.4. Assume that "Supplies" is a predicate which is satisfied only by members of the relation "supplies", and "Out-of-stock" is only satisfied by members of the relation "out-of-stock". The complete predicate expression shown will be satisfied if the variable "m" is set to the manufacturer component of a tuple in the result of:

supplies[2=1]out-of-stock

and p to a part component of the same tuple. The predicate expression will be false for pairs of values (m,p) which do not occur in one of the result tuples.

Similarly, we can replace relational expressions containing union and difference by logical expressions containing <u>or</u> and <u>not</u>. For instance we might write:

Out-of-stock(p) <u>or</u> Obsolete(p)

as an approximate replacement for the relational expression:

out-of-stock ∪ obsolete.

Again we assume that "Out-of-stock" and "Obsolete" are

satisfied respectively by members of "out-of-stock" and "obsolete". (Although there is no rigid rule, predicates will often be distinguished from functions and sets by an initial capital letter.) The predicate expression shown will be true if "p" has as a value a member of the result of the relational expression. It will be false otherwise.

The logical operators <u>and</u>, <u>or</u> and <u>not</u> in the language approximately cover the relational operators "join", "restrict", "cartesian product" and the set operations. However instead of acting directly on relations, they act on predicate expressions which are satisfied by relations.

The replacement for the project operator will be dealt with shortly, but as has been mentioned, its general implementation involves quite lengthy computations. This motivated the introduction of functions which produce values other than logical ones. Using only predicates, the definition of "Low-on-stock" in terms of "Number-on-hand" might be written:

Low-on-stock(p) ≡ (∃q) (Number-on-hand(p,q) <u>and</u> q<10).

The quantifier, whose operational equivalent is a projection, can be eliminated if a function "number-on-hand" is used in place of the predicate:

Low-on-stock(p) ≡ number-on-hand(p)<10.

During the processing, the quantifier is re-introduced, but only in a restricted form which is much easier to implement.

Expressions standing for sets of values are also allowed. Internally, sets and predicates behave almost identically, so that in effect, functions can return predicate values. Sets were introduced partly because they are more natural in some contexts than are predicates (for instance to indicate the domain of a function) and partly because they were needed in the description of representations.

A set constructor is provided. For instance we can write:

{(m,p) | Supplies(m,p) and Out-of-stock(p)}

to mean the set of pairs (m,p) which satisfy the predicate expression. The result is the relation:

(supplies[2=1]out-of-stock)[1,2]

(where the additional projection is needed to eliminate a duplicate column).

The set contructor acts in much the same way as a lambda expression, the variables "m" and "p" being bound by their appearance before the |.

In spite of their similarity, predicates and sets differ in

that the system assumes that a set is stored as a list of its members while a predicate is stored as a logical array. This will be examined in greater detail in the next section.

The language contains a number of operators which act on sets. The general existential quantifier is included as a function <u>some</u> which takes a set argument. It gives the value <u>true</u> if the set contains at least one member. For instance in place of:

Can-be-supplied(p) ≡ (∃m)(Supplies(m,p))

or relationally:

can-be-supplied=supplies[2]

we can define:

Can-be-supplied(p) ≡ <u>some</u> { m | Supplies (m,p)}.

The syntax is somewhat cumbersome, but was chosen to match the structure used within the implementation and because it generalises more easily. A practical query language would probably contain a large number of similar operators acting on sets or functions. These correspond to English words like "a", "the", "all", "most", "largest", "sum", "product", "average" and so on. A few have been added to give an insight into their requirements. The function <u>number</u> produces a count of the members of a set and the function <u>sum</u> sums the results of applying a function to a given set of arguments. The function <u>the</u> is similar to the iota operator of predicate calculus and extracts the only member

of a set. The way that the, number, sum and some were included was influenced, with an eye to future extension, by Cresswell's book (Cresswell 1973). In "Logic and Language" he expresses the semantics of English words as operators on lambda expressions in a related way. To keep the language small, some of the obvious operators (for instance set union) have been omitted. The implementation, in fact, supports a wider language than that discussed.

New predicates and functions can be defined in terms of known ones in the normal way. All such definitions have global scope. For example, "cost-on-hand" can be defined by:

cost-on-hand(p) = cost(p) * number-on-hand (p).

Definitions may not be recursive. Although some examples of recursion can apparently be handled, we will not consider them here.

A set expression standing on its own is understood to mean a set of values to be printed, and thus to represent a request for information. The expression should have no unbound variables.

To illustrate the style of the language, some simple definitions and queries might be expressed as follows:

1. Define Out-of-stock in terms of number-on-hand:

   Out-of-stock(p) ≡ (number-on-hand(p)=0).

2. Find the set of parts used in assembly A:

   {p| Uses(A,p)} .

3. Find the parts used in A which are either out of stock or obsolete:

   {p| Uses(A,p) <u>and</u> (Out-of-stock(p) <u>or</u> Obsolete(p))} .

4. Find the British manufacturers who can supply any out of stock part:

   {m| British(m) <u>and</u> <u>some</u> {p| Out-of-stock(p) <u>and</u> Supplies(m,p)}}

5. Define the components of an assembly as the parts it uses:

   components(x) = {p| Uses (x,p)} .

6. Find the number of components of A which cost more than 100p each:

   <u>number</u> {p| p <u>in</u> components (A) <u>and</u> cost(p) > 100}.

7. Find any manufacturers who can supply any of the components of A:

   {m:mfr|<u>some</u> {p|p <u>in</u> components(A) <u>and</u> Supplies(m,p)}} .

8. Find manufacturers supplying all components of A:

   {m:mfr| <u>not</u> <u>some</u> {p|p <u>in</u> components(A) <u>and</u>
   <u>not</u> Supplies(m,p) }} .

   Note how the variable m is bound to the set "mfr", representing the set of all manufacturers.

9. Find manufacturers who give a discount on all components of A:

   {m:mfr|<u>not</u> <u>some</u> {p| p <u>in</u> components(A) <u>and</u>

not (Supplies(m,p) and
discount(m,p)>0) }} .

10. Find manufacturers who give a discount on all parts they can supply to assembly A:

{m:mfr|not some {p|p in components(A) and
Supplies(m,p) and
not discount(m,p)>0 }} .

11. Find the assemblies and the total number of parts used in each:

{(a,n)| a in assemblies and n=sum(qty,uses-of-assembly(a))}.

3.3 Data representation

The next question we have to resolve is the way that the data organisation is reflected in Language F.

In the examples we have used predicates (such as Supplies, Uses and so on) as a standard representation-independent way of referencing the data. We could equally well have standardised on the use of sets and relations. For example 4 we might have written:

{m|m in British and
some {p| p in out-of-stock and (m,p) in supplies}}

where British, out-of-stock and supplies are all sets. This corresponds rather more closely to the equivalent relational expression. Whichever standard form is chosen, definitions must be provided for the predicates or sets used in the

expressions, in terms of structures which actually occur in the stored data. These definitions could be stored on a system library so that, to express the retrieval, it would not be necessary to know what these definitions are, nor even that they exist.

This section gives a number of examples to illustrate the sort of definitions which could be used to relate the predicates or sets to the actual data structures stored. This shows the way that a data organisation is modelled in the language. The examples illustrate that the amount of detail assumed about the representation is roughly the same as that in Hoare's abstract data structures (Hoare 1972a).

1. Following Hoare's use in "Notes on Data Structuring" consider the following type description:

```
type part-file   = sparse powerset (part-record)
type part-record=(partnumber:string,
                  number-on-hand:integer,
                  cost:integer).
```

A part file consists of a set (sparse because most of the possible members are probably absent). Each part-record consists of a cartesian product: a partnumber, a number-on-hand and a cost.

Figure 3.1 shows one possible physical representation of this simple sequential file. Many other representations could fit this type description. The fields shown could appear in a different order, be non-contiguous or even be kept on different physical devices without in any way effecting the logical characteristics of the representation.

Further, the records could be stored contiguously in storage or they could be chained together (in one or both directions) by explicit pointers. However in representations like these, while all the elements in the set can be found by reading through the sequence, there is no direct way to determine whether a particular record is in the set or not. This must be done by a sequential search.

If we assume that retrieval programs treat the data as if it were a collection of relations, then we would need to store the following definitions in the system library:

partnumbers = {(p,n) | partnumber(p)=n }

numbers-on-hand= {(p,q) | number-on-hand(p)=q}

costs = {(p,c) | cost(p)=c }

parts =part-file

The first three definitions relate the sets assumed by the program and the three selector functions in the abstract data structure description of the representation.

part-number	number-on-hand	cost
6BA NUT	1000	1
4BA NUT	0	2
2BA NUT	200	3
6 BA BOLT	500	1
4 BA BOLT	0	1
2BA BOLT	200	2

Figure 3.1 Sequential organisation of part-number, number-on-hand and cost.

Part-number (parts , part-numbers)

parts	part-numbers
PART1	6BA NUT
PART2	4BA NUT
PART3	2BA NUT
PART4	6BA BOLT
PART5	4 BA BOLT
PART6	2 BA BOLT

Figure 3.2 The relation part-number.

There is a direct correspondence between the set "parts" and the sparse powerset "part-file" in the representation. Because the representations of sparse powersets most commonly encountered are sequential, and do not allow for direct testing, the system assumes this method of storage.

If the program referred to the data through predicates a very similar set of definitions would be needed:

Partnumber(p,n) ≡ partnumber(p)=n
Number-on-hand(p,q) ≡ number-on-hand(p)=q
Cost(p,c) ≡ cost(p)=c
Part(p) ≡ p <u>in</u> part-file.

In place of the four predicates or sets we could construct from the data a more conventional single relation:

{(n,q,c) | <u>some</u> {p | partnumber(p)=n <u>and</u> number-on-hand(p)=q <u>and</u> cost(p)=c <u>and</u> p <u>in</u> part-file }}.

Each tuple in this relation has three components, a partnumber, a number-on-hand and a cost. However, because of the relatively complex nature of this definition, the possiblility of using a standard relational description will not be considered.

2. Figure 3.3 illustrates an indexed file containing similar data. The index enables a part-record to be retrieved given its partnumber. Numerous techniques are

available for implementing this index, the traditional method being typified by the indexed - sequential organisation, which uses a mixture of direct indexing and sequential searching. The XRM system (Lorie 1974), on which SEQUEL is based, uses hash-tables and Wedekind describes how relational data can be stored in a generalisation of binary trees called B-trees (Wedekind 1974) (Bayer 1975). Hoare classes all these as implementations of a <u>sparse array</u>, that is an array where the number of elements stored in much less than the number of possible index values.

If the indexing array in figure 3.3 is called "part-with-number", a relation partnumbers' can be defined by:

partnumbers' = {(p,n) | part-with-number(n)=p}.

This should of course be the same relation as is defined by:

partnumbers = {(p,n)|partnumber(p)=n}

as the index is supposed to produce the record containing the appropriate part number. The representation stores the relation redundantly, both the array part-with-number and the collection of fields partnumber being concrete representations of the same relation. Both figure 3.1 and figure 3.3 have the same relational description; they differ only in their representation. For the latter organisation the following definitions would be needed:

partnumbers = {(p,n) | partnumber(p)=n}

= {(p,n) | part-with-number(n)=p}

numbers-on-hand = {(p,q) | number-on-hand(p)=q}

costs = {(p,c) | cost(p)=c}

parts = part-file.

The set "partnumbers" has two alternative definitions. Either may be used in a retrieval.

3. Finally, figure 3.4 shows an organisation of two files representing a number of additional relations. The diagram depicts a physical representation using DBTG-like (Codasyl 1971) structure, sets being connected by chains (or rings) of pointers. The diagram could be re-arranged to use physical contiguity in place of these chains or to display a more hierarchical structure. Such re-arrangements would not effect the type definition.

The definition of simple abstract relations in terms of the functions provided by the type definition of figure 3.4 is given by:

partnumbers = {(p,n) | n=partnumber(p) }

= {(p,n) | p=part-with-number(n)}

numbers-on-hand = {(p,q) | q=number-on-hand(p)}

costs = {(p,c) | c=cost(p)}

costs-on-hand = {(p,c) | c=cost-on-hand(p)}

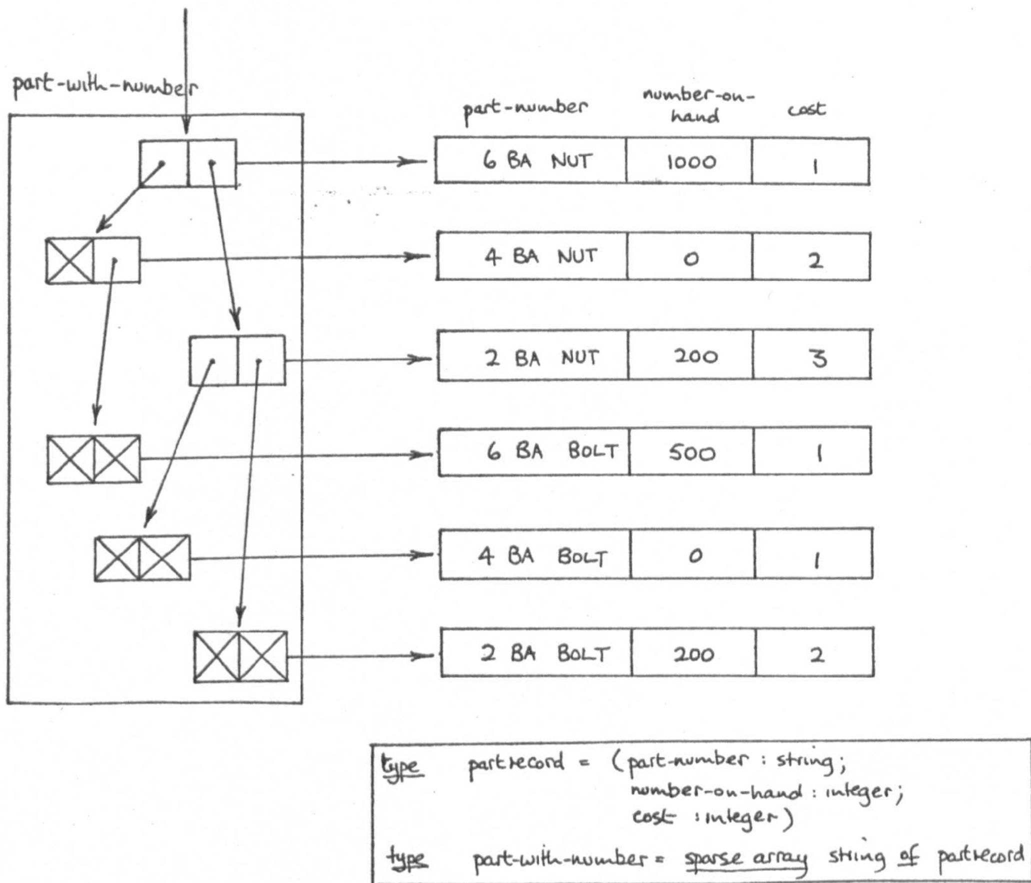


Figure 3.3 Inclusion of part-number index.

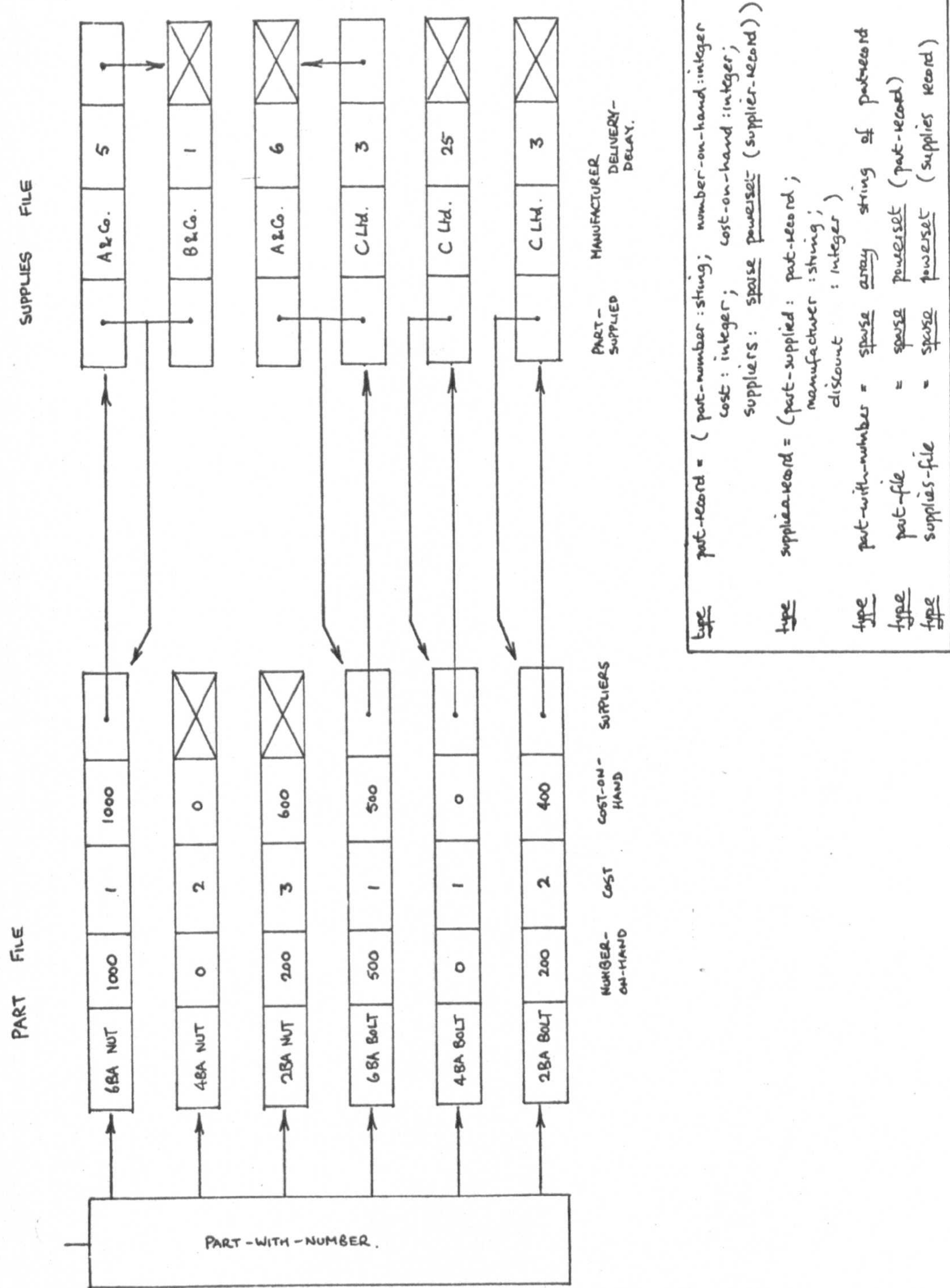


Figure 3.4 Representation with two files.

$$\begin{aligned}
\text{supplies} &= \{(s,p) \mid p=\text{part-supplied}(s)\} \\
&= \{(s,p) \mid s \ \underline{\text{in}}\ \text{suppliers}(p)\} \\
\text{manufacturers} &= \{(s,m) \mid m=\text{manufacturer}(s)\} \\
\text{discounts} &= \{(s,q) \mid q=\text{discount}(s)\}.
\end{aligned}$$

Two relations (partnumber and supplies) are represented redundantly. In addition a field cost-on-hand has been included which could be computed using the equivalence:

cost-on-hand(p) = number-on-hand(p) * cost(p).

These examples show how different data organisations can be seen as concrete representations of relational data. By working with relatively narrow relations and Hoare's abstract data-structures, notably arrays, powersets and cartesian products, the representation can be expressed in relational terms in a fairly straight-forward way.

If a relation is stored redundantly it will have more than one definition. When accessing the data we may choose to use either of the definitions. Also sets will be assumed to be stored sequentially, as this seems to be the most common type of representation. All the members in the set can be retrieved, but there is no inherent provision for testing membership. Because of this assumption, to express the complete range of functions provided by a "powerset" in "Notes on data structuring", two definitions are needed. For example, in:

Out-of-stock(p) ≡ p <u>in</u> out-of-stock-list

≡ out-of-stock-array(p)

if we choose the first definition we can sequence through the set members, but by choosing the second definition, in terms of a sparse logical array, a test for membership can be made.

3.4 Background to the implementation

Having shown how queries and representations can be defined in language F, this section gives a brief introduction to the method of processing.

In our implementation, predicate expressions are used in a number of different ways. To illustrate these, consider the expression:

Supplies(m,p).

This has two free variables m and p. If we give each of them a value then the predicate Supplies can be applied and a true or false result obtained. For example, assuming the data in figure 2.1, it will be true if m="A&Co". and p="6BA NUT". Instead, knowing neither of the variable values, we could draw up the set of <u>all</u> values of m and p so that the expression is true. The data could be arranged as a table like that in figure 3.5. This is very similar to the table of figure 2.1 except that now the columns are headed by

variable names rather than domain names and there is no significance in the column ordering. Such a table of value-assignments is like a relation, but generalised to allow arbitrary column names.

