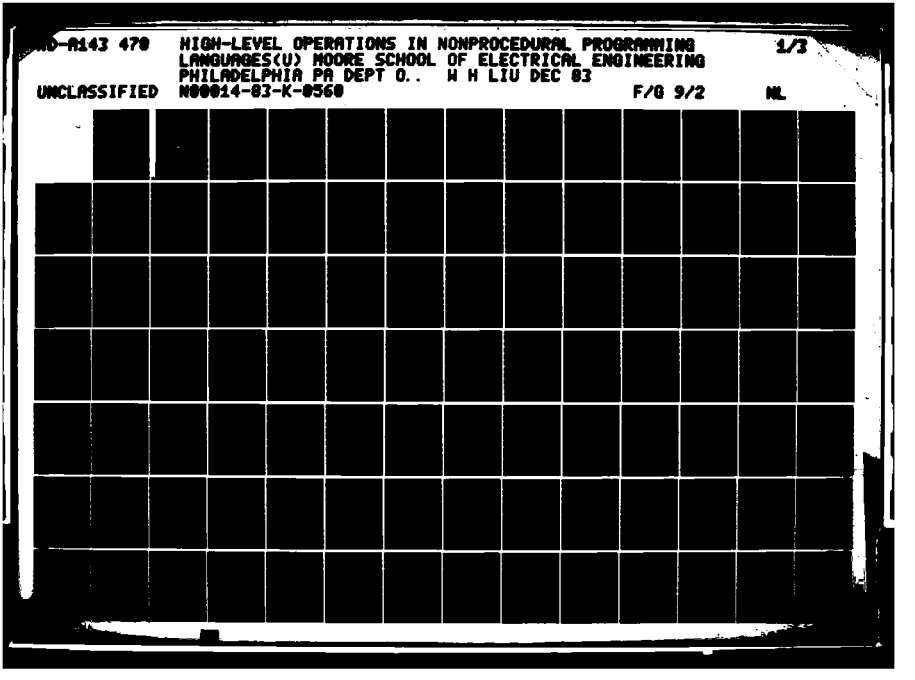
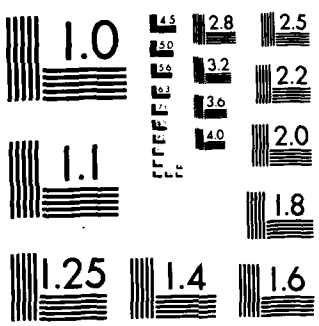

AD-A143 470
HIGH-LEVEL OPERATIONS IN NONPROCEDURAL PROGRAMMING LANGUAGES(U) MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA PA DEPT O.. W H LIU DEC 83
1/3
UNCLASSIFIED
N00014-83-K-0560
F/G 9/2
NL




MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD-A143 470

Technical Report

# HIGH-LEVEL OPERATIONS IN NONPROCEDURAL PROGRAMMING LANGUAGES

December 1983

by

Wu-Hung Liu


DTIC
JUL 30 1984
A

*UNIVERSITY of PENNSYLVANIA*

*The Moore School of Electrical Engineering*

PHILADELPHIA, PENNSYLVANIA 19104

DTIC FILE COPY

This document has been approved for public release and sale; its distribution is unlimited.

84 7 20 064

University of Pennsylvania
Department of Computer and Information Science
Moore School of Electrical Engineering
Philadelphia, Pennsylvania 19104

Technical Report

# HIGH-LEVEL OPERATIONS IN NONPROCEDURAL PROGRAMMING LANGUAGES

December 1983

by

Wu-Hung Liu

Submitted to
Information System Program
Office of Naval Research
Under Contract N00014-83-K-0560

Moore School Report

# HIGH-LEVEL OPERATIONS

# IN

# NONPROCEDURAL PROGRAMMING LANGUAGES

Wu-Hung Liu

A DISSERTATION

in

Computer and Information Science

Presented to the Graduate Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of the Doctor of Philosophy.

1983

Supervisor of Dissertation

Graduate Group Chairperson

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER | 2. GOVT ACCESSION NO. AD-A142470 | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle) High-Level Operations In Nonprocedural Programming Languages | | 5. TYPE OF REPORT & PERIOD COVERED Technical Report |
| | | 6. PERFORMING ORG. REPORT NUMBER Moore School Report |
| 7. AUTHOR(s) Wu-Hung Liu | | 8. CONTRACT OR GRANT NUMBER(s) N00014-83-K-0560 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Pennsylvania, Moore School of Electrical Engineering, Dept. of Computer Science Philadelphia, PA 19104 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS |
| 11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Information Systems Program, Code 437 Arlington, Virginia 22217 | | 12. REPORT DATE December 1983 |
| | | 13. NUMBER OF PAGES 243 |
| 14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) | | 15. SECURITY CLASS. (of this report) Unclassified |
| | | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Nonprocedural languages, Equational languages, High level operations, Operation on data structure, MODEL, Optimization, Program Transformation, Decomposing higi level operation, Matrix operations, Relational algebra operations, Operations on files, Tree structured data

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Despite having many advantages, nonprocedural programming languages have limited acceptance because of the inefficiencies of their implementations, especially for high-level operations. Although optimization of high-level operations has beem incorporated in some language processors, it has been applied locally, treating the high-level operands as indivisible entities and thus reducing the possibilities for global optimization.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

This dissertation presents a source-to-source transformation approach for efficient implementation of high-level operations in nonprocedural programming languages. Efficiency is achieved via decomposing the high-level operations into sets of basic ones, prior to their being translated into the underlying graph representation. Global optimization hence becomes possible through source language manipulation, scheduling program events, and code generation.

The developed transformation technique is capable of handling general structured operands such as multi-dimensional arrays of structures. Thus the operations are more powerful, as compared to those proposed previously which are limited to simpler structures.

Guidelines and tools are suggested for deriving transformation rules which are best suited for the application of efficient storage allocation schemes. To demonstrate its feasibility, the methodology has been applied to the MODEL language and automatic program generator.

The specific contributions of this research include:

a) A source-to-source transformation scheme for efficient implementation of high-level operations,

b) Operations are made more powerful by allowing more generally structured operands,

c) Special indexing patterns are identified and used for the application of efficient storage allocation schemes,

d) New building blocks for the underlying array graph are suggested, and

e) The expressive power of the MODEL language is increased substantially through the incorporation of high-level operations.

TABLE OF CONTENTS

LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

## 1.1 OBJECTIVES

The primary objective of this research has been to develop a methodology for efficient implementations of high-level operations in nonprocedural definitional programming languages. The developed methodology has been applied to the MODEL language and automatic program generator.

## 1.2 MOTIVATION

Based on the degree of abstraction, programming languages can roughly be classified into three categories: low-level, high-level, and very high-level. Low-level languages, such as assembly languages, provide abstractions at the machine instruction level, alleviating the need for the user to think about programming in terms of instruction

or data codes. High-level languages such as FORTRAN, PL/I or PASCAL, have more function and control abstractions as well as richer data types. They offer advantages such as machine-independence, suppression of irreverent details, and reduction of the scope of programming errors. Very high-level definitional languages carry the abstraction even further. The elimination of explicit sequencing control allows the programmer to specify the desired outcomes as a function of the input, free from concern with the step-by-step statements of a computation. This applies not only to operands which are elementary data but also to complex structured data entities.

Very high-level languages which have no explicit sequencing control are generally referred to as nonprocedural languages [Leav74]. The underlying concept of nonprocedural programming languages is to discard the conventional von Neumann view of sequential computation, thus allowing the expression of computations to be more natural, easier, and less error-prone. Nonprocedural languages are also more powerful because of their high-level data structures and the high-level operations.

Despite the many advantages, nonprocedural programming languages have limited acceptance because of the inefficiencies of their implementations, especially for

high-level operations. Although optimization of high-level operations has been incorporated in some language processors, it has been applied locally, treating the high-level operations as indivisible entities and thus retaining the user's global structuring of the computation. In the area of database access languages, decomposition and optimization have been used, however, only for queries and not for general computations. Abstract data type has also been suggested for high-level operations in nonprocedural languages. The approach has solved the modularity problem, while the efficiency problem remains.

MODEL is a general purpose nonprocedural programming language. The MODEL processor accepts very high level specifications and compiles them into a high-level target language (PL/I). Important features of the MODEL system include its abilities to verify the completeness and consistency of specifications and to resolve some inconsistencies and ambiguities automatically. However, the lack of high-level operations in the initial MODEL implementation made the specification of some complicated computations lengthy and inconvenient. It has been necessary to increase the expressive power of the language by providing high-level operations. Since efficiency is an important issue in nonprocedural programming languages, efficient implementation has been a major concern in the

incorporation of these operations. It has also been necessary to retain in the extension the powerful analytical methods incorporated in the processor to perform checking and global optimization.

## 1.3 HIGH-LEVEL OPERATIONS PROVIDED

This section briefly describes the operations that were selected for the implementation in MODEL. Generally, all the operations operate on tree structured operands. The selected tree (or subtree) data structure is assigned a name.

The implemented operations are:

1. Simple assignments and operations:
   Assignments such as A=B and the operations +, -, *, and /, for example A=B+C. The latter means that respective terminal nodes of B and C are added to define the respective terminal nodes of A. In addition, the conditional operation A=IF cond THEN B ELSE C ... may be used.

2. Matrix operations:
   Transposition, multiplication, and inversion.
   A constant UNIT may also be used as operand.

3. Relational operations

UNIQUE: Eliminate duplicated elements from a one-dimensional array.

UNION: Form a one-dimensional array with elements from two one-dimensional arrays.

DIFF: Eliminate those elements from a one-dimensional array which are in another one-dimensional array.

PRODUCT: Form a one-dimensional array which is the Cartesian product of two one-dimensional arrays.

4. Miscellaneous data manipulations

SELECT: Select elements from a one-dimensional array to form another one-dimensional array.

MERGE: Merge two one-dimensional arrays to form another one-dimensional array.

SORT: Sort elements of a one-dimensional array.

COLLECT: Convert a one-dimensional array to a two-dimensional array.

FUSE: Convert a two-dimensional array to a one-dimensional array.

CONCAT: Concatenate two one-dimensional arrays to form a one-dimensional array.

These operations are selected based on a survey and comparison of high-level operations provided in various languages. A general improvement provided here over the surveyed operations is the extension to allow more general tree structures as operands. A flexible selection scheme allows any node in a structure to be operated on. The

operations provided in MODEL are therefore not limited to only certain kind of fixed structures.

Several operations are selected based on the experience of APL. APL has illustrated that some array operations are useful in rearranging or selecting array elements. SELECT, MERGE, COLLECT, SORT and FUSE are the important ones among the array operations. They are provided in MODEL to handle arrays of general structures.

The matrix operations are taken from the matrix algebra. They are provided because of the frequent needs to write equations and expressions involving vectors and matrices. Besides matrix multiplication, inversion and transposition, basic operations are also extended piecewisely to accept vectors and matrices as operands.

The Relational Algebra has been used as a standard of comparison for the theoretic expressive power of relational database access languages. In order to make MODEL at least as powerful as the Relational Algebra, or relational complete, four more operations - UNIQUE, UNION, DIFF and PRODUCT are added.

The provided high-level operations described above constitute a basic set of frequently used ones. Their power and the flexibility in using them essentially eliminate the

need of the user to define additional operations.