Now given a logical expression such as:

Out-of-stock(p) <u>or</u> Obsolete(p)

we can similarly use it in more than one way. Given a value for p, it can be evaluated conventionally to produce a logical result. Alternatively we could find all the values of p which satisfy the expression. To do this we could first find all the values which satisfy Obsolete(p) and all the values which satisfy Out-of-stock(p), and then take the union of these two sets. Because the expression contains a free variable, the <u>or</u> is implemented like a relational union, obtaining its result in the form of a table of values for p.

To find all values for m and p which satisfy:

Supplies(m,p) <u>and</u> Out-of-stock(p).

we can proceed in much the same way. First we find all the values of m and p which satisfy the first operand (figure 3.5) and all values of p which satisfy the second (figure 3.6). We then combine these tables in such a way that any entry in the result satisfies the complete expression. The

m	p
A & Co.	6 BA NUT
A & Co.	6 BA BOLT
B & Co.	6 BA NUT
C Ltd.	6 BA BOLT
C Ltd.	2 BA BOLT
C Ltd.	4 BA BOLT

a) Data satisfying Supplies(m,p).

p
4BA NUT
4BA BOLT

b) Data satisfying Out-of-stock(p).

m	p
C Ltd.	4BA BOLT

c) Data satisfying Supplies(m,p) <u>and</u> Out-of-stock(p).

<u>Figure 3.5 Sample tables of value-assignments</u>

table which is formed is illustrated in figure 3.7, and the operation which constructs it approximates to a relational join.

The same result can be obtained by another method. First the complete table for one operand, say the first, is generated. Then each entry is taken in turn and tested to see whether it satisfies the second expression. If it does then the particular value-assignment satisfies both expressions and can be included in the result. After all entries in the first table have been processed all possible members of the result will have been produced.

In this computation both the suggested evaluation methods have been used, one to obtain a set of possible results from Supplies(m,p) and one to select or reject them according to the truth of Out-of-stock(p). Using this method there is no need to generate the entire table for the second operand. This is useful if the table is very long, and would be essential if it were infinite.

Finally to create the set:

{(m,p) | Supplies(m,p) <u>and</u> Out-of-stock(p)}.

we form the table of value-assignments which satisfy the internal predicate expression as before. Then the columns are ordered according to the list of variables given and the

variable names removed. This constructs the required relation.

Operations can be defined on tables of value-assignments corresponding to each of the logical connectives and to the existential quantifier. This produces an algebra similar to (although differing in detail from) that described by Hall, Hitchcock and Todd (Hall, Hitchcock and Todd 1974).

The next chapter describes how these ideas are applied in the current implementation, which takes the form of a compiler. The compiler translates from Language F to an Algol-like language. If a complete interpreter for Language F were built, it would have the advantage that data could be obtained about the actual number of accesses needed to answer queries on a sample data-base. However, unless the data-base is of a realistic size the results obtained might be deceptive. The difference between a sequential search and an indexed look-up can be quite small if the data is small, but will become much more marked as the data becomes larger. Further, much of the code in a complete system is devoted to supporting routines such as storage allocation and index management. These low-level functions are particular to the detailed physical organisation chosen and are not really relevant to the task in hand. Moreover, unless they are carefully optimised, they can have a significant bearing on the results obtained. By generating

another program we can choose to ignore these implementation details and match the output code to the level of detail assumed in the abstract data structures.

CHAPTER 4

COMPILATION

The process of compiling Language F can be decomposed into two phases, a pre-processing stage and a subsequent code generation stage. The code generator converts a logical expression into a program to construct a relation. However it accepts only a very limited class of expressions, those which correspond exactly to a possible method of constructing a set. For example it will accept:

p <u>in</u> parts <u>and</u> <u>not</u> obsolete(p)

if "parts" is a stored set and "obsolete" is a stored logical function, but it will report failure if given the equivalent expression:

<u>not</u> obsolete(p) <u>and</u> p <u>in</u> parts.

To make this expression acceptable to the code generator we have a pre-processor. This accepts complex expressions, simplifies them, and generates a series of alternative forms for input to the code generator. The alternative expressions it produces come from two sources. First a function may have more than one definition and a second series of possibilities come from applying commutivity, associativity and distribution to logical expressions.

This two stage structure was chosen to keep the

implementation of the operators and the application of transformations to the initial expression as independent from one another as possible. Although the parts are considered separately, it would not be sensible to generate all the alternative forms first and then to apply the code generation algorithm to each. Practically therefore, both stages execute in parallel, the pre-processor producing alternatives on demand.

The programs produced are similar in style to to those used by Hoare in "Notes on Data Structuring". The main aim was that their logic should be easy to follow. Only simple control structures such as *if* statements and *for* loops are used, although it will become clear later that co-routines should also have been included.

The following sections deal in more detail with the compilation process. Section 4.1 considers the pre-processor and the transformations which it applies. Then section 4.2 introduces the method of code generation and subsequent sections consider the rather more interesting implementation of the various operations. The final section, 4.8, discusses how one program can be selected when more than one possibility exists.

4.1 Pre-processing

The pre-processing phase converts complex logical expressions to a series of alternative and simpler expressions. Principally, functions which are not primitive are replaced by their definitions and nested function applications are removed.

1. To remove non-primitive functions, the usual evaluation rule for a function definition (roughly rule "I" in Landin 1966) is applied:

```
. . F(M) . . where F(x)=L
        => . . L where x=M . .
```

This replaces a reference to a function (F) by a copy of the procedure body (L), binding arguments and parameters. For example:

```
Supplies(m,p) and Out-of-stock(p)
              where  Supplies(m,p) ≡ m in suppliers(p)
```

is converted to:

```
m in suppliers(p) and Out-of-stock(p).
```

Notice that for a redundant representation, where a function has more than one definition, there will be more than one possible expansion. For example:

```
Supplies(m,p) and Out-of-stock(p)
```

where Out-of-stock(p) ≡ p *in* out-of-stock-list
≡ out-of-stock-array (p)

produces two expansions:

Supplies(m,p) *and* p *in* out-of-stock-list

Supplies(m,p) *and* out-of-stock-array (p).

Both of the possibilities must be considered in turn by the compiler.

2. In exactly the same way, set definitions can be removed, using the rule:

. . M *in* S . . *where* S={x|L}

=> . . L *where* x=M . .

For example:

p *in* Out-of-stock-list

where Out-of-stock-list = {x| number-on-hand(x)=0}

=> number-on-hand(p)=0.

A set definition is treated as if it were a definition of a predicate, but with a different syntax. The expression: M *in* {x|L} is treated exactly as if P(M) *where* P(x)≡L had occurred.

An example where both functional and set reduction can be applied is:

m <u>in</u> Suppliers(p)

<u>where</u> Suppliers(p) = {x|p <u>in</u> supplied-by(x)}

=> m <u>in</u> {x|p <u>in</u> supplied-by(x)}

=> p <u>in</u> supplied-by(m).

Applying these rules leads to one or more equivalent logical expressions which contain only stored (or other primitive) functions and sets. The process incorporates the representation of the data into the program. A further simplification is then made to eliminate nested functional expressions.

3. When a predicate (other than =) occurs with an argument which is a functional expression, the following transformation is made, P standing for the predicate:

P(f(x)) => (Et)(P(t) <u>and</u> t=f(x)).

Similarly, functional expressions are flattened:

r=g(f(x)) => (Et)(r=g(t) <u>and</u> t=f(x)).

In both the resulting expressions, there can be at most one value of t which satisfies the quantified expression, as in each case the only possible value is determined by the result of a function. A special implementation is given to the existential quantifier when this condition is known to hold, and to retain the special case the ∃ sign is written as E.

The transformations can be justified in the following way. First we replace the expression f(x) by $(\iota t)(t=f(x))$:

$$P(f(x)) \Rightarrow P((\iota t)(t=f(x))).$$

Then removing the iota:

$$P((\iota t)(t=f(x))) \Rightarrow (Et)(P(t) \underline{\text{and}}\ t=f(x)).$$

The latter step is justified by the equivalence given by Carnap (Carnap 1958) which is paraphrased by:

$$P((\iota x)(Q(x))) \equiv (\exists! x)(Q(x))\ \underline{\text{and}}\ (\exists x)(P(x)\ \underline{\text{and}}\ Q(x))$$

(where $(\exists! x)(Q(x))$ means there is a unique x so that Q(x) from (Kleene 1967)). In the transformation, the uniqueness condition has been dropped. We will see the effect of this shortly.

As an example, using the first transformation,

cost(p)>3

is reduced to:

(Er)(cost(p)=r <u>and</u> r>3).

Similarly the more complex expression:

r = cost (p) * number-on-hand (p)

becomes:

(Ex)(Ey)(x=cost(p) <u>and</u> y=number-on-hand(p) <u>and</u> r=x*y).

Historically the transformation was carried out so that functions could use the existing mechanism for predicate

expressions. Since a predicate expression is compiled to a statement, it tends to produce programs composed of simple statements rather than compound expressions. These were found to be easier to understand. It also has the advantage that the equivalent of the iota operator (the) can be dealt with at the same time.

By using just the last step in the transformation, appearances of the can be eliminated:

    P(the{x|Q(x)}) => (Ex)(P(x) and Q(x)).

As a simple example, supposes we encounter the expression:

    cost(x)>4,

where cost is not stored directly, but is defined in terms of its inverse:

    cost(p) = the {c|p in parts-with-cost(c)}.

(A function parts-with-cost is stored, which produces a set of parts given a cost. The cost of a part p is given by the (unique) cost c in whose result set the part occurs).
Replacing the infix > by "greater-than", we get:

    greater-than (cost(x),4)
 => greater-than (the {c|x in parts-with-cost(c)},4)
 => (Ec)(x in parts-with-cost(c) and greater-than(c,4)).

However if we generate all the values of x which satisfy:

(Ec)(x in parts-with-cost(c) and c>4)

not all of them are necessarily soltions to:

cost(x)>4

since we should exclude any for which c is not unique (that is any parts which are recorded as having more than one cost.) Dropping the uniqueness condition from the transformation produces code which may give erroneous output.

The effect of these transformations is to reduce a logical expression to a series of expressions which contain only primitive predicates with variables as arguments, expressions of the form: y=f(x), y in f(x), a restricted existential quantifier and expressions involving some, sum and number. The code generator assembles a program from components corresponding to these forms.

Finally, the pre-processor will, if called on to do so, generate alternative forms of each expanded expression by re-ordering the operands of and and distributing and over or. A complete list of the transformations applied appears in Appendix A.

As an example, given an expression to find the assemblies

using any out-of-stock part:

{a| <u>some</u> {p|uses(a,p) <u>and</u> Out-of-stock(p)}}

together with the definitions:

Uses(a,p) ≡ p <u>in</u> parts-used-by(a)

Out-of-stock(p) ≡ number-on-hand(p)=0

describing how the predicates are represented in storage, the pre-processor potentially produces the pair of expressions:

{a| <u>some</u> {p|p <u>in</u> parts-used-by(a)
<u>and</u> number-on-hand(p)=0 }}

{a| <u>some</u> {p|number-on-hand(p)=0
<u>and</u> p <u>in</u> parts-used-by(a) }} .

These expressions only refer to primitive functions. Each of the atomic expressions has one of the forms listed, and all orderings of the conjuncts appear.

As a more complex example, consider the expression given in chapter 1, to find the assemblies using more than 1000 4BA NUTs a week:

{a|rate-used(use(a,"4BA NUT"))>1000}.

With the definitions:

rate-used(r)=quantity(r)*weekly-output(assembly(r))

use(a,p) =<u>the</u> {r|part(r)=p <u>and</u> assembly(r)=a}

the preprocessor will expand it as follows:

```
{a| quantity(r)*weekly-output(assembly(r))>1000
          where r=use(a,"4BA NUT")}
```

```
=>
```

```
{a|(EwExEyEz)(z>1000 and z=x*y
                 and y=quantity(r)
                 and x=weekly-output(w)
                 and w=assembly(r)
        where r=use(a,"4BA NUT") }
```

removing the nested expressions.

```
=>
```

```
{a|(EwExEyEz)(z>1000 and z=x*y
                 and y=quantity(r)
                 and x=weekly-output(w)
                 and w=assembly(r)
      where r=the {r|part(r)="4BA NUT" and assembly(r)=a }}
```

substituting for "use".

```
=>
```

```
{a|(EvEwExEyEz)(z>1000 and z=x*y
                  and y=quantity(v)
                  and x=weekly-output(w)
                  and w=assembly(v)
                  and part(v)="4BA NUT"
                  and assembly(v)=a } ,
```

applying the rule for <u>the</u> simultanously to the expressions:

w=assembly(<u>the</u> {. .})

y=quantity(<u>the</u> {. .}).

It must be emphasised that, although there are a large number of possible orderings of these conjuncts, it is very unlikely that it will be necessary to generate them all.

4.2 Code generation

The code generator attempts to convert the expressions produced by the pre-processor directly into programs, replacing each logical operator which occurs by a simple code sequence. The code produced must satisfy certain constraints. For instance, a variable cannot be referenced unless it has previously been given a value. The generator checks that conditions such as this are obeyed and if not, the generation process fails and another equivalent expression must be tried.

For data retrieval it is assumed that we usually need to print all the members of a set or relation. If the set is S, then the code to do this could be written in a simple Algol-like language as:

<u>for</u> x <u>in</u> S <u>do</u> write(x) <u>od</u>.

If S is stored, this simple program is an adequate description of the processing. It would not be difficult to

implement such a loop in a standard programming language using any of the sequential representations of sets discussed in section 3.3.

Suppose now that the set to be printed is not stored but consists of an expression. For instance S might be:

{p|p <u>in</u> parts <u>and</u> number-on-hand(p)=0}

A program to print its members can be obtained by first writing it in the form:

```
for x in {p|p in parts and number-on-hand(p)=0} do
  write(x)
od
```

and then applying a series of transformations which remove the complex expression between the <u>for</u> and <u>do</u> in favour of a more elaborate control structure. Each transformation substitutes a simple implementation for the principle operator in the expression.

In the example the first operator to be removed is <u>in</u>. Its implementation requires no more than the substitution of actual for formal variables, removing p in favour of x. This gives:

```
for x|x in parts and number-on-hand(x)=0 do
  write(x)
od.
```

The resulting partly developed program illustrates the general form of the statements which are processed. It can be read: "for all values of x which satisfy the expression: x _in_ parts _and_ number-on-hand(x)=0, execute the statement write(x)". The loop control variable x occurs before the | and a logical expression determining its possible values occurs after it. In general there may be more than one control variable so that in the model form: _for_ x|P _do_ C _od_ x is strictly a set of identifiers. However when x is a unit set we will not distinguish it from its only member. Also there may be no control variables at all, so that x may be empty. The _for_ loop then reduces to an _if_ statement.

To continue with the example, the next operator to be removed is _and_. Its implementation introduces a nest of statements:

```
for x in parts do
 L1: if number-on-hand(x)=0 then
   L2: write(x)
     fi
od
```

The expressions in the loop header and the _if_ clause cannot now be decomposed further and the generation process is complete. The pre-processor will have guaranteed that the set "parts" and the function "number-on-hand" do not have definitions so that these are assumed to be stored. If the

set "parts" were a sequence of records and the function "number-on-hand" a field-name in each record, the program could be implemented quite easily in a standard language.

Looking more carefully at its execution, the for loop sets x successively to all members of the set "parts". At label L1, x will take on every value which satisfies the expression "x in parts". The set of program states at L1 corresponds to the set of value-assignments to "x in parts". The inner if statement receives all these values and produces at L2 only those which also satisfy "number-on-hand(x)=0". The write statement prints this set.

During execution the possible values of x are generated one by one by the loop and each is processed to completion by the loop body before the next is tried. No intermediate set is stored, as the generation and processing of its members are interleaved. This method of organising a calculation involving sets or lists parallels the coroutining of PRTV. An analagous method has been used by Abrams to avoid storing temporary vectors in the evaluation of APL (Abrams 1970). Similar ideas lie behind the "stream processing" functions reported by Burge (Burge 1975) and "dynamic lists" in POP2 (POP2 1971). Because the code responsible for generating the intermediate structure only produces a new element when it has to, Henderson and Morris have called the technique "lazy evaluation".

4.3 Primitive Statements

Recall that retrieval causes processing to begin with a statement which has the form:

for x in S do write(x) od

and the predicate is repeatedly processed to produce a nest of for loops containing only atomic predicates. The three types of atomic logical expression produced by the pre-processor give rise to three types of primitive statement, a standard for loop, a let block and an if statement. In a completely expanded program they can only mention functions and sets which are directly stored or easily computed.

A standard for loop is generated by the transformation:

```
for {x}|x in S do C od
            => for x in S do C od
```

The transformation reflects the fact that, assuming C does not exit abnormally, the resulting code sequence executes C with all values of x satisfying x in S. For the conversion to work the set of variables to be determined (the set preceding the |) must have x as its only member. No code could be generated for example from:

```
for y| p in parts do C od
```

and given this expression the code generator will report failure.

In general, x may be a list of variables, when {x} is understood to mean the set of variables in the list x (which must all be different). For example:

> <u>for</u> {m,p}| (m,p) <u>in</u> Supplies <u>do</u> C <u>od</u>
>
> => <u>for</u> (m,p) <u>in</u> Supplies <u>do</u> C <u>od</u>.

Again the transformation is straight-forward, the final code sequence generating all values of m and p satisfying the expression.

A <u>let</u> block is generated by:

> <u>for</u> {x}|x=e <u>do</u> C <u>od</u>
>
> => <u>let</u> x=e; C.

Since there must be one and only one value of x which satisfies the predicate expression, the <u>for</u> loop, which would otherwise be produced, can be supplanted by a simpler <u>let</u> block. The equality may be reversed so that the unknown variable appears on the right rather than the left without effecting the code, but unknowns may not appear on both sides. A simple example of <u>let</u> is:

```
for q| number-on-hand(p)=q do C od
=>
 let q=number-on-hand(p); C.
```

The final transformation may be used for any predicate function:

```
for {}| P do C od
    => if P then C fi.
```

The set of variables to be determined must in this case be empty, so the loop is a degenerate form with no control variable. Examples are:

```
for {}| Out-of-stock(p) do C od
   => if Out-of-stock(p) then C fi
for {}| number-on-hand(p)=q do C od
   => if number-on-hand(p)=q then C fi.
```

As there is normally special-case code when the set of variables to be determined is empty, it is convenient to make the transliteration between <u>for</u> and <u>if</u> whenever this occurs. In general an <u>if</u> statement may also have an <u>else</u> clause, but following normal practice, it will be omitted when it contains only a null statement.

Although <u>if</u> can be used for the majority of predicates, we have made the assumption that sets are held sequentially. Consequently we will disallow code sequences of the form:

```
if x in S then C fi
```

when S is stored. The danger of allowing such code is that, should it occur in a loop, the set S may be searched many times and the resulting program might be very inefficient. It would be costly for example to attempt to intersect two sets S1 and S2 using:

```
for x in S1 do
  if x in S2 then C fi
od.
```

Since <u>in</u> has to be implemented on a sequence by searching the entire set, the code amounts to:

```
for x in S1 do
  for y in S2 do
    if x=y then C fi
  od
od.
```

By disallowing this code, the compiler is forced to look for an alternative. For instance it may be able to process both sets sequentially in a collate operation.

4.4 Conjunction

The simple example given earlier shows that code for a conjunction can be constructed by inserting one loop within another. In general, if P and Q are two predicate expressions and the variables used in P are a set p, then the transformation takes the form:

```
for x| P and Q do C od
=> for (x∩p)|P do
       for (x-p)|Q do
           C
       od
   od.
```

The outer loop generates all values for any of the unknown variables in x which are used in P (these are given by (x∩p). The inner loop accepts these values and in turn generates values for any remaining variables (those in the set difference x-p) so that Q is also satisfied. The statement C is executed with all values satisfying the conjunction.

As an example consider generating the set of values:

{(x,y)| x <u>in</u> 1..20 <u>and</u> y <u>in</u> 1..10 <u>and</u> x+y=5}.

Taking the principle operator to be the first <u>and</u>, the

generation proceeds:

```
for {x,y}| x in 1..20 and y in 1..10 and x+y=5 do C od

=> for x|x in 1..20 do
     for y|y in 1..10 and x+y=5 do
        C
     od
   od
```

by expanding the and into a nest of loops.

```
=> for x in 1..20 do
     for y|y in 1..10 do
       for {}| x+y=5 do
          C
       od
     od
   od
```

simplifying the first loop header and expanding the second and.

```
=> for x in 1..20 do
     for y in 1..10 do
        if x+y=5 then C fi
     od
   od
```

replacing the central for loop by an if statement.

This is only one of the possible programs (and clearly not the best) which could be used to generate the same set of values. Other orderings of the conjuncts would produce different programs. Discussion of the choice of a suitable order is deferred until section 4.8.

4.5 Disjunction

To print the values which satisfy the expression:

x <u>in</u> 1..5 <u>or</u> x <u>in</u> 7..10

the following program would suffice:

```
for x in 1..5 do write(x) od;
for x in 7..10 do write(x) od
```

It consists of two statements generating values in the individual sub-ranges concatenated together.