## 1.4 CONTRIBUTIONS

This research has accomplished the development of a source-to-source transformation methodology for efficient implementation of high-level operations in nonprocedural languages. The methodology has been applied to the MODEL language in providing matrix operations and general data structure manipulation functions. Efficiency is achieved via decomposition of high-level operations into elemental ones, thus allowing general global optimization at various level to be applied. More specifically, the accomplishments include:

a) A new source-to-source transformation scheme which enables automatic selection of data representations and allows the applications of global optimization at the source, the scheduling, and the code generation levels.

b) The structured operands are not restricted to rectangular arrays of scalars (as in APL) or flat tables (as in relational database languages). They can be multi-dimensional arrays of arbitrary hierarchical structures. This makes the operations more powerful.

c) To obtain efficiency the transformation especially use

subscript expressions of selected forms and secondary indirect indexing arrays. In particular, sawtooth arrays and sublinear arrays are introduced.

d) The efficient implementation of high-level operations required a new type of building blocks in the array graph used to represent the specification. Integral operations are suggested to facilitate the decomposition of high-level operations which are procedural in nature.

e) A complement of high-level operations has been selected for incorporation in the MODEL language. This increases the expressive power of the language and makes the specification of computations easier. The data-flow analysis philosophy and the verification power of the MODEL processor have been preserved.

## 1.5 APPLICABILITY OF THE METHODOLOGY

The question of how widely is the methodology usable can be answered by examining the elements upon which the methodology relies. Basically, the methodology is based on:

a) Source-to-source transformation,

b) Data flow graph, and

c) Global optimization on memory usage.

The first element, source-to-source transformation, does not

impose any restriction on a language processor, since it can always be realized as an independent processing phase. The other two elements, data flow graph and global optimization on memory usage, do depend on the language processor.

For nonprocedural programming language processors, the methodology is directly applicable, since the data dependency relationships are readily available from the source specification. It is equally applicable to other language processors provided they internally use data flow representations.

As an example of illustrating general applicability of the methodology, consider a procedural language processor. Since the source program specifies the execution sequence rather than data dependencs, data flow graphs showing data dependency between variables have to be constructed before the methodology can be applied. The processing therefore contains:

a) analyze the program,

b) construct data flow graph,

c) optimize use of memory, and

d) generate code.

## 1.6 ORGANIZATION OF THE DISSERTATION

This dissertation consists of nine chapters.

Chapter 1 is the introduction to the dissertation. It gives the objectives and motivation of the research, and summarizes what have been accomplished.

Chapter 2 surveys the related work in the area of nonprocedural programming languages and high-level operations. Various efforts in providing efficient implementations of high-level operations are examined.

Chapter 3 presents the extended MODEL language. It briefly introduces the basic MODEL language, and then describes in detail the high-level operations provided, including their syntax, the checking performed, and some examples of using them.

Chapter 4 discusses the main issue in nonprocedural language implementation - efficiency. Justification of achieving efficiency via source-to-source transformation is given.

Chapter 5 gives an overview of the extended MODEL processor. It describes various phases of the processor, from syntax analysis to code generation. The internal data flow model employed by the processor is described.

Chapter 6 describes in detail how the source-to-source transformation is carried out in the MODEL processor. The transformation process involves mainly two phases - the preprocessor which extracts data structure information, and the syntax analyzer which invokes the transformation procedures.

Chapter 7 analyzes the source-to-source transformation problem and gives the transformation rules for every high-level operations provided in the MODEL language. Techniques used in deriving the transformation rules are given.

Chapter 8 describes the analysis and checking performed for achieving higher efficiency.

Chapter 9 summarizes the work and suggests some further researches.

The syntax of the extended MODEL language (in EBNF/WSC - Extended BNF with Subroutine Calls) is included as Appendix A. Complete examples of the transformations are given in Appendix B.

# CHAPTER 2

# SURVEY OF RELATED WORK

The idea of operating on high-level structured data objects as a whole was initially introduced in the programming language APL [Iver62] in the early 60's. Operations on vectors and matrices were incorporated into the language to allow less procedural specification of computations. The extensive use of APL has also demonstrated the flexibility of array structures for various types of problems. In Section 2.1, the APL language is examined. Various proposals for extending the APL data structures are surveyed.

Section 2.2 reviews the programming language LISP because of its functional properties and its need for efficient handling of lists. It faces some similar problems as far as selecting efficient data structures is concerned.

The use of abstract data type to implement high-level operations is reviewed in Section 2.3. Although the abstract data type approach has mostly been proposed for procedural high-level languages, it may readily be used in nonprocedural ones [Sang80].

Section 2.4 reviews the very high-level language for set manipulations, SETL. The selection of efficient data structures in SETL and its storage optimization techniques are examined.

Section 2.5 reviews some database access languages. The way queries are decomposed for optimization is similar to the decomposition technique suggested in this research. They both are ways of reducing the complexities of operations for easier analysis.

## 2.1 APL

APL has a number of distinguishing characteristics; among them are high-level operations on entire data structures, very simple sequence control, and very simple syntax and semantics. It recognizes only two types of scalar data - numeric and character, and one type of structured data - rectangular array of homogeneous scalars. There are no explicit variable declarations. The recognition of array structure allows functions and

operations to be applied to elements in the whole arrays. This implicit application eliminates the need for a programmer to write loops or save intermediate results as required in other languages.

However, the simplicity of APL's data structures also limits the usefulness of the language for some applications. The limitation that arrays must be rectangular and homogeneous makes the representation of nonhomogeneous set of data difficult. This inconvenience has prompted continuous interest in generalizing the APL data structures. There have been a number of proposals on how to incorporate non-uniform data structures into APL. The approaches can be basically divided into three categories. All of them deal with generalized arrays, or tree-like structures.

The first category is to generalize the definition of an array to allow nested arrays. A nested array is a recursive data structure where the elements of an array may be arrays themselves. The nested view of array suggests that the structure and selection functions of current APL may be extended without modification. However, new primitive functions are still needed to manipulate the nesting level of arrays, and new operators to assist the item-wise operations [Gull79] [Ghan73].