In general we make the transformation:

```
for x|P or Q do C od
   => for x|P do C od;
      for x|Q do C od.
```

The loop body C has been duplicated in this construction. If the code is short the duplication is of little consequence. If the code were more lengthy it might be better to create a subroutine containing C and insert a call

in each loop body.

The code sequence is strictly an implementation of a disjoint set union. If there are values which simultaneously satisfy both operands of an _or_, the code will generate them twice. Provided only simple predicate connectives are used the duplication will go undetected and the only effect would be to reduce the program efficiency. But were we to attempt to count the number of items in a set by counting the total number of loop iterations, then too large a result would be obtained. Similarly, when printing the values, some will appear twice. Also, while we have not been explicitly concerned with the order in which values are produced, the code generates all values from one operand first followed by those from the other. It does not interleave the values properly if they come from an ordered set.

The natural implementation of _or_ for this more general case is to use a symmetric merge. A merge requires that the two operands generate values in the same order, but has the advantage that an ordered result is produced and that duplicate values can be removed easily. As was remarked earlier, many relational systems use merge not only to implement _or_, but in different variations for each operation. An interpreter constructed initially to implement the operations on value-assignments was no

exception to this. To use collate operations widely may well be expedient in practice, but it does obscure situations when a simpler method could have been used.

No symmetric merge operations are generated at present. The major problem being that to express the process a pair of coroutines is needed. While the generation of values which satisfy:

out-of-stock(p) <u>or</u> obsolete(p)

might be indicated as:

<u>for</u> p <u>in</u> (out-of-stock <u>or</u> obsolete) <u>do</u> - <u>od</u>

when the operands of the <u>or</u> are expressions, no convenient way to express the process could be found. The use of a "resume" statement as in Simula 67 does not produce very clear code, neither does the alternative (which would be necessary if using a standard high-level language) of simulating this with conventional loop structures. As a temporary measure, the semi-colon is replaced by a comma (to indicate the parallel execution of the two statements) when the necessity for a merge has been detected. Occurrences of its use do not seem to be very common in practice.

The problem with inclusive or does not occur if there are no variables to be determined. We can then use a standard transformation whose general form is:

```
if P or Q then C1 else C2 fi
  => if  P then C1 else
     if Q then C1 else C2 fi fi.
```

To illustrate its use consider the expression:

```
{(x,y)| y in 1..10 and x+y=5 and (x in 1..5 or x in 7..10)}
```

The generation proceeds:

```
for {x,y}| y in 1..10 and x+y=5 and (x in 1..5 or x in 7..10)
do  C  od
 => for y in 1..10 do
      let x=5-y;
        for {}| x in 1..5 or x in 7..10 do C od
    od

 => for y in 1..10 do
      let x=5-y;
        if x in 1..5 then C else
        if x in 7..10 then C fi fi
    od.
```

One may feel in this case that the expansion has gone too far since the program is probably clearer with the or than with its if then else replacement. However a complete expansion is sometimes needed to eliminate compound expressions within disjuncts and so is always carried out.

4.6 Negation

A not operator can only be compiled in a context where there are no variables left to be determined. We cannot, without searching the universe, determine values which do not satisfy an expression. The only transformation used is:

```
if not P then C1 else C2 fi
 => if P then C2 else C1 fi.
```

As an example the expression:

```
{(m,p) | Supplies(m,p) and not (p="4BA NUT" or p="4BA BOLT")
        where Supplies(m,p) = (m,p) in supplies-list }.
```

becomes:

```
    for (m,p) in supplies-list do
      if not(p="4BA NUT" or p="4BA BOLT") then write(m,p) fi
    od
 =>
    for (m,p) in supplies-list do
      if p="4BA NUT" or p="4BA BOLT" then else write(m,p) fi
    od
 =>
    for (m,p) in supplies-list do
      if p= "4BA NUT" then else
      if P= "4BA BOLT" then else write(m,p) fi fi
    od.
```

4.7 Projection and related operations

An equivalent of the existential quantifier must be included to give the same facilities as the project operator in the relational algebra. In the restricted form derived from functional expressions, it is known that at most one value can satisfy the quantified expression, and this produces a very easy implementation. As an example, consider listing the values satisfying

p in parts and r=cost-on-hand(p).

Section 4.1 shows that this reduces to:

(Ex)(Ey)(x=number-on-hand(p) and y=cost(p) and r=times(x,y)
and p in parts).

A program which produces all values of p,x,y and r satisfying the quantified expression at label L is:

```
for p in parts do
   let x = number-on-hand (p);
      let y = cost (p);
         let r = times (x,y) ;
          L: C
od
```

Clearly execution will reach the label once for each value of p and with r containing the required value. Consequently we can just ignore the quantifier and assuming that the set

"parts" contains no duplicates, no two iterations will have the same value of p.

The same should be true, although less obviously, in the example:

Cost(p) *in* 1..10

which, if Cost is defined in terms of its inverse, reduces to:

(En)(p *in* parts-with-cost(n) *and* n *in* 1..10).

(A very similar example was given in section 4.1). Code to execute statement C with all values of p and n satisfying the quantified expression is:

```
for n in 1..10 do
    for p in parts-with-cost(n) do
     C
    od
od.
```

Again the quantifier can be ignored and no duplicate values of p should be produced, although this is not obvious from the code. (Notice, though, that the values of p are not produced in any sorted order).

Consequently the existential quantifier, when it is known that at most one value can satisfy the expression, can be

implemented simply by forgetting it. This contrasts with the more general case. Suppose, for example, we have the expression:

<u>some</u> {m | Supplies(m,p) } <u>and</u> Out-of-stock(p)

giving, as values of p, the out-of-stock parts for which at least on supplier is recorded. It would be more conventionally written:

$(\exists m)$(Supplies(m,p)) <u>and</u> out-of-stock(p).

With suitable representations, we might, using the same method as before, create the program:

```
for p in out-of-stock do
    for m in suppliers (p) do
     C
    od
od
```

and forget the quantifier. However, execution will reach C many times for each value of p and if we printed a list of values of p produced in this way it would contain many duplicates. We can prevent this by branching out of the in-most loop once one value has been found:

```
for p in out-of-stock do
    for m in  suppliers (p) do
        C; goto L
    od
L: od
```

Execution will now reach statement C at most once for each value of p. In examples with just simple connectives and, or, not the advantage of the lazy method of evaluation has been to avoid the need to store temporary sets of values. Here the generation of a set can be prematurely terminated when the existence of the remaining members is of no interest.

Generally we make the transformation:

```
if some S then C1 else C2 fi
   => for x in S do
          C1; goto L
      od;
      C2;
     L:
```

Statement C1 is executed once if a value is found satisfying the test. C2 is executed if no value is found.

Other operations

A number of other operations can be performed in a similar way. For example to count the number of suppliers of each out-of-stock part we could, in similar circumstances, use the code:

```
for p in out-of-stock do
   let r=0;
      for m in suppliers(p) do
       r:=r+1
      od;
     C
od.
```

When statement C is executed, r will contain the number of suppliers of part p. The general implementation is:

```
let r = number S; C
   =>  let r=0;
         for x in S do
             r:=r+1
         od ; C
```

The use of let in the resulting code sequence is of dubious legality as the variable it introduces takes on more than one value. However we regard the loop as part of the construction of its initial value.

In exactly the same way to find the sum of some items we can use:

```
let r = sum (S,f); C
 =>   let r=0;
        for x in S do
          r:=r+f(x)
        od; C
```

Remember the function sum takes two arguments, a set and a function to be applied to its elements. The items to be summed do not necessarily constitute a set.

All these examples clearly have the same form. The function some differs somewhat because its result is a logical value and because, once one value has been found, no more need be considered. Other functions with a similar structure for example all, minimum, one-of, could readily be implemented in the same way by modifying the initial value and the statement in the loop body.

The transformations shown in the appendix are slightly more general than those given here to allow for arrays.

Temporary Structures

Suppose that we need to expand:

<u>for</u> p| <u>some</u> {m|Supplies(m,p)} <u>do</u> C <u>od</u>,

that is we need to execute C with all parts p supplied by at least one manufacturer. The implementation of the restricted quantifier E is to ignore it. If we try the same method and assume that Supplies is stored as a set:

Supplies(m,p) ≡ (m,p) <u>in</u> supplies-list

the <u>for</u> loop reduces to:

<u>for</u> {m,p} | (m,p) <u>in</u> supplies-list <u>do</u> C <u>od</u>.

The code is simply:

<u>for</u> (m,p) <u>in</u> supplies-list <u>do</u> C <u>od</u>.

Certainly C will be executed with p taking on all parts supplied by some manufacturer, since the code goes through the entire supplies file extracting all parts. However each part will be produced many times. We cannot, as in an earlier case, terminate the production of duplicates by a branch, so to eliminate the duplication the values must be stored:

```
let S=empty;
   for  (m,p) in supplies-list do
          S:=S union p
   od;
for p in S do C od.
```

The parts occurring are saved in a temporary set S (implemented, say, by a binary tree). When execution of the first loop is terminated, S contains the set of values required. The second loop executes statement C once for each value found.

The substitution can be generalised:

```
   for {x} | some S do C od
=>  let T=empty;
       for ({y}∪{x})|y in S do
            T:=T union x
       od;
    for x in T do C od.
```

With the functions <u>number</u> and <u>sum</u> it may be necessary to create a temporary array of values. The transformation used for <u>number</u>, introducing an array A is:

```
    for {r}∪{x}|r=number S(x) do C od
 => let A(x)=number S(x);
       for {x}∪{r}|x in domain(A) and r=A(X) do C od.
```

For example, suppose we needed the number of manufacturers who can supply each out-of-stock part.
The set to be printed is:

```
{(p,r) | Out-of-stock(p) and r=number {m|Supplies(m,p)}} .
```

If the representation were:

```
Supplies(m,p) ≡ (m,p) in supplies
```

then we would need to expand:

```
for {p,r} |
   Out-of-stock(p) and r=number {m|(m,p) in supplies}
do write (p,r) od.
```

No code can be generated without first creating an array. The reason for this is that p appears on the left of an in in the second clause. The only possibility would be to scan the set supplies many times, once for each out-of-stock part. So applying the transformation to produce an array, we get:

```
let  A(p) = number {m|(m,p) in supplies};
for  {p,r}| p in domain(A) and r=A(p) do
  if Out-of-stock(p) then
    write(p,r)
  fi
od.
```

Expanding the let according to the implementation of number

gives:

```
let A(p)=0; comment A is a sparse array, and all elements
                  are defaulted to zero.
   for (m,p) in supplies do
        A(p):= A(p)+1
   od;
for p in domain(A) do
  let r=A(p);
    if Out-of-stock(p) then
       write(p,r)
    fi
od.
```

The code could be improved by noticing that A stores the counts for all parts, whereas only those in out-of-stock are used. It would be better to test this condition before storing into the array. The intended transformation with this effect was unfortunately found not to be possible within the current implementation structure.

The complete list of expansions used during code generation are shown in Appendix A.

4.8 Choosing a program

Although many of the alternative expressions produced by the pre-processor will fail to generate a code sequence, there will be some occasions where more than one result is possible.

An important source of alternatives is the ability to take the clauses of a conjunctive expression in any order. For the arithmetic example given earlier:

$$\{(x,y) \mid x \ \underline{in} \ 1..20 \ \underline{and} \ y \ \underline{in} \ 1..10 \ \underline{and} \ x+y=5\}$$

many of the orderings could be used and they are illustrated in figure 4.1. In the diagram each node has been labelled with a subset of the clauses. At I this is the empty set and at O all the clauses are included. Assuming members of these sets are connected by and, each node is associated with a logical expression. Each edge in the figure is labelled by the heading of an if, for or let statement. The various programs to construct the set of interest are represented by a sequence of directed edges from I to O.
For example the original program:

```
for x in 1..20 do
   for y in 1..10 do
      if x+y=5 then C fi
   od
od
```

is represented by the edges through the chain of points 1,2,6,8. The logical expressions at the nodes define the sets of states which occur at intermediate points in the program.

When dealing with stored relations rather than computed ones the possibilities will be limited by the representation. An example which is structurally similar to the last is:

$\{(m,p) \mid$ Out-of-stock(p) and Supplies(m,p) and British(m)$\}$.

Suppose that the predicates are defined in terms of stored structures by:

Out-of-stock(p) ≡ p in out-of-stock-list ≡ out-of-stock(p)
British(m) ≡ m in british-list ≡ british(m)
Supplies(m,p) ≡ m in Suppliers(p).

Both Out-of-stock and British are represented so that they provide the functions of a "powerset". The possible programs (without using a collate operation) are then illustrated by figure 4.2. Where no transformation is possible the connection between two nodes is shown as a dotted line. Only one path between I and O is possible

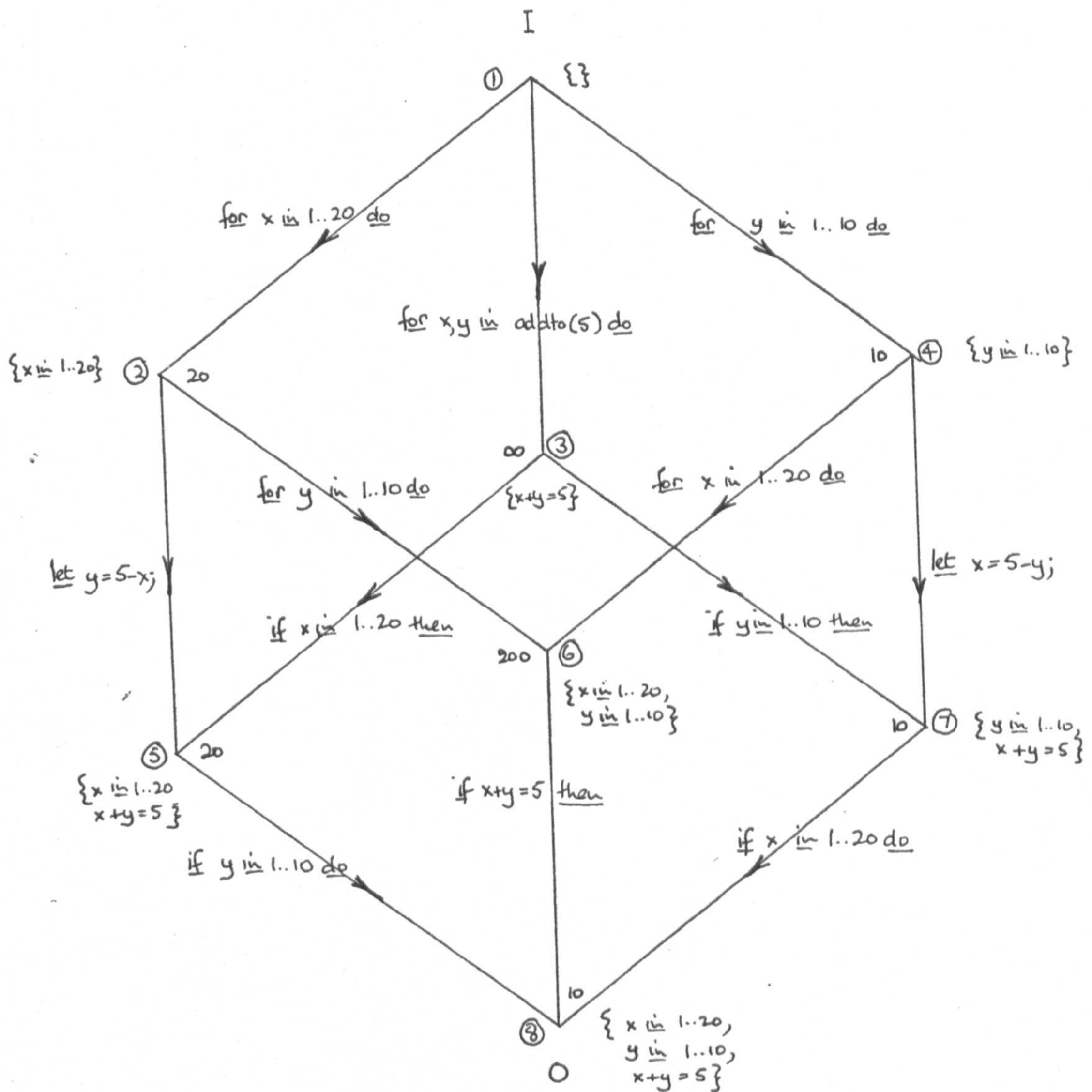


Figure 4.1 Possible programs for x in 1..20 and y in 1..10 and x+y=5.

representing the program:

```
for p in out-of-stock-list do
   for m in Suppliers(p) do
      if british(m) then C fi
   od
od.
```

If instead the predicate Supplies were represented by "supplied-by":

```
Supplies(m,p) ≡ p in supplied-by(m)
```

then the possibilities are shown in figure 4.3. This differs from figure 4.2 only in that a different vertical edge is solid. The only program then is:

```
for m in british-list do
   for p in supplied-by(m) do
      if out-of-stock(p) then C fi
   od
od.
```

Now suppose that "Supplies" is given a redundant representation, storing both "supplies" and "supplied-by". The possibilities are given by overlaying the two figures and either of the two programs are possible.

The least controversial approach we could take to the problem of selecting a program is to generate and print them

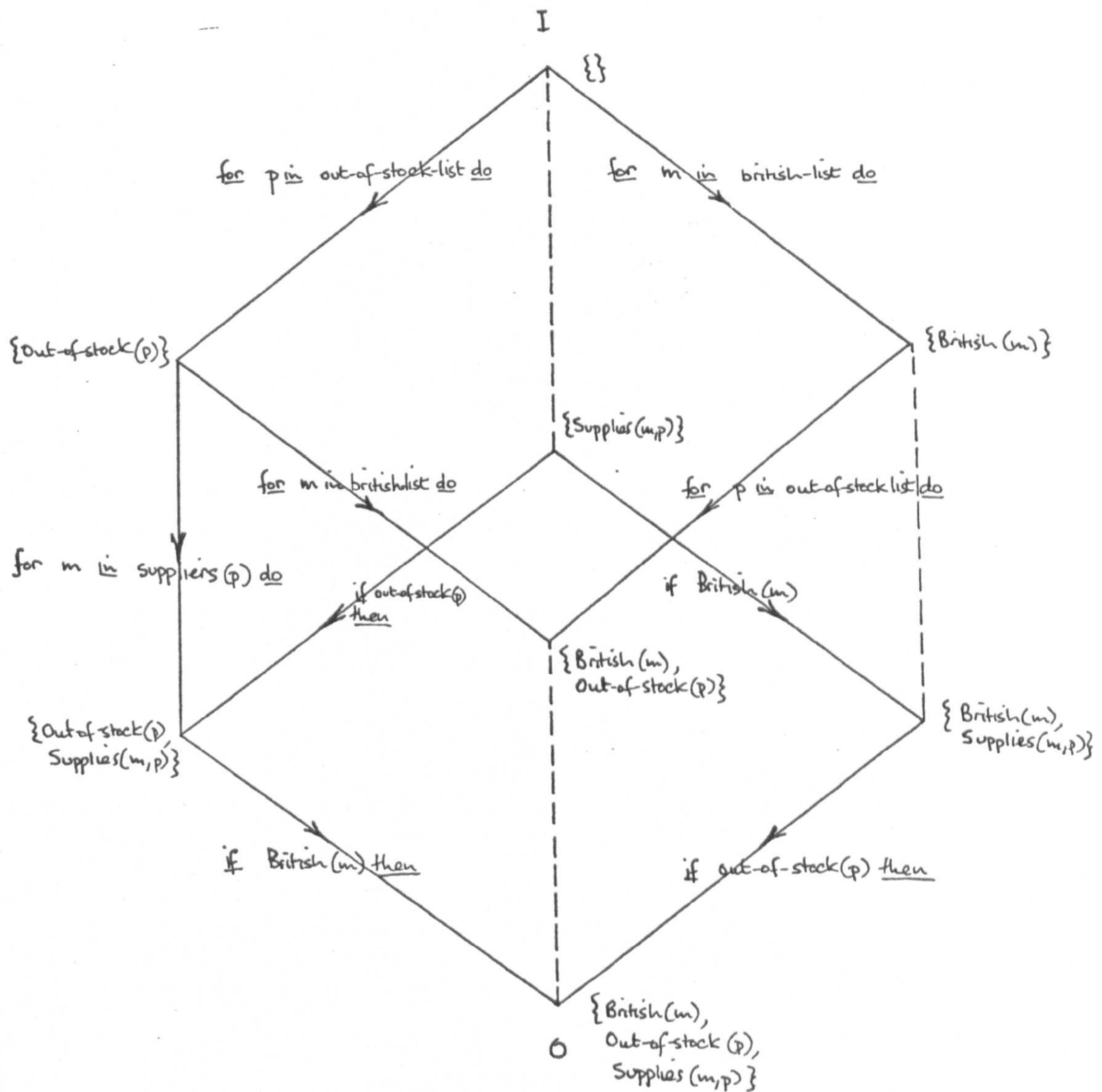


Figure 4.2 Possible programs for British(m) and Supplies(m,p) and Out-of-stock(p).

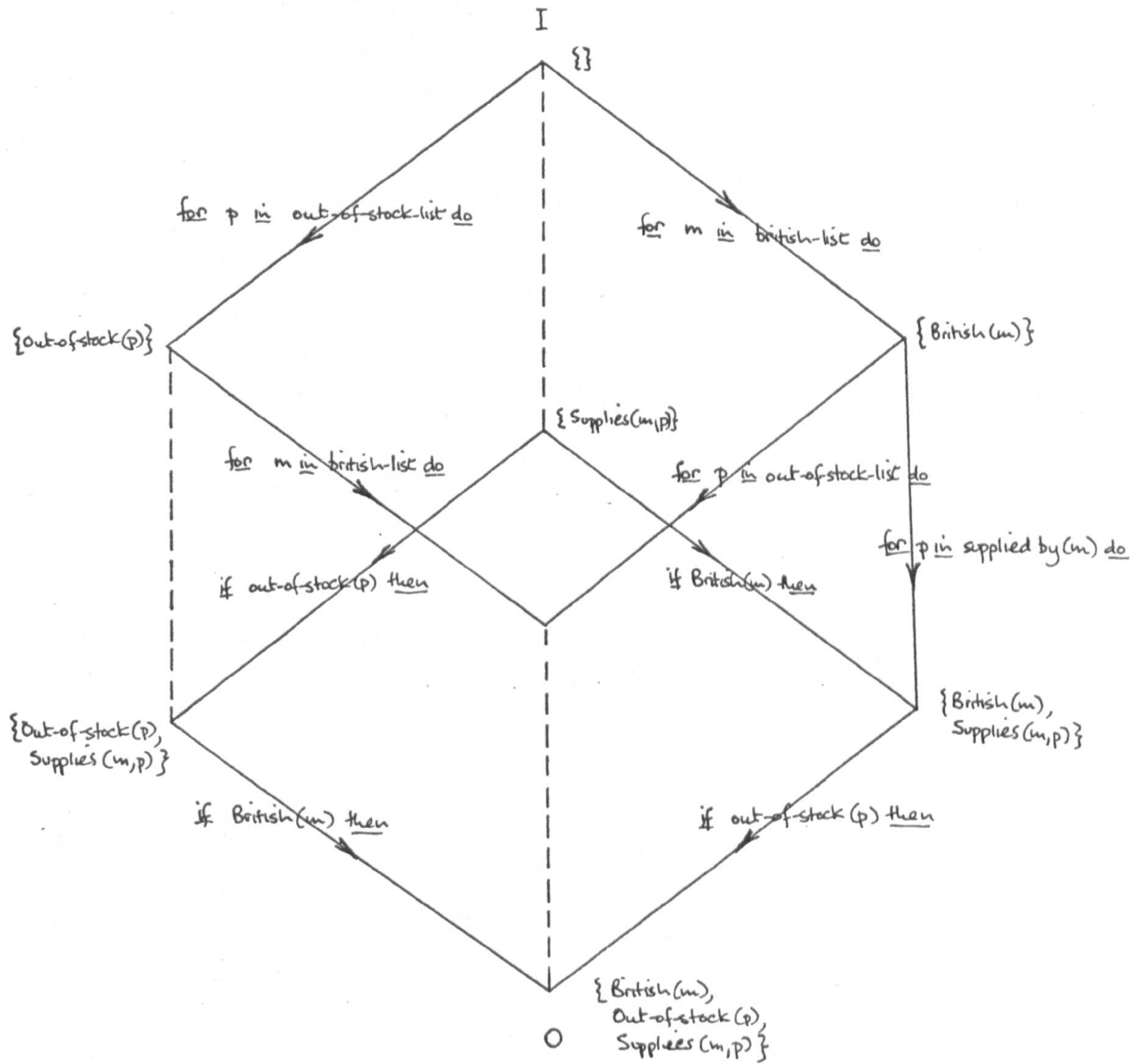


Figure 4.3 Further possibilities for British(m) and Supplies(m,p) and Out-of-stock(p).

all. However this makes the implementation rather slow and awkward to code. It was also found that in a number of cases where only one solution was expected, a large number would be produced. These differed only in minor matters such as the ordering of independent let blocks.

A second alternative is to try to select the program which incurs the least execution cost. Some selection on a cost basis would probably be needed in an automated data retrieval system. As the actual costs of individual operations are not known, a possible approximation to this is to minimise the number of operations performed.

For the arithmetic example in figure 4.1 we might argue as follows:

First, consider programs which contain the line connecting I to 3 and so start:

    for (x,y) in addto(5) do

The function "addto" is expected to produce all pairs of numbers whose sum is given by the parameter. For positive integers this might be possible, but if negative integers were included there would be an infinity of such pairs. Although it is possible for execution of the loop to be terminated prematurely (for example by the existential quantifier), in general a program starting in this way would

not stop. Point 3 has been annotated with the number of iterations (infinity) to show this.

Ignoring these programs, next we consider those starting:

```
for x in 1..20 do
```

The loop body will be executed twenty times and again this number of iterations is shown. The body itself will comprise either line 2,6,8:

```
for y in 1..10 do
   if x + y = 5 then C fi
od
```

or line 2,5,8:

```
let y = 5 - x ;
   if y in 1..10 then C fi
```

both of which execute C with the same set.

The second alternative is to be preferred, as the inner test is executed only once per iteration, while in the first the test is executed ten times.

Similarly, the best program starting with line 1,4 is:

```
for y in 1..10 do
   let x = 5 - y ;
      if x in 1..20 then C fi
od.
```

This is better than the last as the loop bodies are very similar, but here it is executed ten, rather than twenty times. This program should therefore be selected.

To guide this choice it is clearly necessary to know the size of the underlying data and to be able to estimate the size of the various intermediate sets. This motivated a study of the constraints on the sizes of sets formed by expressions in the underlying algebra. Although (with difficulty) some bounds can be obtained no useful simplifying principles were discovered and the rules are probably of only marginal value in practice. For example while the maximum size of the intersection of two sets A and B is the size of the smallest of A and B, the number of members to expect in the result is highly dependent on semantic factors.

The implementation takes a third and very simple approach, producing the first complete program it finds. It was felt that, in the long run, it would be better to rely only on syntactic information in selecting a program rather than try to make deductions from actual data sizes. Working toward this we need to reduce the choice as much as possible and

then to arrange the search so that programs with the least execution cost are encountered first.

Toward the first goal, consider the two alternatives depicted in figures 4.2 and 4.3. While they produce identical sets, if we were to execute them their results would look very different. Each produces a list of manufacturer-part pairs, but in the list produced by the first program, entries containing the same part occur consecutively, while the second will list all entries for one manufacturer together. Practically we would probably not want to give the system the freedom to choose the way that the output is ordered as it is very unlikely that a list sorted by parts would serve the same purpose as a list sorted by manufacturers. The need to make the choice can be avoided by requiring that the output be specified as an appropriate array of sets. The present implementation does not handle arrays sufficiently generally to allow this, but uses the list of output variables as an indication of the result order. If the list were (m,p), then the code would vary the value of p most rapidly. Although we have tried to ignore the question of set ordering, it is apparent that for proper output presentation and to include symmetric merge operations, the notion of an ordered representation of a set should be included.

Fixing the output order avoids many of the choices which

must otherwise be made. The only change this makes in the transformations is to replace the set of variables x in <u>for</u> x|P <u>do</u> C <u>od</u> by a vector of variables, substituting for the union operator on such sets, vector concatenation.

There will, even with this change, be some occasions when a choice of code exists. Suppose we need to find the set of parts p satisfying:

```
Out-of-stock(p) and Obsolete(p)
```

where Out-of-stock and Obsolete are both stored as "powersets":

```
Out-of-stock(p)  ≡ p in out-of-stock-list  ≡ out-of-stock(p)
Obsolete (p)     ≡ p in obsolete-list      ≡ obsolete(p).
```

There is a choice between:

```
for p in out-of-stock-list do
   if obsolete(p) then C fi
od
```

and:

```
for p in obsolete-list do
   if out-of-stock (p) then C fi
od.
```

The shorter list should be processed sequentially, but without a knowledge of set sizes the implementation arbitrarily prefers the first program, sequencing through

the left-hand argument of the <u>and</u> operation. If a collate operation were available it would be better to treat this symmetric situation by symmetric code. A binary merge technique (Knuth 1973) for example achieves good performance on a wide range of sets. For genuinely asymmetric cases an alternative syntax could be introduced.

The tables in appendix A show the order in which the search for a code sequence is made. These illustrate, for instance, that no distribution of <u>and</u> over <u>or</u> takes place unless a code sequence cannot be produced without it.

Chapter 6 gives a series of examples which show the complete compilation process in action and the way that this simplified method of choice works in practice.

CHAPTER 5

SELECTIVE UPDATE

A long term aim of this work has been to further the understanding of the choice of data organisation. If we restrict consideration to data retrieval only, then increasing the amount of redundancy in the stored data will always tend to improve the overall processing time. If the number of alternative structures is increased then it becomes more likely that one will exist to suit a randomly chosen retrieval. The choice of a data organisation reduces to an analysis of the trade-off between retrieval time and storage space. However when the data is subject to change, each update must be propagated through all the relevant data structures. Update times will therefore tend to increase as more redundancy is employed and the optimimum data organisation will depend on the pattern of retrievals and updates. Florentin (Florentin 1972) suggests that update is usually the critical factor in determining this organisation.

These thoughts suggested that an assignment statement should be added to the earlier mini-language and the effect of assignments on the stored data structures investigated. Section 5.1 introduces the method which is used and section 5.2 describes the assignments considered. Subsequent sections discuss the effect of the assignments on data

structures whose contents are defined by the allowable expressions. Ultimately this should allow a comparison to be made between the situations when an expression is computed on each retrieval and when the expression is stored. In the latter case it must be recomputed on each update. Section 5.3 covers the relatively simple case of structures defined only by logical expressions, corresponding approximately to relational expressions, but without the project operator. Section 5.4 then illustrates the extensions necessary to handle more general functions.

5.1 Introduction.

It is assumed that the user views the information in the data-base as being contained in a number of variables. If a change occurs in the stock position of, say, 4BA NUTs this may be reflected by an assignment such as:

Number-on-hand("4BA NUT"):=250.

The assignment changes the function stored in the variable Number-on-hand.

When dealing with update it becomes necessary to distinguish the variable in which information is held from the function or set which happens to be its current value. While we need to make this distinction, that is in this section, section 5.2 and section 5.3, we will distinguish variables by an initial capital letter. Their values will start with a small

letter, regardless of whether they are predicates, functions or sets. Thus Number-on-hand stands for a variable, while "number-on-hand" stands for the function which is its current value. Outside these sections, initial capitals do not have this significance.

Ideally, in making an assignment, there should be no danger that the data held in the data-base will become internally inconsistent. That is, in the example there should be no information held which could imply that there are other than 250 4BA NUTs in stock. This mutual independence of the data is one of the aims of Codd's "third-normal-form". We will therefore assume that variables are separated into two groups, those which are primitive, and those whose values are given by definitions. The primitive variables, which correspond to a third-normal-form view of the data, can be updated freely, but no assignments are allowed to variables which have a definition. The definitions give their values at all times.

For example, an assignment can be made to Number-on-hand, as it is assumed to be in the primitive group, but an assignment is not explicitly allowed to Out-of-stock-list. Its value is given at all times by the definition:

Out-of-stock-list= {p | Number-on-hand(p)=0} .

The effect is as if the value of the variable is computed

each time it is referenced.

Thus although for retrieval, the data may appear to contain many variables whose values are interdependent and these may be used without distinction, for update only a certain group of primitive variables can be changed and all the others take on consistent values.

In spite of the notional separation into variables which store primitive data and those with computed values, there is no need for the actual storage representation to correspond to this. Variables whose values are given by a definition may be stored, provided the value kept in storage is always the same as that given by the definition, and variables which are apparently primitive can be computed, if sufficient data exists elsewhere. We will not be concerned with this last possiblity, but will consider the effect of keeping stored data corresponding to a definition.

Keeping such redundant data may improve the overall system performance. For example, suppose that the parts which are out-of-stock are frequently the subject of enquiries. It may often be necessary during retrievals to obtain the set Out-of-stock-list. If this is permanently in storage there will be no need to search the complete list of parts to obtain its members. This may result in a considerable saving of compute time (at the expense of the extra

storage).

The PRTV system (Todd 1975), for example, will store data corresponding to definitions. When a defined relation is referenced and the data it contains has been computed, the relation is saved in permanent storage. Until the space it occupies is needed for other purposes, subsequent references to the definition can make us of the pre-computed data. The system attempts to maintain in storage data corresponding to the most frequently used definitions.

The PRTV system cannot update this redundant data when an assignment occurs. Any pre-computed relations which may be effected by the assignment must be discarded. The question which concerns us here is how the redundant data should be changed so that it is still correct after the assignment.

With the assignment:

Number-on-hand("4BA NUT"):=250 ,

suppose Out-of-stock-list is stored. Its value must be changed so that it is still given by the definition. It will then not be noticeable that the value is being stored and not computed on reference. One way to guarantee the validity of the redundant data would be to recompute it completely. Whenever the system receives an assignment to Number-on-hand, it could discard the old value of

Out-of-stock-list and re-calculate it by an assignment such as:

Out-of-stock-list := $\{p|\text{Number-on-hand}(p)=0\}$.

The processing would be much the same as would be needed to respond to a request to print the list of out-of-stock parts. A scan would be made of all parts, and those with a number-on-hand of zero collected. Instead of sending them to the printer, these would be stored as the value of Out-of-stock-list. This is obviously grossly inefficient, since the new list will differ from the old one by at most one member. Intuitively, for the assignment given, the correct action is just to delete the member "4BA NUT" from Out-of-stock-list, if it is there. The characteristic of this better method is that it makes use of the earlier value in storage. Rather than re-computing the value completely, it merely makes a small change to the earlier one.

<u>strength</u> <u>reduction</u>.

The transformation to get such an assignment uses the same principle as a standard optimisation carried out by some compilers (Allan 1970,Cocke and Schwartz 1971,Gries 1970). In this context it is used to "strength-reduce" operations occurring within loops. For instance the code:

```
for I:=1 until 10 do
  X:=4*I; . . .
od
```

in the right circumstances is changed to:

```
X:=0;
for I:=1 until 10 do
  X:=X+4; . . .
od.
```

In the unoptimised loop, X is always given the value 4*I at the start of the loop body. It is assumed that there are no other assignments to X, so that during execution of the loop X always has the value 4*I. In calculating each new value of X the old value is ignored.

In the optimised code, the multiplication in the assignment has been "strength reduced" to an addition which utilises the value of X from the previous iteration. As addition is a simpler operation than multiplication, the optimised loop should run more quickly than the original.

The argument which leads to this optimisation runs as follows:

Suppose, on some iteration, I has the value i and X has the value x, so that x=4*i. On the next iteration, I will have some value i' and X some value x'. Again these values will

be related by:

```
    x'=4*i'.
      =4*(i+1)
```

since I is incremented by unity between one iteration and the next. This has expressed the new value of X in terms of the old value of I.

```
      =4*i+4
      = x +4
```

distributing the multiply and replacing 4*i by x. This has obtained the new value of X in terms of the old value. To generate this new value the assignment:

```
    X:=X+4
```

is used.

More generally, suppose we wish to maintain an equivalence:

```
    Y=f(X)
```

between two variables X and Y.

Suppose X has the value x and Y has the value y, so that y=f(x), and an assignment:

```
    X:=u(X)
```

occurs. We need to set the new values of X and Y, say x' and

y', so that:

$$y'=f(x').$$

Because of the assignment, x' will be related to x by:

$$x'=u(x).$$

Substituting we get:

$$y'=f(u(x))$$

relating y' to the old value of X. Now if we have an equivalence of the form:

$$f(u(x))=v(f(x))$$

the new value of Y can be expressed:

$$y'=v(f(x))$$
$$=v(y)$$

so that the assignment:

```
Y:=v(Y)
```

will give Y a new value with the property that Y=f(X).

The problem is to find a suitable set of equivalences with the form:

$$f(u(x))=v(f(x))$$

so that the data-base definitions can be treated in this

way. Osman in his PhD thesis (Osman 1974), mentions the possibility of performing the optimisation for set operations in PRTV.

5.2 Assignments.

The assignments which will be considered have the form:

function [arguments] := valued-expression |

predicate [arguments] := logical-expression.

If the function or predicate is stored in an array, then the assignment corresponds to the update of an array element. For example:

Number-on-hand("4BA NUT") := 250

alters the value of the array element indexed by "4BA NUT" to 250.

The effect of the assignment may be described by defining:

update(a,t,r)=if t then r else a.

update will be used as a replacement for if then else when describing the effect of an assignment. The substitution clarifies the meaning of some of the expressions.

Following an assignment:

A(i):=v

the new value of A, say a', will be related to the old value "a" by:

a'(x)=<u>if</u> x=i <u>then</u> v <u>else</u> a(x)
=<u>update</u>(a(x),x=i,v).

For the example assignment given earlier, the new value of Number-on-hand will be given by:

number-on-hand'(p)=<u>update</u>(number-on-hand(p),p="4BA NUT",250)
=<u>if</u> p="4BA NUT" <u>then</u> 250
<u>else</u> number-on-hand(p).

The functions number-on-hand' and number-on-hand are the same everywhere except for an argument "4BA NUT", where the new function gives 250 regardless of the value returned by the old function.

To create an assignment to a stored item of data, the first step is to obtain its new value in terms of the old one. The next two sections deal in detail with the way this is done. Usually, the relationship takes the form:

r'(x)=<u>update</u>(r(x),t(x),v(x))
=<u>if</u> t(x) <u>then</u> v(x) <u>else</u> r(x)

where r' is the new value of the stored variable R and r is its old value. Then it necessary to generate an assignment to alter the stored value of R appropriately.

In the expression, the new value of R differs from the old value only at values of x which satisfy the test t(x). For these the element R(x) must be set to contain v(x). Other elements of R are to remain unchanged. The code used to make the assignment is:

<u>for</u> x|t(x) <u>do</u> R(x):=v(x) <u>od</u>.

This generates all values of x which satisfy t(x). These value index the elements of R which must be updated. Within the loop the new value v(x) is assigned to them.

For the example:

number-on-hand'(p)=<u>update</u>(number-on-hand(p),p="4BA NUT",250)

we get:

```
for p|p="4BA NUT" do
  Number-on-hand(p):=250
od.
```

Simplifying this according to the processing in chapter 5 gives firstly:

```
let p="4BA NUT";
  Number-on-hand(p):=250
```

then eliminating p gives:

Number-on-hand("4BA NUT"):=250.

5.3 Logical expressions

This section describes how the new value of a variable defined using a logical expression can be obtained in the required form. Of necessity, the examples are of rather an uninteresting nature. A more realistic example is given in the next section.

Suppose that a sparse array (or set of logical fields) is maintained in storage and its value is given at all times by the definition:

Unavailable(p) ≡ Out-of-stock(p) or Obsolete(p).

If an assignment occurs which changes the value of "Out-of-stock" then the value of "Unavailable" may also need to be changed so that the equivalence remains true after the update. If delivery were received for some part "a", then an assignment caused by the change might be:

Out-of-stock(a):=false.

In terms of the function update, the new value of the variable Out-of-stock will be related to the old one by:

out-of-stock'(p)

≡ update(out-of-stock(p), p=a, false).

We would like to construct an assignment to "Unavailable" which uses the old value in the same way. In other words,

we would like to express the new value of "Unavailable" in the form:

unavailable'(p)

$\equiv$ update(unavailable(p),t,r)

where t and r are expressions giving the positions at which the update is to take place and the result to be assigned. Now the new value "unvailable'" must be related to the new value "out-of-stock'" by the defining equivalence. So, after the assignment we must have:

unavailable'(p)

$\equiv$ out-of-stock'(p) or obsolete(p)

$\equiv$ update(out-of-stock(p), p=a, false) or obsolete(p).

Now, appendix C shows that:

update(P,t,r) or Q $\equiv$ update (P or Q, t and not Q, r).

Substituting P$\equiv$out-of-stock(p), Q$\equiv$obsolete(p) we find that the new value of "unavailable" is:

update (out-of-stock(p) or obsolete(p),

p=a and not obsolete(p),false)

$\equiv$ update (unavailable(p), p=a and not obsolete(p), false).

The last expression is in the required form, giving the new value of Unavailable in terms of the old one. The code to update the variable must assign the value false to any

element Unavailable(p) where p satisfies the expression: p=a _and_ _not_ obsolete(p). The code:

```
for p| p=a and not Obsolete(p) do
    Unavailable(p):=false
od
```

is needed. This simplifies to:

```
let p=a;
   if Obsolete(p) then else Unavailable(p):=false,
```

so that the additional assignment:

```
if Obsolete(a) then else Unavailable(a):=false
```

maintains the consistency of the stored data. Intuitively this assignment is correct. If Obsolete(a) is _true_ then the value of Unavailable(a) is unaffected by the change to Out-of-stock(a), but if Obsolete(a) is _false_ then Unavailable(a) must be given the same value as Out-of-stock(a).