The second category of approaches is simply to add a new data structure to APL, separate from the original array structures. Recursively defined trees have been suggested as a new data type for this approach [Alfo76] [Vass73]. Because the newly introduced data structure is separate from the existing arrays, a completely new set of functions has to be added to manipulate the new data structures. This complicates the language. The approach is considered too radical.

The third category of approaches extends the current APL arrays to a less general multi-dimensional ragged arrays, rather than recursively nested ones. With the introduction of carrier array [Lown81], the data-driven semantics of the primitive functions and operators is preserved. The rank structure of the array, which permits the implicit iterations, is maintained. But some primitive functions are given extended definitions in order to apply to both scalar and non-scalar data objects uniformly.

Besides the limitations of its data structure discussed above, APL has storage allocation problem when large amount of data are to be manipulated. This is due to the limited amount of storage space allocated for the work space associated with each APL session. Even with the incorporation of file handling functions to create, remove

or update components of files, the user still can not view a file as a whole and apply high-level operations in single steps, as it is done with structures residing in the work space. This in a sense forces the user to return to the 'word at a time' looping and procedural processing style of lower level languages.

The techniques proposed in this research allow the incorporation of high-level operations with the preservation of high-level data abstraction. As will be described in the following chapters, the user still can view a structure, including files, as a whole unit and apply high-level operations on it with single assertions. The MODEL processor automatically employs suitable storage allocation schemes to avoid the allocation of main storage for the entire structure.

## 2.2 LISP

LISP is a well known functional language. LISP data can be either atoms or lists. Atoms are numbers or strings of characters. Elements in a list can be either atoms or other lists. Basic operations in LISP include:

CAR(s) - returning the first element of the list s.

CDR(s) - returning the list that remains when the first element of s is deleted.

CONS(a s) - returning the list that results from prefixing the atom a to the list s.

In spite of semantic elegance, excessive runtime storage overhead is required because of the repeated copying of list structures implied by the nature of the operations. Most LISP implementations allow explicit store operations such as RPLACA, shifting the responsibility of storage management to the user. This destroys the functional property of the language.

A technique of improving LISP's efficiency is using lazy evaluation [Hend76]. This can be illustrated by the evaluation of

CAR(CONS(x y))

The evaluation of list y can be avoided by delaying it until it is needed, which may never happened. The result of the above evaluation is always equal to the result of evaluating x alone. The general problem with this method is how to determine when an evaluation step is really necessary. Propagation schemes must be used to pass the 'need to be evaluated' property.

Another technique for efficient list handling is by using the I-structures [Arvi80]. An I-structure is a list with monotonic creation and consumption. This property allows a language processor to acquire storage only when an element is to be produced, or to release it as soon as the element is consumed, thereby to retain only as many elements in main storage as needed, instead of the whole structure.

It is interesting to note that the properties of the I-structures are recognized in MODEL automatically. Virtual or window storage allocation schemes (described in the next chapter) are used in the MODEL processor for those variables whose values are defined and used in a monotonic fashion.

## 2.3 ABSTRACT DATA TYPES

Like structured programming, abstract data type emphasizes locality of related collection of information. The properties of a data structure and its operations are specified in a separate unit of the program. The idea is to treat the data structure thus defined as a whole, operated on only by those operations defined for it. The programmer is then only aware of the data type and its operations, not their implementations. It is a way of extending the set of available types for the programmer while at the same time keeping the data referencing simple and program logic

manageable.

A number of high-level programming languages provide some facilities for supporting abstract data type. Pascal has a set of data structuring constructs that are suitable for defining data abstractions. The Package construct in ADA can be used to isolate a set of related definitions from the rest of the program. It also has generic definitions which allow many similar abstractions to be generated from a simple template.

CLU [Lisk77] and Alphard [Shaw77] are languages designed to support the use of abstract data type in program construction. They both provide a mechanism to enforce information hiding - or encapsulation - for better control over the scope of names, a type definition mechanism for separation of the specification and representation of a data type, and an operator definition mechanism for defining type specific operators.

Abstract data type has been applied to the nonprocedural language NOPAL [Sang80]. It was primarily used as a tool to achieve modularity. The specification of an abstract data type is independent of its use. The sub-unit where an abstract data type is defined could be different from the sub-unit which uses it. An abstract data type can also be recursively defined to obtain more

complicated structures such as stack of stacks. The main advantage is in decomposition of a problem. It allows operations on larger units of data. When these larger units of data correspond to some concept naturally occurring in the problem domain, the specification can be conveniently written in terms of these concepts.

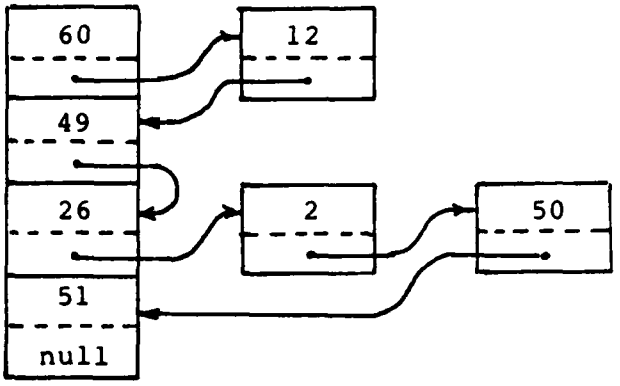
The major disadvantage of the above approach is the inefficiency in storage usage. First, in the generated program for the specification of an abstract data type, storage space is allocated for the representation of each of the variables of abstract data type. The storage space can not be shared or used again. Secondly, the module that defines the type and the one that uses it are analyzed independently, eliminating the possibilities for global optimization.

For the incorporation of high-level operations, the decomposition techniques introduced in this dissertation solves the efficiency problem by transforming each high-level function into an equivalent set of basic operations (without using additional interim variables). The decomposed function is analyzed together with other parts of the specification, thus allowing global optimization.

## 2.4 SETL

SETL is a very high-level set oriented language. The basic data structures of SETL are sets and tuples. Sets are unordered collections of objects with the constraint that a given element can not appear more than once. Tuples are actually one-dimensional vectors except that they have dynamic length. They are used to represent ordered sequences and unordered "bags" where identical elements may occur. The primitive data types include integers, real numbers, and strings. SETL provides the usual set theoretic operations (union, intersection, etc.), existential and universal quantifiers, and set formers. For execution control, SETL uses conventional control structures similar to those of PASCAL.

SETL is weakly typed and requires no variable declarations. The selection of internal representations for various data types thus becomes very important in achieving both storage and execution efficiencies. Basically, tuples are represented by arrays and sets by linked hash tables, as illustrated in Figure 2.1. While the linked hash tables provide flexibilities in adding and deleting set members, they require large overhead in storage space and involve time-consuming hashing operations.




The hash code for an integer is assumed to be its value mod 4.

Figure 2.1 Linked Hash Table Representation for the set {49,2,60,12,50,26,51}

The early approach to resolve the efficiency problem was to allow the programmer to supply the variable usage information manually to the compiler via a set of data declarations [Dewa79]. The idea is to share internal representations, mainly in the linked hash tables, based on the knowledge about all possible values each variable may assume. For example, if A is always a subset of B, then A and B can share one representation, through some pointer mechanism, as opposed to having their own copies. The data

declaration sublanguage used to control the data representation selection is a supplement to the 'pure' SETL language. A program of the pure SETL language to which data structure declarations have been added is called a supplemented program. The SETL system is designed to ensure that the function of the supplemented program is equivalent to the base-language program which it incorporates.

The technique of manual data structure selection, described above, has been improved so that the selection of appropriate data representations are automatically done during the compile-time without the help from the user [Scho81]. Instead of obtaining the information about data usage from the user, the system performs a global analysis of the way in which variables are used and related to each other.

Besides reducing the number of hash tables and their sizes, the SETL processor also employs conventional optimization techniques such as moving of invariant code out of a loop, removal of unnecessary copying operations [Fred83], and common subexpression elimination [Fong77].

## 2.5 DATABASE ACCESS LANGUAGES

The information represented in a database is made accessible to its users by database access languages. Since a database system is set up based on certain data model, the operations provided by a database access language depend very much on the nature of the data structures allowed in the model.

For network and hierarchical data models, the database access languages are usually coupled to the database systems either by defining subroutines that execute database requests when called (e.g., DL/I-PL/1 of IMS), or by embedding database constructs into an existing language and using a preprocessor to translate these constructs into run-time calls on the database systems (e.g., DML of DBTG). One major characteristics of the languages for these models is that the user always has to explicitly specify the navigation needed in the network or hierarch to reach the desired records. These languages are basically procedural.

For relational data models [Codd70], the user perceives the database as a collection of flat tables of tuples. The operations are set-oriented in that the retrievals and modifications are specified for all appropriate tuples in a relation (or set) simultaneously. The user does not specify the sequencing of traversals and need no position

indicators. It is a higher-level approach in terms of nonproceduralness. Languages for relational data models can be either algebraic (e.g., ALPHA), predicate calculus (e.g., QBE) or a combination of both (e.g., SEQUEL). They all provide basic set and tuple operations. These languages are usually added to the database systems as independent new languages in which database facilities are integrated into the language environment.

An important part of processing database access languages is the optimization of queries. Since the plan (or procedure) to access data is formulated by the language processor, the choice of an efficient access path becomes its primary task. The general strategy in query optimization is to transform a multivariable query into a set of simpler queries. There are two categories in this kind of transformation:

(a) Substitution. Let Q(X1,...,Xi,...,Xn) be a n-variable query, where the X's denote variables ranging over n relations. The query can be transformed into a set of (n-1) variable queries by substituting Xi with all its possible values. After the transformation, the simpler queries are of the form Q(X1,...,w,...,Xn), where w is one of the actual value Xi may have. Repeated substitution may eventually reduce the query to a set of

1-variable queries.

(b) Reduction. A query Q(X1,...,Xi,...,Xn) can be divided into two queries Q1(Xi,...,Xn) and Q2(X1,...,Xi-1,Yi) of n-i+1 and i variables respectively, where Yi is the result of Q1. In other words, an interim relation is obtained by evaluating the given query partially and use the partial result for the evaluation of the remaining simpler one. This procedure can be repeatedly applied to the reduced subqueries.

The abstract database access language proposed by Codd [Codd72], the relational algebra, has been used as a standard of comparison for the theoretic expressive power of relational database access languages. A language is relationally complete if it can formulate a manipulation that yields the same result as any relation definable in the relational algebra. There are five basic operations in the relational algebra: Union, Set Difference, Cartesian Product, Projection, and Selection. The method proposed in this research is used to provide all these operations to show the feasibility of the suggested techniques, and to make MODEL at least as powerful as the relational algebra.

# CHAPTER 3

# THE EXTENDED MODEL LANGUAGE

MODEL is a nonprocedural language for specifying computations. The language provides facilities for specifying what the data objects are and their inter-relating equations, rather than how to compute them. For example, there are no explicit loops or I/O controls. Equations in MODEL have the same meaning as in mathematics. There are no side effects. The order in which the statements appear in the specification is irrelevant.

A specification in MODEL is accepted by the MODEL language processor. It is checked for completeness and consistency, and a program in the target language (currently PL/I) is generated. The detailed description of the language can be found in [LuKS82] and [Schw83]. Here in this chapter, Section 3.1 briefly reviews the elementary operations in the language. Section 3.2 describes the

extensions for high-level structure operations which are of concern here.