To deal with a complex expression, the equivalences may need to be applied iteratively. For example, suppose:

```
Supplies-good-part(m,p) ≡ Supplies(m,p) and not Obsolete(p)
```

The assignment: Obsolete(a):=_true_ produces a new value of Obsolete(p) given by:

```
update(obsolete(p), p=a, true).
```

The new value of Supplies-good-part(m,p) is:

```
Supplies(m,p) and not update(obsolete(p),p=a,true)
```

First the innermost expression is treated, using the equivalence from appendix C:

```
not update(P,T,L) ≡ update(not P,T,not L).
```

We get:

```
Supplies(m,p) and update(not obsolete(p), p=a, false).
```

Next update is carried through the and using:

```
P and update(Q,t,r) ≡ update(P and Q,t and Q,r).
```

The result is:

```
update(Supplies-good-part(m,p), p=a and Supplies(m,p),false).
```

Assuming that Supplies is defined in terms of the representation:

```
Supplies(m,p) ≡ m in suppliers(p)
```

the complete code needed for the assignment is:

```
for m in suppliers(a) do
   Supplies-good-part(m,a):=false
od;
Obsolete(a):=true.
```

The equivalences for <u>and</u>, <u>or</u> and <u>not</u> allow the strength reduced assignments to be formed for expressions constructed using the logical connectives. However as will be seen in the next section this does not extend to the majority of relational expressions because expressions containing a project operator (or its equivalent <u>some</u>) cannot always be optimised. It is necessary to recompute at least part of a general projection on each update. We should also note that even with just the simple logical connectives, not all expressions can be optimised. To obtain an assignment for example to:

R ≡ P <u>and</u> Q

when P changes, we have implicitly assumed that Q remains constant. This need not be so, as in:

R(x,y,z) ≡ Father(x,y) <u>and</u> Father(y,z)

if the predicate "Father" should change. The transformations do not always work in such a situation.

5.4 Other functions

An outline of the rules for strength-reducing assignments for most of language F are contained in Appendix A. Rather than writing the rules as equivalences, they have been expressed directly in statement form. For example, to maintain the equivalence: R ≡ P <u>and</u> Q when an assignment:

for x|T *do* P:=V *od*

occurs, it is necessary to make the additional assignment:

for (x∪r)|T *and* Q *do* R:=V *od*

(where r is the set of free variables in R). The last statement results from applying the corresponding equivalence.

A significant omission from the appendix is a rule for *the*. *the* is a partial function and its result is undefined unless the argument is a unit set. To cater for this we need to allow stored functions to give undefined results for some of their arguments. In a data-base situation this simply corresponds to fields with temporarily undefined values. It appears quite possible to carry this through, but the implications are very wide. For example the two sets:

{p|number-on-hand(p)=0}

{p|*not* number-on-hand(p)≠0}

need not be identical if number-on-hand may produce undefined values. The first set contains parts which have a number on hand of zero. The second contains all parts except those with a non-zero number on hand, and so contains not only parts with a zero number on hand, but also any whose number-on-hand is undefined.

If the right results are to be obtained, the system cannot

use the equivalence: P ≡ _not_ _not_ P, distinguishing between "known to be true" and "known not to be false". Although for retrieval the implementation appears not to violate this intuitionist logic, the update rules are made slightly more complex and have not been included.

The appendix contains a new type of assignment which is only generated internally. It is clear for example that "becomes equal to" is not always appropriate for updating sets, as we may wish to modify the set contents rather than change its value completely. To make such small changes the two statements:

S:_plus_ x   and   S:_less_ x

are used. The first adds x as a new member of S. The result is not defined if the set already contains x. In practice an implementation should check whether an attempt is being made to duplicate an existing member and issue an error message if this is so. Analagously, S:_less_ x deletes an existing member x from S and an implementation is expected to complain if S does not contain x. These assignments were chosen in preference to normal union and difference so that the set is always changed.

An example.

To illustrate some of the resulting code sequences we will

take a simplified airline reservation system as an example. This is to contain data about a number of "bookings". A booking associates a "passenger" with a "flight". If a passenger p is booked onto a flight f these will be a (single) booking, b say, so that:

passngr-booked(b)=p
<u>and</u> flight-booked(b)=f.

There may also be other information about the booking, for example the date when the booking was made. Flights are identified externally by their flight-number and departure date. It is also assumed that data is kept giving the aircraft which is scheduled to operate each flight. This data is represented by a function "assigned-aircraft". Auxiliary information is also held about each aircraft in use, its type, seating capacity and so on. The situation is represented diagramatically in figure 5.1, which shows by labelled arcs the various functions connecting the sets Bookings, Passengers, Flights and Aircraft.

When a passenger is to be booked onto a flight, a check should first be made that the flight is not already full. The set of bookings already made for each flight is given in terms of the primitive function "flight-booked" by:

bookings(f) = {b|flight-booked(b)=f}.

The number of seats which could be booked on each flight is

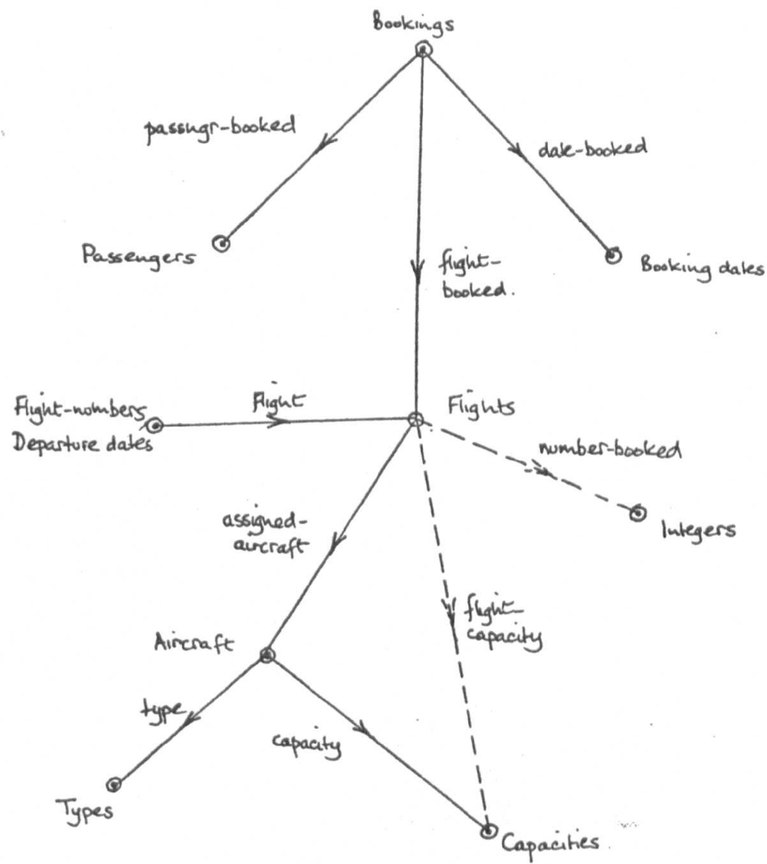


Figure 5.1 Simple reservation system.

given by:

flight-capacity(f) = capacity(assigned-aircraft(f)).

As no aircraft should be overbooked, we should at all times have:

flight-capacity(f)>= number bookings(f).

When making a booking, we should check that this constraint will not be violated. To do this, we could evaluate the expression for the particular flight. However it would not be sensible to count the number of passengers already booked on at each request for a seat. Clearly it would be better to keep permanently the number of passengers booked onto each flight. The set of fields needed has a value given at all times by:

number-booked(f)=number bookings(f).

Similarly, to avoid referring to the Aircraft file, an extra field could be added to flight records to contain the value of "flight-capacity". Again the value is always to be given by the definition. (The redundant functions are shown in figure 5.1 as dashed arcs.)

When making a booking it will only be necessary to compare these two fields in the flight record to make the check. To compensate for the easier data retrieval, the redundant fields must be recomputed at each update and some

assignments which effect them will now be considered.

The first update is the change to the aircraft operating a flight. Suppose the assignment:

```
assigned-aircraft(flight("BE300","TUES")):="VISCOUNT1"
```

is made. In the absence of the redundant data, the code sequence:

```
let f=flight("BE300","TUES");
    assigned-aircraft(f):="VISCOUNT1"
```

merely setting the appropriate field, would be adequate. However, because the value of "flight-capacity" is stored and its value depends on that of "assigned-aircraft", this must be updated as well.

The definition of "flight-capacity":

```
flight-capacity(f) = capacity(assigned-aircraft(f))
```

has the form: R=F(E). Appendix A contains the rule that to maintain this equivalence following the assignment:

<u>for</u> x|T <u>do</u> E:=V <u>od</u>

we need the additional assignment:

<u>for</u> $(x \cup r)$|T <u>do</u> R:=F(V) <u>od</u>.

Making the substitutions:

```
R=flight-capacity(f)
F=capacity
E=assigned-aircraft(f)
V="VISCOUNT1"
x=r={f}
T= (f=flight("BE300","TUES") )
```

we find that the assignment needed is:

```
for f|f=flight("BE300","TUES") do
  flight-capacity(f):=capacity("VISCOUNT1")
od
```

or:

```
let f=flight("BE300","TUES");
  flight-capacity(f):=capacity("VISCOUNT1").
```

This has the effect of setting the capacity of the flight to that of the new aircraft. Notice that since we have altered the value of "flight-capacity", we may also have effected the truth of:

```
flight-capacity(f)>= number booked(f).
```

We should check that the flight is not now overbooked. The checking of a constraint such as this needs exactly the same processing as would be used to maintain its value in storage. However instead of saving the resulting value we would merely verify that the value computed is true.

Another logically possible, but unlikely change is that an

aircraft may have some seats removed so changing its capacity. Suppose that:

```
capacity("CONCORD1"):=90.
```

Again we must update "flight-capacity". Appendix A contains the rule that to maintain an equivalence R=F(E) given an update:

```
for x|T do F(E'):=V od
```

needs the additional assignment:

```
for (x∪r)|T and E=E' do R:=V od.
```

Making the substitutions:

```
R=flight-capacity(f)
F=capacity
E=assigned-aircraft(f)
E'="CONCORD1"
V=90
x=r={f}
T=true
```

we obtain:

```
for f|assigned-aircraft(f)="CONCORD1" do
  flight-capacity(f):=90
od.
```

This has the effect of changing the flight capacity of all flights using the aircraft being altered.

To perform the assignment, if the representation contains only the functions shown in figure 5.1, there will be no alternative but to search all the possible flights:

```
for f in Flights do
 if assigned-aircraft(f)="CONCORD1" then
      flight-capacity(f):=90
 fi
od;
```

However, if chains were kept through all flights using the same aircraft we could find those of interest directly and so improve the code.

The last change we will consider is the alteration of a booking from one flight to another. If the booking to be altered is B, then the assignment might be:

```
let b=B;
 let F=flight("BE300", "TUES");
   flight-booked(B):=F.
```

The stored function "number-booked" depends on "flight-booked" so will need to be changed. To find the changes, it is convenient to divide its definition into a number or parts which can be treated separately:

```
number-booked=number bookings(f)
bookings(f) = {b| Flight(b,f)}
Flight(b,f) ≡ (flight-booked(b)=f).
```

The change to the predicate Flight can be found by again applying rule 9 from the appendix, exactly as for the first assignment in this section. In this case we have:

```
R    = Flight(b,f)
F(x)= (x=f)
E    = (flight-booked(b)=f)
V    = flight("BE300","TUES")
T    = (b=B)
```

leading to the assignment:

```
for {b,f}| b=B do
  Flight(b,f):=(flight("BE300","TUES"))
od.
```

Continuing with the next part of the definition:

```
bookings(f)={b|Flight(b,f)} .
```

The appendix suggests the rule that, to maintain:

```
R={y|P}
```

when the assignment:

```
for x|T do P:=V od
```

occurs, we need the pair of loops:

```
for (x∪r)|T and y in R do R:less y od;
for (x∪r)|T and V      do R:plus y od.
```

To keep "bookings" up to date we need:

```
for {b,f}|b=B and b in bookings(f) do
  bookings(f):less b
od;
for {b,f}|b=B and f=flight("BE300","TUES") do
  bookings(f):plus b
od
```

or:

```
let f=flightbooked(B);
  bookings(f):less B;
let f=flight("BE300","TUES");
  bookings(f):plus B.
```

This has the effect of subtracting B from the bookings of the original flight with which it was associated, and adding it to the new flight.

The last step is relatively straightforward. Given that:

```
number-booked(f)=number bookings(f),
```

when a member is removed from bookings(f), one must be subtracted from "number-booked(f)". This is true because we know that the member was indeed there beforehand. Similarly, when a member is added to the set, one must be added to "number-booked". The final assignment results from applying this rule from the appendix:

```
let f=flight-booked(Γ);
```

```
    number-booked(f):=number-booked(f)-1;
  let f=flight("BE300","TUES");
    number-booked(f):=number-booked(f)+1.
```

The effect is to subtract one from the number-booked on the original flight, and to add one to that of the new flight.

As a final remark, there is not always a strength reduced method of updating a structure defined using the function <u>some</u>. As an illustration, we might consider keeping a logical flag to indicate that at least one passenger is booked onto a flight:

```
  not-empty(f) = some bookings(f).
```

If a new member is added to the bookings of some flight F, then it will clearly become not-empty. However if a booking is cancelled for flight F there is no alternative but to re-evaluate the expression "<u>some</u> bookings(F)" in the new state. It was this problem which finally caused the author to reject of Codd's relational algebra as a means of constructing the data of interest. The project operator has the same characteristics as <u>some</u>, but must be used in almost every interesting relational expression. In consequence very few redundant structures so defined can be efficiently updated.

CHAPTER 6

AN EXPERIMENTAL SYSTEM

A number of systems have been written to test ideas as they developed. Their purpose has been to check the algorithms, uncover any practical difficulties and to stimulate further thought. No attempt has been made to produce a complete and workable system and the facilities provided have always been kept to the minimum necessary for experiment.

The examples later in the chapter show the output from the most recent of these systems. It compiles a definition of a set into abstract Algol-like code and can also handle simple updates. Like earlier versions, it is coded in Algol-W (Algol-W 1972) and at the time of writing consists of about a thousand lines of code. It was developed on the IBM Model 168 at Newcastle University running under the Michigan Terminal System (MTS).

6.1 Outline description.

The main components of the system and the overall data-flow are shown in figure 6.1. The processing is divided into three main stages, executed one after the other. This arrangement results largely from the non-deterministic nature of the compilation process. The input stage converts statements to internal data structures which may be scanned repeatedly by the code generator in its search for a

satisfactory code sequence. Only when the choice has been completely made is the rather lengthy job of generating formatted output text begun.

The input stage is divided in the usual way into two sub-parts. A statement is first broken into a string of words. A word may be an identifier, an integer, an operator (any string of other characters) or one of the special words "(" ")" ";" "," and " itself. The syntax analyser then builds expressions from these words. The analyser is table driven and relies mainly on operator precedence to govern its actions. Operators are allowed to bind with different strengths on the two sides. For example the infix operator "|", used to indicate a set construction, binds tightly on the left where a parameter is expected, but loosely on the right where a logical expression may appear. In operation, the analyser runs two stacks, one for operators and one for operands in the conventional way.

The syntax accepted is given in appendix A. It is fairly close to that used in the earlier chapters, the differences coming from the more restricted character set which must be used. The table of operator priorities shown can be modified by the input command PRIORITY.

The analyser builds a list-structure to represent each program statement, each element in the structure containing

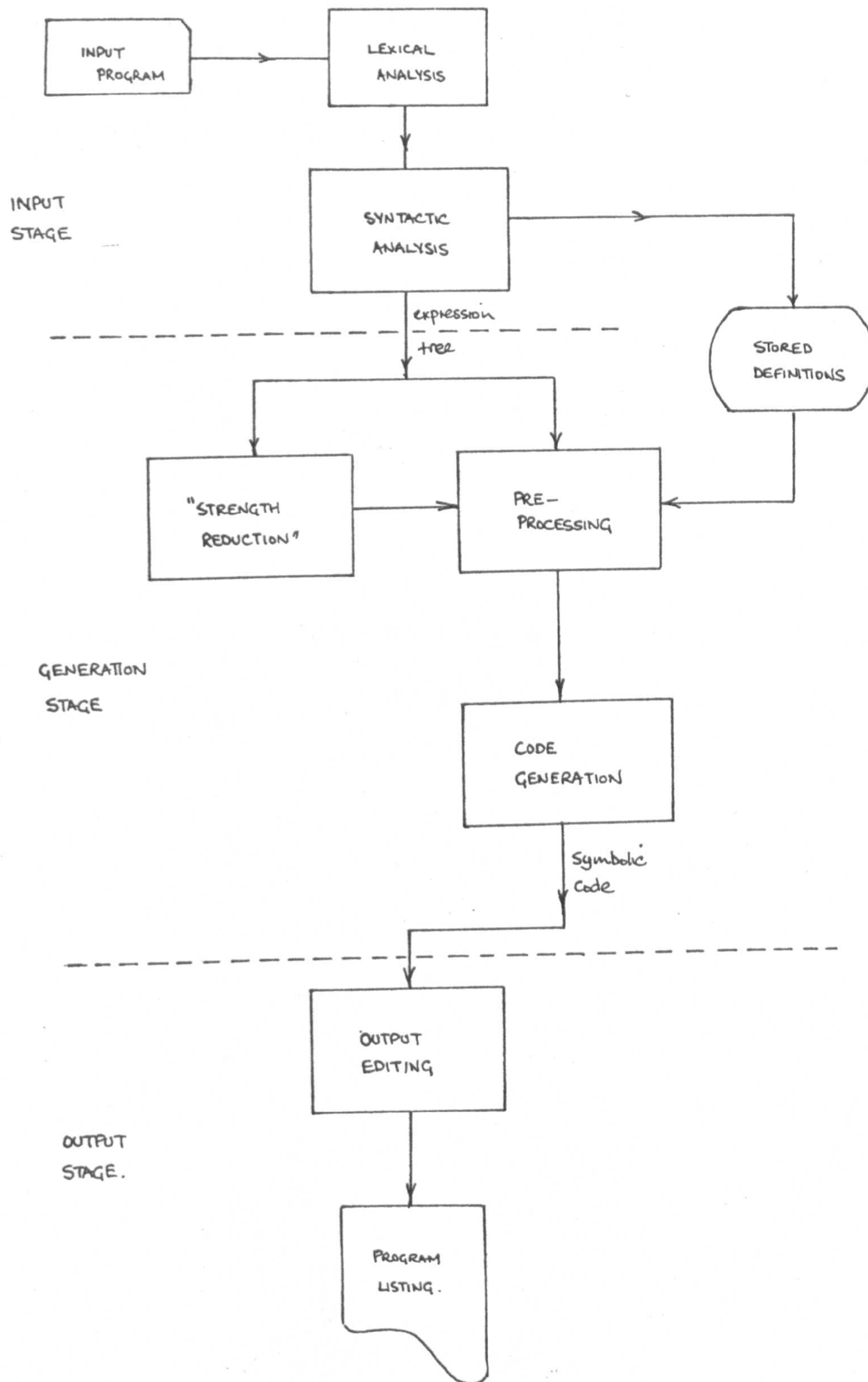


Figure 6.1 Structure of the experimental system

a reference to a function and a reference to its argument. The structure is saved if it is part of a definition or passed to the remaining phases if it is to be compiled.

Most of the logic is contained in the next phase, that of code generation. The three main sub-units are the expression pre-processor, the strength-reducer and the code generator itself. As this part of the system has been subject to continual modification, the transformations have been represented in the code as directly as possible. Conceptually, the pre-processor produces from each input expression a series of alternatives for the code generator to try. In practice it is sensible to run the pre-processor in a lazy manner, so that the generation of an alternative can be stopped as soon as the code generator finds that no code is possible and so that the whole pre-processor can be stopped as soon as one complete code sequence is found. This leads to a fairly unconventional program structure, making very heavy use of procedure parameters, effectively simulating coroutines. Apart from this peculiarity, the coding is a straight-forward transliteration of the rules in Appendix A and table 5.1.

If a code sequence is found, the internal data structure generated is passed to the output routine for printing. The use of <u>if</u> <u>fi</u>, <u>do</u> <u>od</u> statement bracketting rather than the normal <u>begin</u> <u>end</u> convention was motivated by the simpler

printing algorithm obtained. As well as formatting the text, the output routine also tries to clean up the program to make it more readable. LET blocks are eliminated where possible (that is where there is only one use of the variable defined or where the defining expression is just a variable or constant), so reconstructing compound expressions. IF statements with a null THEN clause: IF p THEN ELSE s FI are replaced by the equivalent form: IF ¬p THEN s FI. An attempt is also made to re-construct expressions in infix operator notation.

Overall, the system has been coded rather in the style of Lisp, with few loops or assignments and a great deal of recursion. No attention has been paid to the efficiency of the code. Rather surprisingly, the bulk of the execution time appears to be spent in the output routine, which is fairly slow.

6.2 Examples

This section shows the output from a run of the system, interspersed by comments. The examples illustrate most of the features discussed earlier.

```
  **LANGUAGE F LOADED.
ECHO ON;
```

The initial message indicates the syntax table which is in

use. The system is then asked to echo all input lines it is given. These appear in column 1 while system output is indented to column 3.