## 3.1 ELEMENTARY OPERATIONS IN MODEL

There are three kinds of statements in MODEL: the header, the data description, and the assertions. The header statements name the generated program and the external files. Data description statements describe the structures and attributes of the variables. Assertions are equations which define some variables in terms of others. A collection of statements which can be processed as a unit is called a specification.

### 3.1.1 DATA STRUCTURES

Data objects in the MODEL language can be hierarchically structured. They can be represented by trees. The nodes in the tree represent variables. A group of nodes with a common parent can be referenced by using the name of the parent node. The name of the root is the name of the entire tree structure. Terminal nodes of the tree represent unstructured elemental data items such as numbers or character strings. The nodes are labelled with variable names. The repetition of a variable is indicated by appending to it the number of repetitions in parentheses, or

alternatively by use of an asterisk if the number of repetition is not fixed.

Figure 3.1 is an example of a MODEL data structure. The tree shows that the structure DEPT consists of a field called DEPTNO, an unspecified number of EMPLOYEEs, and ten PROJECTs. An EMPLOYEE consists of two fields, EMPNO and NAME. A PROJECT consists of a PJNO and an unspecified number of EQUIPs, which in turn consists of the ITEMNO and DESC fields.

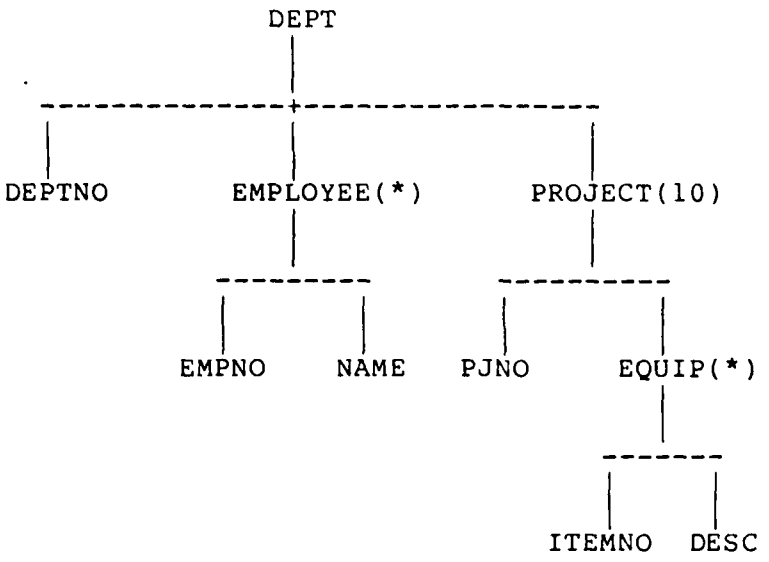



Figure 3.1 An Example of a Data Definition Tree

A repeating node variable may be viewed as an array. If there are n repeating nodes along the path from the root node to a node, then the node variable represents an n-dimensional array. The range of a dimension (number of repetitions) may vary depending on the indices of higher level dimensions. Hence the shape of the array may be jag-edged. A n-dimensional jag-edged array can be viewed as an n-level nested list. A two-dimensional array with

constant ranges m and n is an m by n matrix. A one-dimensional array with range n is also referred to as a vector. Arrays in MODEL can also be viewed as sets, if all elements in an array have distinct values.

Data structures are declared using data description statements. They are essentially the linearized form of the respective data definition tree, plus the type specifications for the terminal variables. The variables at the lowest level of the tree (or terminal nodes) are denoted as fields. A field must have a primitive data type attribute such as a number or a character string. Variables at nonterminal nodes may be either records or groups, where a record is unit of transfer of information from or to an external device. Root variables may be denoted as files. Variables denoting fields are called field variables, and those denoting records, groups, or files are called structured variables. The data description statement for the data structure of Figure 3.1 is:

```
1 DEPT IS FILE,
  2 DEPTNO IS FIELD(NUM(4)),
  2 EMPLOYEE(*) IS GROUP,
    3 EMPNO IS FIELD(NUM(6)),
    3 NAME IS FIELD(CHAR(30)),
  2 PROJECT(10) IS GROUP,
    3 PJNO IS FIELD(NUM(5)),
    3 EQUIP(*) IS GROUP,
      4 ITEMNO IS FIELD(NUM(10)),
      4 DESC IS FIELD(CHAR(30));
```

### 3.1.2 ASSERTIONS

Assertions are essentially equations. The basic format of an assertion is

```
<variable> = <expression>;
```

The variable on the left hand side of the equal sign, called the target variable, is defined by the expression on the right hand side. Variables referenced in the expression are called source variables. A more general form of the assertion is:

```
<variable> = IF <condition>
             THEN <expression>
             ELSE <expression>;
```

The following are examples of assertions:

```
A = B + 5;
C(3) = IF A<0 THEN X(1) ELSE Y(2);
```

A special kind of variables, called subscript variables, are used to denote indices of referenced structures. A subscript variable assumes all the values of positive integers from 1 through the respective dimension range of the array. For example, let I and J be subscript variables, then

```
A(I,J) = IF I=J THEN 1 ELSE 0;
```

defines a unit matrix.

### 3.1.3 AN EXAMPLE OF A MODEL SPECIFICATION

Figure 3.2 is a complete example of a MODEL specification. It specifies a task of separating a sequence of sorted records into groups, where all records in a group share a common property, in this case the same account number. The specification views the input and the output files as one-dimensional and two-dimensional arrays of records respectively, and expresses the correspondence between the input and output records in terms of the indices of respective records. Two interim indexing arrays are used for the two dimensions of the output records.