1.
```
X|X IN 1..10;
  =>
   FOR X IN 1..10 DO
    WRITE(X)
   OD.
```

The "|" in the input line acts as infix set constructor, no curly brackets being available in the character set. The program produced by the system is preceded by =>. It should print all members of the set defined. In this case only a <u>for</u> loop is needed.

2.
```
X|X IN 1..10 AND X>3;
  =>
   FOR X IN 1..10 DO
    IF X>3 THEN
     WRITE(X)
    FI
   OD.
```

The additional condition results in a program consisting of nested <u>if</u> and <u>for</u> statements.

3.
```
X|X>3 AND X IN 1..10;
  =>
   FOR X IN 1..10 DO
    IF X>3 THEN
     WRITE(X)
    FI
   OD.
```

The order of appearance of the clauses does not, in this case, effect the outcome.

4.

```
X:1..10 | X>3;
  =>
   FOR X IN 1..10 DO
    IF X>3 THEN
     WRITE(X)
    FI
   OD.
```

The infix operator ":" is used to constrain a variable to a set. In this case the result is the same as in the two previous examples. However, constraints introduced in this way are treated specially. Such a set is used as a source of values only if there is no other alternative.

5.

```
X| X IN (1..5 OR 6..10) AND ODD(X);
  =>
   FOR X IN 1..5 DO
    IF ODD(X) THEN
     WRITE(X)
    FI
   OD;
   FOR X IN 6..10 DO
    IF ODD(X) THEN
     WRITE(X)
    FI
   OD.
```

Here, the disjunct is used to generate values for x, the OR acting as a disjoint set union. The print algorithm duplicates the code to be executed rather than creating a subroutine called from within each <u>for</u> loop.

6.

```
X:1..10 | X>5 OR 3>X;
  =>
   FOR X IN 1..10 DO
    IF X>5 THEN
     WRITE(X) ELSE
     IF 3>X THEN
      WRITE(X)
     FI
    FI
   OD.
```

In this example the OR is not used to generate values, but only to test them. An _if_ _then_ _else_ structure is used to do this.

7.

```
X:1..10 | NOT X>5 ANDOR X>3;
  =>
   FOR X IN 1..10 DO
    IF X>5 THEN
     IF X>3 THEN
      WRITE(X)
     FI ELSE
     WRITE(X)
    FI
   OD.
```

The negation (which binds more tightly than ANDOR, the operator standing for a normal inclusive or) is implemented by reversing the arms of the conditional generated.

8.

```
(X,Y)| X IN 1..20 AND Y IN 1..10 AND X+Y=5;
  =>
   FOR X IN 1..20 DO
    FOR Y IN 1..10 DO
     IF X+Y=5 THEN
      WRITE(X,Y)
     FI
    OD
   OD.
```

Here values for a list of variables are to be printed. The code always arranges to vary the value of the last variable in the list most rapidly. The compiler cannot produce the better code sequence which calculates Y from X+Y=5 without a series of alternative definitions for the operation "+". If the definitions are provided, the code produced depends on the order of appearance of the conditions.

The next examples show the effect of various data representations in simple cases.

9.
```
P:PARTS | OUT_STOCK(P) AND OBSOLETE(P);
  =>
   FOR P IN PARTS DO
    IF OUT_STOCK(P) THEN
     IF OBSOLETE(P) THEN
      WRITE(P)
     FI
    FI
   OD.
```

OUT_STOCK and OBSOLETE do not have definitions and so are assumed to be stored as functions of PARTS. The most likely representation would be as logical flags in each part record, when the names OUT_STOCK and OBSOLETE would be interpreted as field selectors. The code tests these fields in each record, printing those which satisfy the condition.

10.
```
LET OUT_STOCK(P)   <= P IN OUT_STOCK_PARTS;

P:PARTS | OUT_STOCK(P) AND OBSOLETE(P);
  =>
   FOR P IN OUT_STOCK_PA DO
    IF OBSOLETE(P) THEN
     WRITE(P)
    FI
   OD.
```

Here the LET is used to define the predicate OUT_STOCK in terms of the members of a set. The set is assumed to be stored as an (ordered) list. The definition could be hidden from a casual data-base user so that he is not aware of the actual representation. Rather than search the complete set of parts, only those in OUT_STOCK_PARTS

are inspected. (The variable OUT_STOCK_PARTS exceeds the twelve-character maximum length and has been truncated in the output program.) If instead OBSOLETE were stored as a list, the situation would be reversed.

11.
```
LET OBSOLETE(P)   <= P IN OBSOLETE_PARTS;

P:PARTS | OUT_STOCK(P) AND OBSOLETE(P);
  =>
   ** CANNOT PRINT -  COLLATE
```

The earlier definition of OUT_STOCK still holds and OBSOLETE is now also represented as a list. The best way to obtain the answer is to collate the two lists, but the print algorithm objects to the choice.

12.
```
LET OUT_STOCK(P)  <= OUT_FIELD(P)
                  <= P IN OUT_STOCK_PARTS;

P:PARTS | OUT_STOCK(P) AND OBSOLETE(P);
  =>
   FOR P IN OBSOLETE_PAR DO
    IF OUT_FIELD(P) THEN
     WRITE(P)
    FI
   OD.
```

This introduces a redundant representation for OUT_STOCK, "<=" preceding each alternative definition. In this case only one code sequence is possible, but if both predicates are stored redundantly there is a choice:

13.

```
LET OBSOLETE(P)    <= OBS_FIELD(P)
                   <= P IN OBSOLETE_PARTS;

P:PARTS | OUT_STOCK(P) AND OBSOLETE(P);
  =>
   FOR P IN OUT_STOCK_PA DO
    IF OBS_FIELD(P) THEN
     WRITE(P)
    FI
   OD.
```

The left hand operand is chosen arbitrarily to generate trial values, although a collate operation could have been used instead.

The next three examples illustrate the effect of keeping a secondary index to a file. In the first, the cost of a part is defined by a field COST_FIELD in each part record only.

14.

```
LET COST(P:PARTS)   <= COST_FIELD(P);

X| 4=COST(X);
  =>
   FOR X IN PARTS DO
    IF 4=COST_FIELD(X) THEN
     WRITE(X)
    FI
   OD.
```

To find the parts whose cost is 4 the only possibility is to search all parts, testing the cost field.

15.

```
LET COST(P:PARTS)   <= COST_FIELD(P)
                    <= THE C:COSTS | P IN PARTS_COSTING(C);

X| 4=COST(X);
  =>
   FOR X IN PARTS_COSTIN(4) DO
    WRITE(X)
   OD.
```

However, here the representation includes both a cost field and also an array PARTS_COSTING. According to the definition this array gives the set of all parts whose cost is C. The definition of COST in terms of PARTS_COSTING amounts to:

C=COST(P) = P IN PARTS_COSTING(C).

The code uses this secondary index to obtain the parts of interest directly.

16.
```
X| COST(X)>4;
  =>
   FOR C IN COSTS DO
    IF C>4 THEN
     FOR X IN PARTS_COSTIN(C) DO
      WRITE(X)
     OD
    FI
   OD.
```

Again the inversion is used, the code first finding all costs exceeding 4 and then the parts having each cost. As each part can have only one cost, the sets of parts produced from each of the costs tried will be disjoint. A part will only be produced once, but the set will not be produced in sorted order. Both to produce acceptable output and for merge operations the sets should be sorted, but the compiler currently fails to check this.

To illustrate the code for SOME, consider finding the manufacturers who can supply any of the out-of-stock parts. The definition of the set is:

$\{ m | \underline{some}\{p | m \ \underline{in} \ suppliers(p) \ \underline{and} \ p \ \underline{in} \ out\text{-}stock \}\}$.

If both "suppliers" and "out-stock" are represented in the way suggested in the expression, "out-stock" as a set and "suppliers" as an array of sets, then the code is:

17.

```
M| SOME( P| M IN SUPPLIERS(P) AND P IN OUT_STOCK_PARTS );
  =>
   LET S1=EMPTY;
    FOR P IN OUT_STOCK_PA DO
     FOR M IN SUPPLIERS(P) DO
      S1 UNION M
     OD
    OD;
    FOR M IN S1 DO
     WRITE(M)
    OD.
```

The code collects all suppliers of out-of-stock parts in the set S1. (variables generated by the compiler have a first letter indicating the type of value - "S" for sets, "A" for arrays and "X" for scalar items - and are numbered sequentially.) The set in this case serves to eliminate duplicate appearances of a manufacturer who supplies more than one part.

An abbreviation has been built in for expressions of this form to make them easier to enter. It comes from Carnap (Carnap 1958, D32-6, p127):

```
R''S = X | SOME (Y | X IN R(Y) AND Y IN S).
```

R''S can be read "the R of S" and is expanded by the syntax analyser to give the right-hand-side.

To illustrate a change in representation, suppose that the set of out-of-stock parts is not stored directly but defined in terms of a logical field in each part record:

18.
```
LET OUT_STOCK_PARTS  <= P:PARTS | OUT_FIELD(P);

 SUPPLIERS''OUT_STOCK_PARTS;
  =>
   LET S3=EMPTY;
    FOR X2 IN PARTS DO
     IF OUT_FIELD(X2) THEN
      FOR X1 IN SUPPLIERS(X2) DO
       S3 UNION X1
      OD
     FI
    OD;
    FOR X1 IN S3 DO
     WRITE(X1)
    OD.
```

The code is similar to the previous example, except that the out-of-stock parts must be found by searching all the parts.

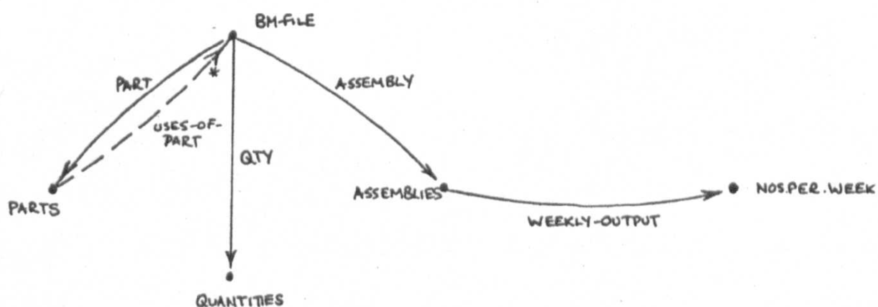
In the next example the definition of OUT_STOCK_PARTS still holds, but this time the suppliers of each part are not stored directly:

19.
```
LET SUPPLIERS(P)  <= M:MANUF | P IN PARTS_FROM(M);

 SUPPLIERS''OUT_STOCK_PARTS;
  =>
   FOR X1 IN MANUF DO
    FOR X2 IN PARTS_FROM(X1) DO
     IF OUT_FIELD(X2) THEN
      WRITE(X1);
      GOTO L1
     FI
    OD
  L1:
   OD.
```

SUPPLIERS is defined in terms of a function PARTS_FROM. The representation is arranged so that each manufacturer record identifies (for example by physically preceding) a sequence of the parts which can be obtained from him. The code which results is somewhat different because it is no longer possible to locate the manufacturers of interest directly. The complete set of manufactures are processed in turn and the parts obtainable from each are tested to determine whether one is out-of-stock. This uses the simpler type of code, a GOTO terminating the search once a part satisfying the condition is found.

In chapter 1 an example was given to find the weekly consumption of the parts used in a number of assemblies. This can be expressed using the function sum. We imagine a set of objects, each representing the use of a part in an assembly. With each of these uses a "quantity-used" (shortened to QTY) is associated, giving the number of the particular part used in that assembly. In addition, each assembly has a "weekly-output" associated with it. The situation is shown diagramatically as follows:



These functions could be represented by two files, one for bill-of-materials data with fields PART,ASSEMBLY and QTY, and one for assemblies containing a field WEEKLY_OUTPUT.

The weekly consumption of a part for a particular use (that is for one assembly) will be given by:

```
LET CONSUMPTION(U)  <= QTY(U)*WEEKLY_OUTPUT(ASSEMBLY(U));
```

These values must be summed for all uses of each part. The set of uses of a part is given by a function USES_OF_PART, the inverse of the function PART. This could be stored directly, for example by following part records by the relevant bill-of-material records, or by chaining together the bill-of-material records which refer to the same part. Given such a representation the code would be:

```
20.
  (P:PARTS,N) | N=SUM(USES_OF_PART(P),CONSUMPTION);
   =>
    FOR P IN PARTS DO
     LET N=0;
      FOR X2 IN USES_OF_PART(P) DO
       N:=N+QTY(X2)*WEEKLY_OUTPU(ASSEMBLY(X2))
      OD;
      WRITE(P,N)
    OD.
```

This processes the parts serially, summing the weekly consumptions from all the relevant bill-of-materials records. However, if the inversion is not stored so that only the functions shown on solid lines in the figure are available, USES_OF_PART must be expressed in terms of the

selector function PART. The code becomes:

```
21.
  LET USES_OF_PART(P)  <= U:BM_FILE | PART(U)=P;

    (P:PARTS,N) | N=SUM(USES_OF_PART(P),CONSUMPTION);
     =>
      LET A1(P)=0;
       FOR U IN BM_FILE DO
        LET P=PART(U);
         A1(P):=A1(P)+QTY(U)*WEEKLY_OUTPU(ASSEMBLY(U))
       OD;
       FOR P IN DOMAIN(A1) DO
        LET N=A1(P);
         WRITE(P,N)
       OD.
```

A sparse array A1 is initialised to zero and the bill-of-materials records are scanned once, accumulating all the sums in parallel. Finally the results stored in the array are printed. A possible inplementation of this abstract code was given in figure 1.2.

The last example of data retrieval is taken from a recent paper by Halstead (Halstead 1975). This compares two programs written to perform the same task. One is written in DSL ALPHA (Codd 1971) assuming relational data, and one is written in COBOL (with Codasyl DBTG extensions) to operate on a particular organisation of the data. The problem is as follows:

> Given a machine X, start date A, stop date B, find the identification number of a person who has adequate skill to operate X and is available between date A and date B to carry out the operation. Schedule this

person if one is located.

We are going to convert a form approximating to the relational program into one which executes like the COBOL. For comparison, the DSL ALPHA and COBOL solutions are given in Appendix B. The DSL ALPHA program assumes the following relations:

(i) PERSON-SKILL(P#,SKILL#)

which relates a person-number (P#) to some skill-numbers (SKILL#), representing the skills the person has.

(ii) MACHINE-SKILL(MACH#,SKILL#)

which associates each machine-number (MACH#) with the skill-numbers which relate to the machine. A person P# can operate a machine M# if there is a skill S# so that the tuple (P#,S#) occurs in PERSON-SKILL and (M#,S#) is in MACHINE-SKILL.

(iii) SCHED(P#,MACH#,SCHED-START-DATE,SCHED-STOP-DATE)

where each member in the relation shows that person P# is scheduled to operate MACH# between the indicated dates.

The DBTG COBOL solution assumes the following representation:

1. A set of person records.
2. A set of machine records.

3. A set of skill-link records.

Person records contain the identification number of a person as a field IDENTIFICATION-NUM. Machine records are indexed by the machine name (such as "X") and each machine record contains the group of schedule entries for the machine. Each record in the group contains the start and stop dates and the person scheduled to operate it between those times. No machine number is included as this information is implied by the group in which the record occurs.

A skill-link record reflects the ability of a person to operate a machine. It corresponds to a SKILL# in the relational version. Each identifies the person and the machine participating in the link. (We will call these fields PERSON and MACHINE, although the COBOL program does not refer to them explicitly.) Each machine record is chained (using a ring of pointers) from a field NEEDS-SKILL to all the skill-link records associated with the machine. This enables all the people who can operate the machine to be located via its set of skill-link records. Each person-record also has a field (HAS-SKILL) giving a similar chain through the skill link records. Figure 6.2 illustrates the representation for a very small amount of data.

Language F cannot express the whole of the problem (neither can DSL ALPHA). We will content ourselves with obtaining

all person records who could operate machine "X" between dates "A" and "B". There is no operation to select an arbitrary member from this set to complete the retrieval. The set can be defined as follows:

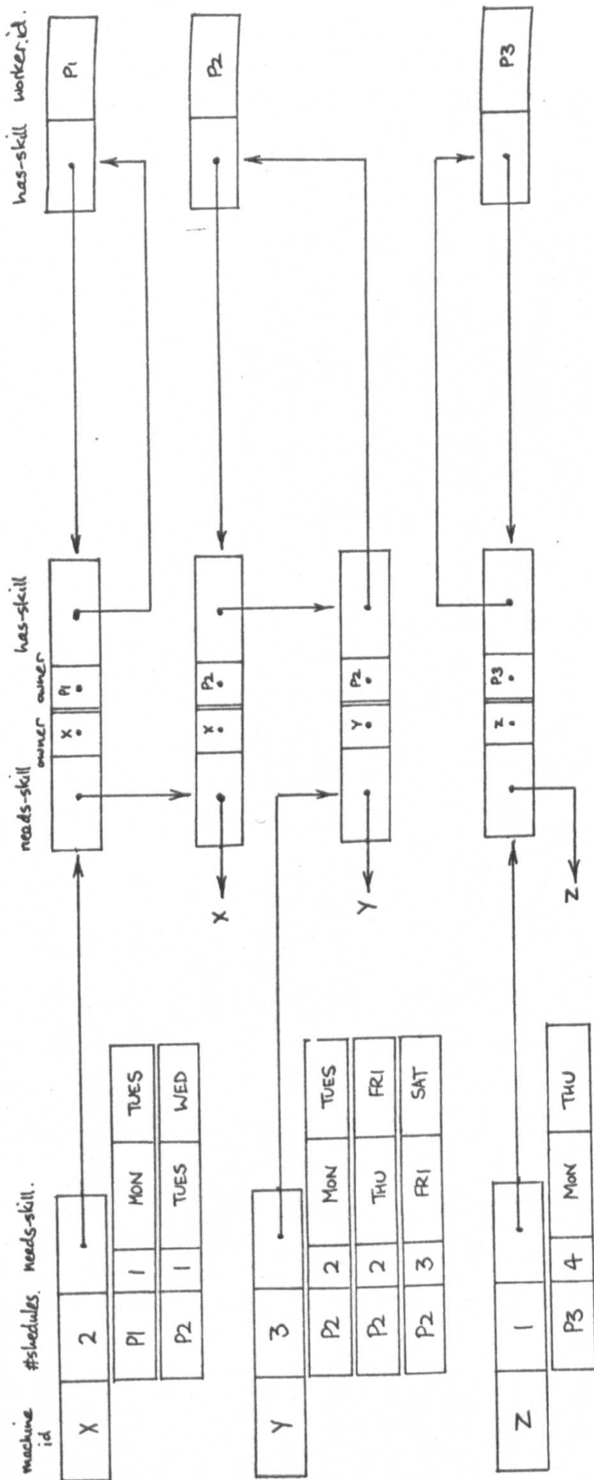
```
let busy-between(p,start,end)
  = some {s|
      end>shed-start-date(s) and
      start<sched-stop-date(s) and
      worker-id(s)=identification-num(p) and
      can-operate(p,sched-machine(s))}
 {p:persons|can-operate(p,"X") and not busy-between(p,"A","B")}
```

This is very similar to the DSL ALPHA program. One minor difference is that there is no explicit reference to skills. These will be introduced in the definition of can-operate. A much more significant difference is the appearance of:

can-operate(p,sched-machine(s))

in the definition of "busy-between". This does not appear in the DSL ALPHA program at all. It is an invariant, and reflects the fact that anyone scheduled to operate a machine will be capable of doing so. It must be included here because the operation of the COBOL program relies on its being true.

Of the identifiers used in the program, only the predicate "can-operate" and the function "sched-machine" are not supported directly by the representation. These must be




```
type sys-machine = sparse-array (string) of machine-rec ;
type machine-rec = (machine-id: string ; schedule-count: integer;
                    needs-skill : sparse powerset (skill-link);
                    schedules : sparse powerset (schedule));
type schedule    = ( worker-id : string; job-code : integer;
                    start-date : integer ; stop-date : integer);
type skill-link  = ( needs-skill-owner : machine-rec;
                    has-skill-owner : person-rec);
type person-rec  = ( worker-id : string ;
                    has-skill : sparse-powerset (skill-link));
```

possible type description.

Figure 6.2 Sample scheduling data

expressed in terms of structures which do exist.
The function sched-machine (MACH# in the relational form) must be expressed in terms of the machine-record in which the schedule occurs:

sched-machine(s) = <u>the</u> {m:machines | s <u>in</u> schedules(m)}

The field "schedules" in a machine record contains all the schedules for the machine. In the COBOL it is a repeating group.

The predicate "can-operate" is represented by the skill-link records and the chains through these. It is represented redundantly:

can-operate(p,m)
≡ <u>one</u> {k:skills | person(k)=p <u>and</u> k <u>in</u> skills-needed(m)}
≡ <u>one</u> {k:skills | machine(k)=m <u>and</u> k <u>in</u> has-skill(p)}.

The function <u>one</u> is the external form of the restricted existential quantifier. It produces the same result as <u>some</u>, but additionally assumes that the argument set contains at most one member. (It is possible to do without <u>one</u> by re-arranging the definition, but little advantage is gained.) The definition says that a person can operate a machine m if one of the set of skill records obtained through the "skills-needed" field of m identifies person p,

or equivalently, if one of the skill-records obtained from the "has-skill" field of p identifies machine m.

The complete compilation was run as an unseen test case in the following way:

22.

```
  PRIORITY CAN_OPERATE 3 3;  (defines CAN_OPERATE as an infix
                    operator with left and right priority 3)
  LET P CAN_OPERATE M
     <= ONE K:SKILLS | PERSON(K)=P AND K IN SKILL_NEEDED(M)
     <= ONE K:SKILLS | MACHINE(K)=M AND K IN HAS_SKILL(P);

  LET SCHED_MACHINE(S)
          <= THE M:MACHINES | S IN SCHEDULES(M);

  LET BUSY_AT (P,START,END)  <=
     SOME S:SCHEDS |
       START>SCHED_STOP_DATE(S) AND
       SCHED_START_DATE(S)>END AND
       WORKER_ID(S) = IDENTIFICATION_NUM(P) AND
       P CAN_OPERATE SCHED_MACHINE(S);

   P:PERSONS | P CAN_OPERATE "X" AND NOT BUSY_AT(P,"A","B");
    =>
     FOR K IN SKILL_NEEDED("X") DO
      LET P=PERSON(K);
       FOR K IN HAS_SKILL(P) DO
        FOR S IN SCHEDULES(MACHINE(K)) DO
         IF "A">SCHED_STOP_D(S) THEN
          IF SCHED_START_(S)>"B" THEN
           IF WORKER_ID(S)=IDENTIFICATI(P) THEN
            GOTO L1
           FI
          FI
         FI
        OD
       OD;
       WRITE(P)
     L1:
     OD.
```

The resulting program matches the operation (but not the form) of the COBOL program quite accurately. It first locates all the skill records associated with machine X. For each the associated person record is found. This

person could operate the machine. To find if the person is busy, the invariant is utilised. All the machines he could operate are inspected. This is done by locating all the skill-link records chained from the person record and from these finding the machines. The schedules in each machine are inspected for conflicing dates and matching identification numbers. If a conflicting schedule is found, the person is rejected via the branch to L1 and the next person tried. (The COBOL program seems to find the next candidate in a slightly different way.)

The process potentially produces all suitable operators. To satisfy the original problem definition the search would have to be terminated after the first candidate had been found.

<u>Examples of update</u>.

The remaining examples show the effect of some simple assignments on data-structures which are stored redundantly.

```
LET UNAVAILABLE(P) <=OUT_OF_STOCK(P) AND OBSOLETE_PART(P);
MAINTAIN UNAVAILABLE(P);
```

The function UNAVAILABLE is first defined to reflect the instantaneous value of the right-hand expression. The MAINTAIN statement indicates to the system that we wish to

consider the effect of keeping its value in storage. Subsequent assignments which effect its value must be followed by an appropriate modification of the stored data.

23.

```
 OBSOLETE_PART("4BA_NUT"):=TRUE;
  =>
   IF OUT_OF_STOCK("4BA_NUT") THEN
    UNAVAILABLE("4BA_NUT"):=TRUE
   FI;
   OBSOLETE_PAR("4BA_NUT"):=TRUE.
```

The code shows the pair of statements which are needed to preserve the truth of the definition.

24.

```
OUT_OF_STOCK("4BA_NUT"):=FALSE;
  =>
   IF OBSOLETE_PAR("4BA_NUT") THEN
    UNAVAILABLE("4BA_NUT"):=FALSE
   FI;
   OUT_OF_STOCK("4BA_NUT"):=FALSE.
```

The code for deletion is symmetric, although the test is in this case not strictly necessary.

The next example illustrates the code generated to maintain the value of a disjunctive expression.

25.

```
LET DONT_USE(P)  <= OUT_OF_STOCK(P) ANDOR OBSOLETE_PART(P);

MAINTAIN DONT_USE(P);

OUT_OF_STOCK("4BA_NUT"):=TRUE;
  =>
   IF ¬OBSOLETE_PAR("4BA_NUT") THEN
    DONT_USE("4BA_NUT"):=TRUE
   FI;
   OUT_OF_STOCK("4BA_NUT"):=TRUE.
```

Notice that the assignment is made to DONT_USE only if its

value needs to be changed. In the next example, the definition of DONT_USE is retained, but OUT_OF_STOCK is defined in terms of NUMBER_ON_HAND.

26.
```
LET OUT_OF_STOCK(P)   <= NUMBER_ON_HAND(P)=0;

NUMBER_ON_HAND("4BA_NUT"):=N;
  =>
   IF ¬0=N THEN
    IF ¬OBSOLETE_PAR("4BA_NUT") THEN
     DONT_USE("4BA_NUT"):=FALSE
    FI
   FI;
   IF 0=N THEN
    IF ¬OBSOLETE_PAR("4BA_NUT") THEN
     DONT_USE("4BA_NUT"):=TRUE
    FI
   FI;
   NUMBER_ON_HA("4BA_NUT"):=N.
```

The first statement in the sequence removes the effect of the original value of NUMBER_ON_HAND. The second statement changes DONT_USE to account for the new value. The first two statements could be combined into an if-then-else structure. An early design decision makes this rather difficult to do in the current implementation however.

The remaining examples show, almost without comment, the code produced by the system for the illustration used in the latter part of chapter 5.

27.

```
LET FLIGHTCAP(F:FLIGHTS)  <= CAPACITY(ASSIGNED_AC(F));

LET BOOKINGS(F)    <= B|FLIGHT(B)=F;

LET NUMBOOKED(F)   <= NUMBER BOOKINGS(F);

 MAINTAIN N=NUMBOOKED(F);

 FLIGHT("MYTICKET"):= "BE300";
  =>
   LET F=FLIGHT("MYTICKET");
    IF ¬F="BE300" THEN
     NUMBOOKED(F):=NUMBOOKED(F)-1
    FI;
   IF ¬FLIGHT("MYTICKET")="BE300" THEN
    NUMBOOKED("BE300"):=NUMBOOKED("BE300")+1
   FI;
   FLIGHT("MYTICKET"):="BE300".
```

The code produced suggests that equality should be given special case treatment in the strength reduction algorithm, as the outer two tests could be made common.

28.

```
 MAINTAIN N=FLIGHTCAP(F);

 CAPACITY("CONCORD1"):= 90;
  =>
   FOR F IN FLIGHTS DO
    IF ASSIGNED_AC(F)="CONCORD1" THEN
     FLIGHTCAP(F):=90
    FI
   OD;
   CAPACITY("CONCORD1"):=90.

 ASSIGNED_AC("BE300"):= "VISCOUNT1";
  =>
   FLIGHTCAP("BE300"):=CAPACITY("VISCOUNT1");
   ASSIGNED_AC("BE300"):="VISCOUNT1".
```

Finally:

```
QUIT
COMPILE TIME: 0.16 SECS.
0000.86SECONDS IN EXECUTION
```

The "compile-time" given is the time spent in the central code generation stage of the system. Most of the remaining time used in the set of examples is spent printing the resulting programs.

CHAPTER 7

CONCLUSIONS AND FURTHER WORK.

The investigation began by considering the effect of the data organisation on the execution of a program. The aim has been to find a practical way of adapting a program to a representation and ultimately to enable assistance to be given in the choice of a data organisation.

Aggregate operations were investigated as a method of overcoming the problems encountered with recursively defined programs, although this has meant that only a limited class of programs could be covered. The interest in data base applications suggested the use of relations to describe the data and the algebra of relations to specify the computation. However in spite of the considerable number of existing implementations based on this principle, the standard relational description developed by Codd did not prove an ideal implementation tool. Its principle disadvantages can be summarised as follows.

In a normal form relational model, all the data with the same key appears in the same relation. For example one relation contains all the properties of "parts". The key field might be the part-numbers. Relations therefore tend to have a large number of domains, each tuple

corresponding approximately to a record in a file. Although this does reflect one possible physical organisation of the data, the relationship with other organisations (for example using networks) which lay the fields out differently can be quite complex, and does not lend itself easily to mechanical processing.

The relational algebra is used to specify the computation of new relations from existing ones. The algebra uses domain numbers to identify elements in a tuple and this makes the manipulation of expressions rather difficult. Although the relational calculus (similar to DSL ALPHA) does use domain names and does not rely on their ordering, a direct implementation using the algebraic operations would not be very efficient.

More fundamentally, the operations are only defined for "flat" or unstructured sets of tuples. They do not recognise the special case which arises when the relation is, in fact, a function. For example this leads to uses of the general project operator when none need occur, and to consequent inefficiency. Also, some of the operations we need more naturally use array-like structures rather than unstructured sets. One example of the effect of using only relations is found in the definition of the relational "divide" operator, which performs the equivalent of a finite universal quanitifier. This can be

used to express the query: "find the manufacturers who give a discount on all parts used in assembly A" but not to express: "find the manufacturers who give a discount for all parts <u>they supply</u> to assembly A". In the first case we can first find the parts in assembly A and then determine, by the divide operation, manufacturers who supply all members of the set. The difficulty with the second example is that the set of parts to be used in the divide is not a constant, but depends on the manufacturer in question. A divide operation which covered this (admittedly unusual) situation is not easy to define and probably needs four domain numbers as parameters in addition to the two relations. Essentially the operation acts in general on arrays of sets and not simply on relations.

Since relations are sets, the fundamental update operations are the addition and deletion of members. The commonly occurring operation to modify an existing member can be expressed by a combination of an addition and a deletion, but because this pair of operations can in general have other effects (it may alter the number of members in the relation), to treat selective update in this way does not produce an efficient implementation. Again, when considering methods to optimise the update of redundant data structures, it is essential to recognise the special case of projection which results from a

functional application. While this can be optimised the more general form can not be. Attempts to deduce the necessary information, show that while it may be possible to do this (e.g. Delobel and Casey 1972) it is quite a complex process.

These conclusions are perhaps confirmed by the existing implementations, which tend to store each relation more or less directly, make few claims about their efficiency (but see Titman (Titman 1974)), and provide little support for update.

A number of relationally based mini-languages were developed in an attempt to overcome these problems. The most recent of these, Language F, has been described. It was designed to allow a limited, but hopefully typical range of data base operations to be expressed and to be processed with reasonable efficiency. The principle changes made during its development can be summarised as follows.

First we have assumed a more primitive type relation, similar to those used by Abrial. Each relation corresponds not to a complete record, but approximately to a single field in a record. Also relations can contain items other than simple integers or strings. This makes it possible to connect the relational description and the

physical data structure by a relatively simple abstraction function. However, using these more primitive relations does not prevent the construction of a standard relational view of the data if this is needed. (For example see (Bracchi 1974).)

The second change was to re-define the operations to remove their dependence on column ordering and to match them exactly to the operators in the standard calculus. While this does not add anything essential, it does make the manipulation of expressions easier and means that only a small step is needed to add array-like structures as functions.

The addition of functions was perhaps more fundamental than was at first realised. Initially, the motivation was to be able to use the special case of the project operator which results from expanding a functional application. This has a much simpler implementation than the general purpose operator. However, with functions which return predicates (or sets) we can describe hierarchical data organisations. Not only does this mean that common physical structures can be described, but also that the description of the computation can use structures which are not normalised relations. We can use functions like <u>number</u> which act on arrays of sets. Although no equivalent of the relational divide operator was included

in Language F, we could readily introduce such an operation. For example, all(s,p) could return "true" if all members of set s satisfy predicate p. (The implementation is very similar to some , in a simple case all values satisfying p are generated, and tested for inclusion in s. It was omitted from the language because of this similarity.) The operation can be used to express any example of "all" and would not be restricted in the same way as the relational divide.

A further advantage of functions is that they allow a natural selective update. We can arrange to assign to a single array element and do not need the artifice of a matched deletion and addition, with its attendent problems.

In summary, there is no need to insist that the data be viewed as a normalised relational structure. One can think of the relation Supplies as a set, a predicate, an array giving the suppliers of each part, an array giving the set of parts supplied by each manufacturer, whichever is convenient. The actual data-structures which are stored can be quite unrelated to the view which is taken and the system will make the conversion. One may, of course, feel that such a disregard of the actual storage structure is dangerous, because the cost of performing a retrieval may not be related in any way to the complexity

of its specification. However this is inherent in any system which attempts to insulate the user from the physical data organisation.

An experimental compiler has been written for the language which produces Algol-like programs. This superceded an earlier interpreter so that the intimate details of the data accessing algorithms could be ignored. In principle the compiler works by applying the standard rules of functional application and equivalences from predicate logic to obtain a series of equivalent expressions in terms of stored functions and sets. These are tested in turn until one is found which can be turned directly into a program to construct the result. Within its limitations, this has been found to work quite well. Two sources of early concern, that compile times would be very long and that a large amount of information about the sizes of sets would be needed to select a program, have not, at least in the experiments, proved to be serious problems. The simplest possible sequencing algorithm has proved adequate for the examples tried, and only in a few instances has extra information been strictly needed to choose the code.

While the findings must be considered to be of a preliminary nature, the experiments suggest that it is practicable to use this method to adapt programs to a

given data representation, or to convert between one representation and another. A high-level data-base system could well operate along the lines suggested and make reasonably efficient use of a given storage structure. In the longer term, a similar system might be useful to outline the effect of a chosen data organisation even when the application is ultimately to be hand-coded.

A great deal of work remains to be done, both in the short and the long term. Immediately, the compiler has a number of known deficiencies, the most obvious of these being the lack of symmetric merge and collate operations. These were omitted from the program printing routine because their inclusion seemed to make the resulting programs unnecessarily complex. Some work has been done toward the addition of a simple printing method, and only small changes are needed in the remainder of the compiler to make the necessary distinction between sorted and un-sorted sets.

Neither the input language, nor the output language were consciously designed. Rather, they have gradually evolved in response to various pressures. As the input language is no longer purely a relational one, further understanding would probably result from re-designing it, and this should be the next major step. The aim of the new design would be mainly to rationalise the underlying

structure, to eliminate all dependence on size information and to allow the definition of functions such as <u>number</u> which currently must be built-in.

The reluctance to accept the short-comings of the standard relational treatment left rather to little time to study the alternative proposed in depth. For instance, it is not clear what programs can, and what programs cannot, be compiled. This depends both on the types of statement which can appear in the output and on the alternative transformations that the compiler will try. As both have been changed many times, no attempt has yet been made to document its detailed input rules. A re-design of the input language and processing algorithms along cleaner lines would give an opportunity to rectify this omission. Also, in the interests of progress, the output language and consequently the transformations are entirely informal. A more stable design would allow for more rigour.

The experiments so far have given some insight into the interaction between the storage structure specified and the processing method which results. However much more experience is needed to understand clearly what governs the choice of data organisation to suit a given suite of programs, and the space and time trade-offs which are often involved. The availability of a richer set of

operators (such as "all","maximum" and so on) would mean that a large number of more realistic examples could be tried and a model of the behavior developed.

It has been noticed that one of the factors influencing the choice of data organisation are the consistency constraints on the data. They must be guaranteed after an update (affecting not only the storage structure, but also for example, the possible ways of synchronising concurrent operations). Also, as example 22 of chapter 6 shows, use can be made of them on retrieval. In Grindlay and Stevens' Systematics (Grindlay and Stevens 1968), consistency conditions are used to specify the output from an information system. For example, we might like the purchasing department of our factory to be permanently aware of the parts which are out-of-stock. They should be informed whenever a part becomes out-of-stock so that they can take appropriate action, and similarly be told when the position is rectified, so that the action can be stopped. Exactly the same methods can be used to maintain the consistency of the data in the purchasing department with that in the central data-base as is used to keep consistency in a redundant storage structure. It is felt that a re-design of the input language should include a deeper look at these invariant conditions, their various uses and the implications that these might have.

Finally a more complete prototype system should be built. This could well interface with an existing programming system designed to handle abstract programs, such as TOPD (Henderson et al. 1975). It would then be possible to judge whether the performance and code quality can be maintained in a more realistic environment.

APPENDIX A

Summary of implemented system

The following four tables show an approximate concrete syntax, the priority of the operators, the transformations done by the pre-processor and the code generation rules. The strength-reduction rules for assignment follow.

The syntax is expressed in the extented Backus-Naur form originally used in the Vienna definition of PL/1 (Urschler 1969). The conventions used are:

(i) Non-terminal symbols are represented simply by lower-case words.

(ii) Alternatives, usually placed on separate lines, are separated by |. (⊥ stands for the terminal symbol |.)

(iii) { } are used as meta-syntactic brackets.

(iv) [ ] indicate that the enclosed phrase is optional.

(v) ... show that the preceding phrase may occur any non-zero number of times.

```
program        ::= [statement-list] QUIT

statement-list ::= statement ; [statement-list]

statement      ::= let-definition    |
                   set-to-compile    |
                   assignment        |
                   maintain-clause   |
                   system-command

let-definition ::= LET function [parameter]   {<=expression} ...

expression     ::= set-exp  |  predicate-exp  |  valued-exp

set-to-compile ::= set-exp

predicate-exp  ::= predicate-function [argument]                  |
                   predicate-exp {AND|OR|ANDOR} predicate-exp     |
                   NOT predicate-exp                              |
                   valued-exp  {= | >= | >}  valued-exp           |
                   valued-exp IN set-exp                          |
                   SOME set-exp

set-exp        ::= set-function [argument]      |
                   parameter ⊥ predicate-exp

valued-exp     ::= valued-function [argument]                |
                   valued-exp  {+ | - | *}  valued-exp       |
                   NUMBER set-exp                            |
                   THE set-exp                               |
                   SUM(set-exp,valued-function)              |
                   identifier | string-constant | integer

parameter      ::= identifier             |
                   identifier : set-exp   |
                   ( [parameter-list] )

parameter-list ::= parameter [, parameter-list]

argument       ::= valued-exp            |
                   ( [argument-list] )

argument-list  ::= argument [,argument-list]

maintain-clause::= MAINTAIN predicate-exp

assignment  ::= predicate-function [argument] := predicate-exp  |
                   valued-function [argument] := valued-exp     |
                      set-function [argument] := set-exp

system-command ::= ECHO  {ON | OFF} |
                   PRIORITY identifier integer integer
```

<u>Implementation syntax</u>

operator	left	right
OR ANDOR	11	11
AND &	10	10
NOT		9
= > >=	8	8
+ -	7	7
*	6	6
SOME NUMBER		2
ONE THE		2
..	1	1
|	2	12
IN	8	2
:	1	2
:=	12	2

Built-in operator priorities

i. function definitions.

```
          LET f(x) <= L1 <= L2 <= L3 .... ; M

                              => M where f(x)=L1
                              => M where f(x)=L2
                                 .....
```

ii. sets.

```
          {x:S|P}             => {x|P}
                              => {x|x in S and P}

          M in {x|P}          => P where x=M
```

iii. nested functions.

```
          P(f(x))             => P(the{t|t=f(x)})

          r=f(g(x))           => r=f(the{t|t=g(x)})
```

iv. iota removal.

```
          M=the{x|p}          => P where x=M

          Q(the{x|P})         => one{x|Q(x) and P(x)}
```

v. distribution etc.

```
          (P or Q) and R      => (P or Q) and R
                              => (P and R) or (Q and R)

          (P and Q) and R     => (P and Q) and R
                              => (P and R) and Q

           P and Q            => P and Q
                              => Q and P
```

Pre-processing transformations

```
i. primitive statements.

   for x|x in S do C od         => for x in S do C od

   for x|x=e do C od            => let x=e; C

   for ()|P do C od             => if P then C fi

ii. logical expressions.

   for x,y|P and Q do C od => for x|P do for y|Q do C od od

   for x|P or Q do C od         => for x|P do C od; for x|Q do C od

   for x|P andor Q do C od

                                => for x|P do C od, for x|Q do C od

   if P and Q then C1 else C2 fi

                         => if P then if Q then C1 else C2 fi else C2 fi

   if P or Q then C1 else C2 fi

                          => if P then C1 else if Q then C1 else C2 fi fi

   if not P then C1 else C2 fi => if P then C2 else C1 fi

iii. set expressions.

   if some S then C1 else C2 fi

                                => for x in S do C1;goto L od; C2; L:

   for x|some S do C od         => let T=empty;

                                    for x,y| y in S do T:=T union x od;

                                    for x in T do C od

   let A=number S; C            => let A=0;

                                   for y,a| y in S do A:=A+1 od; C

   let A=sum(S,f); C            => let A=0;

                                   for y,a| y in S do A:=A+f(y) od; C

   for r,y| r=number S do C od => let A(y)=number S;

                                     for r:domain(a),y|r=A(y)  do C od
```

Code generation summary

	definition	Update	Result
		for x|t do	
1.	R ≡ P and Q	P:=V	for x∪r|T and Q do R:=V od
2.	R ≡ P or Q	P:=V	for x∪r|T and not Q do R:=V od
3.	R ≡ not Q	P:=V	for x∪r|T do R:=not V od
4.	R = {y|P}	P:=true	for x∪r|T and not y in R do R:less y od
		P:=false	for x∪r|T and y in R do R:plus y od