The first three lines of the specification in Figure 3.2 constitute the header statements. They specify the name of the specification - GROUPING, the source file - F1, and the target file - F2. Lines 5 to 8 describe the structure of the source file F1, consisting of unspecified number of records (F1R(*)), with two fields in each record F1R, an account number - ACCT and a description of the account - DESC. The statement in lines 10 to 14 declares the structure of the target file F2, which consists of groups - F2G. Each group has record structures - F2R, which in turn contain two fields with the same names as those in the input

records - ACCT and DESC. Ambiguities in referencing are avoided by prefixing field names with the corresponding file names, such as F2.ACCT AND F1.ACCT.

The subscript I is used to index the one-dimensional input records F1R. It is declared in line 21. X and Y are declared in lines 18 and 19. They are declared without any information on their dimensionalities. This information is resolved by the MODEL processor automatically. They denote the indices of elements of the two-dimensional output records. The assertion in lines 23 to 27 defines the values of the elements of array X in terms of I. It indicates that when an input record account number changes, i.e., F1.ACCT(I)^=F1.ACCT(I-1), a new group is initiated and the group index is incremented by 1, otherwise the the group index remains the same. The assertion in lines 29 to 33 defines the elements of array Y also in terms of I. It indicates that in each group the indices start from 1 and are incremented by 1.

The assertion in line 16 defines the number of input records. The assertion in line 38 defines the number of output groups, as equal to the value of the last element in X, or X(SIZE.F1R). The assertion in lines 39 and 40 define sizes of output groups, as equal to the values of the last element of the respective group in Y, i.e., when

F1.ACCT(I)^=NEXT.F1.ACCT(I).

Finally the assertions in lines 35 and 36 define the output fields in terms of appropriate input fields, indexed by the two arrays X and Y.

```
 1      MODULE: GROUPING;
 2      SOURCE: F1;
 3      TARGET: F2;
 4
 5      1 F1 IS FILE,
 6        2 F1R(*) IS RECORD,
 7          3 ACCT IS FIELD(NUM(10)),
 8          3 DESC IS FIELD(CHAR(30));
 9
10      1 F2 IS FILE,
11        2 F2G(*) IS GROUP,
12          3 F2R(*) IS RECORD,
13            4 ACCT IS FIELD(NUM(10))
14            4 DESC IS FIELD(CHAR(30);
15
16 a1: SIZE.F1R = 123;
17
18      X IS FIELD(NUM(5));
19      Y IS FIELD(NUM(5));
20
21      I IS SUBSCRIPT;
22
23 a2: X(I) = IF I=1
24                 THEN 1
25                 ELSE IF F1.ACCT(I)^=F1.ACCT(I-1)
26                      THEN X(I-1)+1
27                      ELSE X(I-1);
28
29 a3: Y(I) = IF I=1
30                 THEN 1
31                 ELSE IF F1.ACCT(I)^=F1.ACCT(I-1)
32                      THEN 1
33                      ELSE Y(I-1)+1;
34
35 a4: F2.ACCT(X(I),Y(I)) = F1.ACCT(I);
36 a5: F2.DESC(X(I),Y(I)) = F1.DESC(I);
37
38 a6: SIZE.F2G = X(SIZE.F1R);
39 a7: SIZE.F2R(X(I)) = IF I=SIZE.F1R |
40                          F1.ACCT(I)^=NEXT.F1.ACCT(I)
41                      THEN Y(I);
```

Figure 3.2 An Example of a MODEL Specification

## 3.2 HIGH-LEVEL OPERATION EXTENSIONS

This section describes the extensions made to the MODEL system for supporting use of high-level operations. Some conventions and notations in referencing high-level structured data are described first in Section 3.2.1. Sections 3.2.2 and 3.2.3 describe the high-level operations in two categories - matrix operations and array manipulation functions. Section 3.2.4 gives an example of use of high-level operations.

### 3.2.1 DEFINITIONS AND SYNTAX

This section summarizes referencing structured data and the requirement of structured variable compatibility.

#### 3.2.1.1 Referencing Structured Data

When using high-level operations, the operands can be referenced in two ways: on a structure aggregate level - which is generally simpler, and on a structure instance level - which is more flexible.

A reference at the structure aggregate level refers to the collection of all instances of the fields in a structure. The syntax of such reference can either be a structured variable name or a field variable name

subscripted with asterisks. For example, consider the data structure:

```
1 A IS GROUP,
  2 AG(3) IS GROUP,
    3 AH(2) IS GROUP,
      4 AF(2) IS FIELD;
```

which define a three-dimensional array of field AF. The collection of all the 3x2x2 elements can be referenced with the structured variable name - A, or with the field variable name subscripted by asterisks - AF(*,*,*). Asterisks and other subscript variables may be mixed in referencing substructures. For example, AG(I) and AF(3,*,*) refer to the I-th and the third 2x2 matrix respectively. As an example of referencing structured operand in assertions on the aggregate level, assuming two more matrix, B and C, defined as:

```
1 B IS GROUP,
  2 BG(2) IS GROUP,
    3 BF(2) IS FIELD;

1 C IS GROUP,
  2 CG(2) IS GROUP,
    3 CF(2) IS FIELD;
```

then either the following assertions (the operators |^, |/ and |* denote matrix transposition, inversion and multiplication respectively):

```
AG(1) = |^ B;
AG(2) = |/ B;
AG(3) = B |* C;
```

or

```
AF(1,*,*) = |^ B;
AF(2,*,*) = |/ B;
AF(3,*,*) = B |* C;
```

define the three matrix AG(1), AG(2) and AG(3) as the transposition of B, the inverse of B, and the product of B and C respectively.

For structures with more than one field descendent, the only way to reference substructures containing only one field is by subscripting with asterisks. For example, if the data structure of B is:

```
1 B IS GROUP,
  2 BG(2) IS GROUP,
    3 BF(2) IS FIELD,
  2 BE(5) IS FIELD;
```

The 2x2 matrix of field BF in this structure can only be referenced by BF(*,*), since B in this case refers to the combination of the array BF(*,*) and the vector BE(*).