		P:=V	for x∪r|T and y in R do
			R:less y
			od;
			for x∪r|T and V do
			R:plus y
			od

5.	R = number S	S:plus y	for x|T do R:=R+1 od
		S:less y	for x|T do R:=R+1 od
6.	R = sum (S,F)	S:plus y	for x|T do R:=R+F(y) od
		S:less y	for x|T do R:=R-F(y) od
7.	R ≡ some S	S:plus y	for x|T do R:=true od
		S:less y	for x|T do R:=some Sod
8.	R = F(E)	E:=V	for x|T do R:=F(V) od
		F(E'):=V	for x|T and E=E' do R:=V od
		od	

Assignment code summary

APPENDIX B

Original solutions to scheduling example

The following two programs, one in DSL ALPHA and one in DBTG Cobol are reproduced from (Halstead 1974).

a) DSL ALPHA

```
GET (into workspace) W (at most) (1) PERSON-SKILL.P#:
    EXIST MACH-SKILL (with)
          (MACH-SKILL. MACH# = X)
          & (MACH-SKILL. SKILL# = PERSON-SKILL. SKILL#)
          & NOT EXIST SCHED (with)
                (SCHED. P# = PERSON-SKILL. P#)
                & (SCHED. SCHED-START-DATE LESS-THAN B)
                & (SCHED. SCHED-STOP-DATE GREATER-THAN A)
MOVE W INTO SCHED-RECORD              (host language)
PUT SCHED-RECORD SCHED.
```

b) DBTG Cobol

```
PROCEDURE DIVISION.
    OPEN PANDJ-AREA, WITH-HOLD, REST.
FIND-MACHINE.
    OPEN XP.
    MOVE MACHINE-NUMBER TO MACH-NUMBER.
    FIND MACHINE-RECORD VIA SYS-MACHINE USING MACH-NUMBER.
    IF ERROR-STATUS = 326 GO TO NOT-IN-DATA-BASE.

FOUND-REC.
    MOVE CURRENCY STATUS FOR MACHINE RECORD TO SAVE-MACHINE.

GET-NEXT-SKILL.
    FIND NEXT SKILL-LINK RECORD OF NEEDS-SKILL SET.
    IF ERROR-STATUS = 326 OR 307 GO TO NO-ONE-AVAILABLE.
    FIND OWNER IN HAS-SKILL SET OF CURRENT OF SKILL-LINK RECORD.
    IF ERROR-STATUS = 322 THEN GO TO GET-NEXT-SKILL.
    MOVE CURRENCY STATUS FOR PERSON RECORD TO SAVE-PERSON.
    MOVE CURRENCY STATUS FOR PERSON RECORD TO CHECK-PERSON-ITEM.
    STORE CHECK-PERSON.
    IF ERROR-STATUS = 1025 GO TO GET-NEXT-SKILL.

CHECK-PERSON-SCHEDULE.
    FIND NEXT SKILL-LINK RECORD OF HAS-SKILL SET;
         SUPPRESS NEEDS-SKILL CURRENCY UPDATES.
    IF ERROR-STATUS = 307 GO TO PERSON-IS-FREE.
    FIND OWNER IN NEEDS-SKILL OF CURRENT OF SKILL-LINK RECORD;
         SUPPRESS NEEDS-SKILL CURRENCY UPDATES.
    IF ERROR-STATUS = 322 GO TO CHECK-PERSONS-SCHEDULE.
    MOVE CURRENCY STATUS FOR MACHINE RECORD TO CHECK-MACHINE ITEM.
    STORE CHECK-MACHINE.
    IF ERROR-STATUS = 1025 GO TO CHECK-PERSONS-SCHEDULE.
    GET MACHINE.
    MOVE 1 TO AVAILABLE.
    PERSON SEE-IF-SCHEDULED THRU SEE-EXIT VARYING
         SCHEDULED-COUNT FROM 1 BY 1 UNTIL SCHEDULE-COUNT
         IS GREATER THAN SCHEDULE.
    IF AVAILABLE = 0 GO TO GET-NEXT-SKILL.
GO TO CHECK-PERSONS-SCHEDULE.

SEE-IF-SCHEDULED.
    IF SCHEDULE-START-DATE IS GREATER THAN SCHEDULE-START
         IN MACHINE (SCHEDULE-COUNT) AND LESS THAN
         SCHEDULE-COMPLETION IN MACHINE (SCHEDULE-COUNT)
              GO TO PERSON-NOT-AVAILABLE.
    IF SCHEDULE-DATE-END IS GREATER THAN SCHEDULE-START
         IN MACHINE (SCHEDULE-COUNT) AND LESS THAN
         SCHEDULE-COMPLETION IN MACHINE (SCHEDULE-COUNT)
         GO TO PERSON-NOT-AVAILABLE.
    GO TO SEE-EXIT.
```

```
PERSON-NOT-AVAILABLE.
   FIND PERSON USING SAVE-PERSON;
        SUPPRESS ALL CURRENCY UPDATES.
   GET PERSON.
   IF IDENTIFICATION-NUM IN PERSON IS EQUAL
        WORKER-IDENTIFICATION IN MACHINE(SCHEDULE-COUNT)
        MOVE 0 TO AVAILABLE, GO TO SEE-EXIT.
   MOVE WORKER-IDENTIFICATION IN MACHINE (SCHEDULE-COUNT)
        TO IDENTIFICATION-NUM IN PERSON.
   FIND PERSON RECORD,
        SUPPRESS HAS-SKILL CURRENCY UPDATES.
   MOVE CURRENCY STATUS FOR PERSON RECORD
        TO CHECK-PERSON ITEM.
   STORE CHECK-PERSON.

SEE-EXIT. EXIT.

PERSON-IS-FREE.
   FIND MACHINE USING SAVE-MACHINE.
   GET MACHINE.
   FIND PERSON USING SAVE-PERSON.
   GET PERSON.
   ADD 1 TO SCHEDULE IN MACHINE.
   MOVE IDENTIFICATION-NUM IN PERSON TO WORKER-IDENTIFICATION
        IN MACHINE (SCHEDULE IN MACHINE).
   MOVE SCHEDULE-START-DATE TO SCHEDULE-START IN MACHINE
         (SCHEDULE IN MACHINE).
   MOVE SCHEDULE-DATE-END TO SCHEDULE-COMPLETION IN MACHINE
         (SCHEDULE IN MACHINE).
   MOVE SCHEDULE-TASK TO JOBCODE IN MACHINE (SCHEDULE IN
         MACHINE).
   MODIFY MACHINE.
   IF ERROR-STATUS = 803
         GO TO PERSON IS FREE.
   CLOSE XP.  GO TO GET-NEW-MACHINE.
```

APPENDIX C

UPDATE EQUIVALENCES

In the following we use the fact that for logical values, <u>update</u>(p,t,r) reduces to (t <u>and</u> r) <u>or</u> (<u>not</u> t <u>and</u> p). This follows from the definition of <u>update</u> in terms of <u>if</u>-<u>then</u>-<u>else</u>.

1. For conjunctions we use:

P <u>and</u> <u>update</u>(Q,t,r)≡<u>update</u>(P <u>and</u> Q,P <u>and</u> t,r)

To show this, it is convenient to take the right-hand-side. Then by using simple propositional logic:

<u>update</u>(P <u>and</u> Q,P <u>and</u> t,r)

≡(P <u>and</u> t <u>and</u> r) <u>or</u> (<u>not</u> (P <u>and</u> t) <u>and</u> P <u>and</u> Q)

≡(P <u>and</u> t <u>and</u> r) <u>or</u> ((<u>not</u> t <u>or</u> <u>not</u> P) <u>and</u> P <u>and</u> Q)

≡(P <u>and</u> t <u>and</u> r) <u>or</u> <u>not</u> t <u>and</u> P <u>and</u> Q

≡P <u>and</u> ((t <u>and</u> r) <u>or</u> (<u>not</u> t <u>and</u> Q))

≡P <u>and</u> <u>update</u>(Q,t,r)

2. For disjunctions we use:

P <u>or</u> <u>update</u>(Q,t,r)≡<u>update</u>(P <u>or</u> Q,t <u>and</u> <u>not</u> Q,r)

Again, taking the right-hand-side:

<u>update</u>(P <u>or</u> Q,t <u>and</u> <u>not</u> Q,r)

≡t <u>and</u> <u>not</u> Q <u>and</u> r <u>or</u> (t <u>and</u> <u>not</u> Q) <u>and</u> (P <u>or</u> Q)

≡t <u>and</u> <u>not</u> Q <u>and</u> r <u>or</u> (<u>not</u> t <u>or</u> Q) <u>and</u> (P <u>or</u> Q)

≡t <u>and</u> <u>not</u> Q <u>and</u> r <u>or</u> <u>not</u> t <u>and</u> P <u>or</u> Q

≡(t <u>and</u> r <u>or</u> <u>not</u> t <u>and</u> P) <u>or</u> Q

≡<u>update</u>(P,t,r) <u>or</u> Q

3. For negation we use:

<u>update</u>(P,t,r)≡<u>update</u>(<u>not</u> P,t, <u>not</u> r)

As before, expanding the right-hand-side:

<u>update</u>(<u>not</u> P,t, <u>not</u> r)

≡ <u>not</u> r <u>and</u> t <u>or</u> <u>not</u> t <u>and</u> <u>not</u> P

≡ <u>not</u> (<u>not</u> (<u>not</u> r <u>and</u> t) <u>and</u> <u>not</u> (<u>not</u> t <u>and</u> <u>not</u> P))

≡ <u>not</u> ((r <u>or</u> <u>not</u> t) <u>and</u> (t <u>or</u> P))

≡ <u>not</u> (r <u>and</u> t <u>or</u> r <u>and</u> P <u>or</u> <u>not</u> t <u>and</u> P)

≡ <u>not</u> (r <u>and</u> t <u>or</u> r <u>and</u> P <u>and</u> t

<u>or</u> r <u>and</u> P <u>and</u> <u>not</u> t <u>or</u> <u>not</u> t <u>and</u> P)

≡ <u>not</u> (r <u>and</u> t <u>or</u> <u>not</u> t <u>and</u> P)

≡ <u>not</u> <u>update</u>(P,t,r).

REFERENCES

P.S.Abrams (1970)

An APL Machine. Doctoral Dissertation, University of Stanford. (Issued by Digital Systems Laboratory, Technical Report No.3).

J.R.Abrial (1974)

Data Semantics. In "Data Base Management", Ed. Klimbe and Koffeman, North Holland, page 1 ff. (From proceedings of IFIP TC.2 Working Conference, Corsica,April 1974).

A.V.Aho,J.E.Hopcroft,J.D.Ullman (1975)

The design and analysis of Computer Algorithms. Adison-Welsley. October 1975.

Algol W. (1972)

Algol - W Programming Manual. Computing Laboratory, University of Newcastle-upon-Tyne.

F.E.Allen (1969)

Program Optimisation. Annual Review in Automatic Programming, Volume 5. Pergamon Press. p.261.

M.M.Astrahan, D.D.Chamberlin (1972)

Implementation of a Structured English Query Language. IBM Research Report, RJ1072. San Jose, July 1972.

R.Bayer (1971)

Lecture notes to the International Summer School. Munich, July 1971. Part II, p.12.

R.Bayer, E.M.McCreight (1973)

Organisation and Maintenance of large ordered Indices. Acta Informatica Vol.1 no.3, March 1973, p.173.

S.M.Bernard (1970)

System/360 Report Program Generator. Prentice Hall, New Jersey.

R.F.Boyce et al.(1973)

Satisfying Queries as Relational Expressions: SQUARE. IBM Research Report, RJ1291, San Jose, October 1973.

G.Bracchi, A.Fedeli, P.Paolini (1972 a)

A language for a relational data-base management system. Proceedings Sixth Annual Princeton Conference on Information Sciences and Systems. March 1972.

G.Bracchi, A.Fedeli, P.Paolini (1972 b)

The Architecture of an Online Information Management System. Proceedings of the ONLINE 72 International Conference. Brunel University, September 1972.

G.Bracchi, A.Fedeli, P.Paolini (1974)

A multi-level Relational Model for data-base Management Systems. In "Data Base Management". Ed. Klimbe and Koffeman, North Holland. p.211.

W.H.Burge (1975)

Stream Processing Functions. IBM Journal of Research and Development. January 1975, p.12.

Burstall, Collins, Popplestone (1971)

Programming in POP2. Edinburgh University Press. p.30.

R.M.Burstall, J.Darlington (1974)

Systematic Development of Programs by introducing Economies of Interaction. Preliminary draft, circulated August 1974.

R.Carnap (1958)

Introduction to Symbolic Logic. Dover Publications, New York.

D.L.Childs (1968)

Feasibility of a set-theoretic data structure. Proceedings IFIP Congress, Edinburgh 1968. North Holland. Booklet 1, p.162.

Codasyl (1971)

Data base Task Group of the Codasyl Programming Language Committee. Report, April 1971.

D.D.Chamberlin, R.F.Boyce (1974)

SEQUEL: A Structured English Query Language. Proceedings 1974 ACM Sigfidet Workshop, Ann Arbor, Michigan. April 1974.

J.Cocke, J.Schwartz (1970)

Programming Languages and their Compilers. Courant Institute, New York.

E.F.Codd (1970)

A relational model for large shared data banks. Communications ACM. Vol.13 no.6. June 1970. p.377.

E.F.Codd (1971)

A data-base sub-language founded on the relational calculus. Proceedings of the 1971 ACM SIGFIDET Workshop on data Description Access and Control. San Diego.

E.F.Codd (1972 a)

Relational Completeness of data base sub-languages. IBM Research Report RJ987. San Jose, March 1972.

E.F.Codd (1972 b)

Further Normalisation of the data base relational model. In "Data Base Systems". Ed. Rustin. Courant Computer Science Symposia, Vol.6. Prentice Hall.

M.J.Cresswell (1973)

Logic and Languages. Methuen Press.

J.Darlington, R.M.Burstall (1973)

A system which automatically improves programs. Proceedings 3rd International Conference on Artificial Intelligence. S.R.I. p.479.

C.J.Date (1975)

An Introduction to data-base systems. Addison Wesley, New York.

C.Delobel, R.G.Casey (1973)

The decomposition of a data-base and the theory of Boolean Switching functions. IBM Journal of Research and Development Vol 17 no 5, September 1973.

E.W.Elcock et al.(1971)

ABSET. A programming language based on sets: motivation and examples. Machine Intelligence. Volume 6. Ed Michie, Edinburgh University Press, p.467.

J.J.Florentin (1972)

Consistency auditing of data bases. Draft paper, circulated August 1972.

J.M.Foster, E.W.Elcock (1968)

ABSYS 1: an incremental compiler for assertions. Machine Intelligence. Volume 3, Ed. Michie. Edinburgh University Press. p.423.

S.L.Gerhard (1974)

Correctness preserving program transformations. Proceedings second ACM/SIGPLAN Symposium on Principles of programming languages. 1974 p.54.

S.P.Ghosh, M.M.Astrahan (1974)

A translator Optimiser for obtaining answers to entity-set queries from an arbitrary access-path network. Information processing 74. North Holland, p.436.

P.C.Goldberg (1974)

Automatic Programming. IBM Research Report. RC5148, Yorktown, September 1974.

D.Gries (1971)

Compiler Construction for digital Computers. Wiley, New York, 1971. p.205.

C.B.B.Grinday, W.G.R.Stevens (1968)

Principles of the identification of information. Proceedings IFIP International Seminar on file organisation. North Holland, November 1968, p.60.

P.A.V.Hall (1974)

Common sub-expression identification in general algebraic systems. IBM (UK) Scientific centre report UKSC 0060.

P.A.V.Hall, S.J.P.Todd (1974)

Factorisation of algebraic expressions. IBM (UK) Scientific Centre report UKSC 0055, April 1974.

P.A.V.Hall, P.Hitchcock, S.J.P.Todd (1974)

An algebra of relations for machine computation. Second ACM SIGACT/SIGPLAN Symposium on principles of programming languages. 1974. p.225.

M.H.Halstead (1974)

Software physics comparison of a sample program in DSL ALPHA and Cobol. IBM Research Report RJ1460, San Jose, October 1974.

I.J.Heath (1972)

Unacceptable file operations on a relational data-base. IBM (UK) Technical Report. TR12.094. March 1972 p.57.

P.Henderson, J.Morris (1975)

The lazy evaluator. Unpublished draft, August 1975.

C.Hewitt (1969)

PLANNER: A language for proving theorems in robots. Proceedings of the International Joint Conference on Artificial Intelligence. Bedford, Mass. Mitre Corp. 1969. p.295.

C.A.R.Hoare (1969)

An exiomatic basis of computer programming. Communications ACM, Vol.12 no.10, October 1969, p.576.

C.A.R.Hoare (1972)

Notes on data structuring. In "Structured Programming", O-J Dahl, E.W.Dijkstra, C.A.R.Hoare. Academic Press, p.83.

I.D.S. (1968)

IDS Reference Manual. GE 625/535. G.E. Information Systems, Phoenix February 1968.

IMS/360 (1972)

IMS/360 General Information Manual. IBM Form Number GH20-0765-3 November 1972.

S.Kleene (1967)

Mathematical Logic. Wiley, p.154.

D.E.Knuth (1973)

The Art of Computer Programming Volume 3, "Sorting and Searching". Addison Wesley. p.205.

D.E.Knuth (1974)

Structured Programming with goto statements. Computing Surveys, Vol.6 no.4, December 1974, p.261.

R.Kowalski (1974)

Predicate Logic as a programming language. Information Processing 74. North Holland, p.569.

M.Krohn, R.Williamson (1972)

Towards an automatic system generator. Software 72. Transcript books, p.72.

J.L.Kuhns (1969)

Logical aspects of question answering by computer. Proceedings third international symposium on Information Sciences. Miami, December 1969. Academic Press.

P.J.Landin (1966)

The next 700 programming languages. Communications ACM, Vol.9 no.3 (March 1966) p.157.

J.Longstaff, F.Poole (1974)

A new reduction algorithm for a relational data-base. Tees-side Polytechnic report. Draft, Circulated, November 1974.

R.A.Lorie (1974)

X.R.M. An extended (N-ary) Relational Memory. IBM Scientific Centre Report, 320 - 2096 Cambridge (Mass). October 1974.

M.G.Notley (1972)

The Peterlee IS/1 System. IBM (UK) Scientific Centre Report UKSC 0018, March 1972.

I.Osman (1974)

Matching storage organisation to usage pattern in a Relational Data Base. Ph.D. Thesis, University of Durham, October 1974. p.29.

F.P.Palermo (1972)

A data base search problem. IBM Report RJ1072, San Jose, July 1972.

P.H.Prowse (1973)

The relational model as a systems analysis tool. BSC Symposium on relational data-base concepts. London, April 1973.

J.T.Schwartz (1970)

Set theory as a language for program specification and programming. Notes, distributed 1970.

M.E.Senko et al. (1973)

Data structures and accessing in data base systems, II - Information organisation. IBM Systems Journal, 1973 no.1, p.45.

P.M.Stocker, P.A.Dearnley (1973)

Self-organising data management systems. Computer Journal vol.16 no.2, (February 1973) p.100.

G.C.H.Sharman (1975)

A new model of relational data base and high level languages. IBM (UK) technical report TR 12-136, February 1975.

A.L.Strnad (1971)

The relational approach to the management of the data base. Proceedings IFIP Congress, Lubjana, 1971. Booklet 5 p.91.

P.J.Titman (1974)

An experimental data-base system using binary relations. In "Data Base Management", ed Klimbe and Koffeman, North Holland, p.351.

S.J.P.Todd (1975)

PRTV: A technical overview. IBM Scientific Centre Report UKSC 0075, May 1975.

F.B.Thomson et al (1969)

REL: A rapidly extensible language system. Proceedings 24th ACM National Conference, New York. p.399.

G.Urschler (1969)

Concrete Syntax of PL/I (ULD Version III). IBM Laboratory Vienna, Technical Report TR.096, June 1969.

H.Wedekind (1974)

On the selection of access-paths in a data base system. In "Data Base Management", ed Klimbe and Koffeman, North Holland, p.385. (Proceedings IFIP TC.2 working conference, Corsica, April 1974).

W.A.Woods, R.M.Kaplan, B.N.Weber (1972)

The Lunar sciences Natural language Information System. B.B.N. Report number 2378. June 1972.

Errata

Page 216 Insert:

R.M. Burstall, J. Darlington.

Some transformations for developing

recursive programs.

Proceedings International conference

on reliable software, Los Angeles April, 1975, p.465.

Page 219 Insert;

P. Henderson, J. Morris.

A lazy Evaluator,

Proceedings of the third A.C.M. symposiom

on the principles of programming languages,

Atlanta Georgia, January 1976 p.95

Page 222 Insert:

J. T. Schwartz.

Automatic and semi-automatic optimisation

in SETL.

Proceedings A.C.M. Sigplan Symposium on very-

high-level languages. Sigplan notices 9, 4 April 1974.