References of structured operands on the instance level are used when the indices of the elements in the resulting array needs to be explicitly expressed in terms of the indices of the elements in the defining array. For example, selecting elements (using the function SELECT, to be

described subsequently) from array B to form array A can be written as

```
A(L) = SELECT(B(I),cond(I,L));
```

meaning that the I-th elements of array B is selected to become the L-th element in A if the condition 'cond(I,L)' is true. This kind of array references are allowed in order to make the array operations more powerful.

3.2.1.2 Structured Variable Compatibility

When defining a structure variable from another structured variable, a correspondence implied between respective fields in the two structures. It is therefore necessary that, corresponding to each field in the defined structure, there is a field in the referenced structure with the same dimensionality and the same data type. Namely the structures of the dependent and the independent variable must match. Let T and S be the dependent and independent variables respectively, the compatibility can be determined in two ways:

by name: for every field t in the structure T, there is a field s in the structure S such that t and s share the same name, same data type and same dimensionality.

by structure: for every field t in the structure T, there is

a field s in the structure S such that t and s share the same sibling position, same dimensionality, and same data type.

Note that compatibility is determined either by name or by structure. The conditions for them can not be mixed. The user has the option of selecting them when using the high-level functions. By default, by-name has a higher priority than that of by-structure.

### 3.2.2 MATRIX OPERATIONS

Matrix operations are denoted by special matrix operators. The operators may be unary (for inversion or transposition) or binary (for multiplication). The following symbols denote the matrix operations:

|* matrix multiplication
|/ matrix inversion
|^ matrix transposition

The operands for these operations are all two-dimensional arrays. However, a one-dimensional array with range n can be treated as a 1 by n matrix. The shapes of the operands referenced in a matrix operation must conform with the shape requirements of that operation. The language processor will

issue error messages if the shapes do not match.

The precedences of the unary matrix operators (|/ and |^) are higher than that of the binary operator (|*), which is in turns higher than those of the basic operators (*, /, + and -).

In the following sections, matrices are denoted by a single capital letter for clarity. In fact, each of them can be a collection of instances along any two dimensions in a multi-dimensional arrays such as

```
AF(...,*,...,*,...)
```

where AF is a multi-dimensional field variables.

### 3.2.2.1 Matrix Transposition

The unary transposition operator |^ is used to reverse the roles of rows and columns of a matrix. The assertion

```
B = |^ A
```

defines B as a matrix whose element at row i and column j is equal to the element at row j and column i of matrix A. If the shape of A is m by n, B must be n by m. If the shape of B is not specified, it will be defined by the processor automatically through propagation.

A |* |/ B

### 3.2.3 ARRAY MANIPULATION FUNCTIONS

The section describes the use of the following array manipulation functions:

1. SELECT: Select elements from a one-dimensional array to form another one-dimensional array.
2. MERGE: Merge two one-dimensional arrays to form another one-dimensional array.
3. SORT: Sort elements of a one-dimensional array.
4. COLLECT: Convert a one-dimensional array to a two-dimensional array.
5. FUSE: Convert a two-dimensional array to a one-dimensional array.
6. CONCAT: Concatenate two one-dimensional arrays to form a one-dimensional array.
7. UNIQUE: Eliminate duplicated elements from a one-dimensional array.
8. UNION: Form a one-dimensional array with elements from two one-dimensional arrays.
9. DIFF: Eliminate those elements from a one-dimensional array which are in another one-dimensional array.
10. PRODUCT: Form a one-dimensional array which is the Cartesian product of two one-dimensional arrays.

The arrays referenced in the above functions are arrays of structures, not restricted to array of scalars. The structure of the array elements can be a branch in the data

structure tree declared in the user specification. The branch is specified by the user with the variable name associated with the root of the branch.

Consider the following data structure:

```
1 A IS FILE,
  2 A1(10) IS GROUP,
    3 A2(20) IS GROUP,
      4 A3(30) IS GROUP,
        5 AF1 IS FIELD,
        5 AF2(7) IS FIELD;

1 B IS FILE,
  2 B1(10) IS GROUP,
    3 B2(20) IS GROUP,
      4 B3(30) IS GROUP,
        5 BF1 IS FIELD,
        5 BF2(7) IS FIELD;

1 C IS FILE,
  2 C1(*) IS GROUP,
    3 C2(*) IS GROUP,
      4 C3(30) IS GROUP,
        5 CF1 IS FIELD,
        5 CF2(7) IS FIELD;
```

and the function CONCAT, whose two arguments are one-dimensional arrays of structures. The assertion

```
C1(*) = CONCAT(A1(*),B1(*));
```

concatenates an array of 10 A1 substructures and an array of 10 B1 substructures to form an array of 20 C1 substructures. The repetition of the C1 substructure, 20, is automatically defined by the language processor. While the following assertion

```
C2(I,*) = CONCAT(A2(I,*),B2(I,*));
```

concatenates the I-th (of 10) row of 20 A2 substructures and the I-th (of 10) row of 20 B2 substructures to form the I-th (of 10) row of 40 C2 substructures. Note that substructure C1 is compatible to substructure A1 and to substructure B1, and C2 is compatible to A2 and to B2.

Furthermore, the MODEL processor allows the omission of subscripts on the left end of the parentheses (the more significant ones) if they are the same and are use at the same positions. Thus the last assertion can also be written as

```
C2(*) = CONCAT(A2(*),B2(*));
```

3.2.3.1 The SELECT Function

SELECT defines an array of structures by selecting elements from another array of compatible structures. An assertion using the SELECT function has the following format:

```
A(L) = SELECT(B(I),cond(I,L));
```

The assertion states that if cond(I,L) is true, then the L-th element of A is equal to the I-th element of B. When the condition does not depend on L, it can be stated as

```
A(*) = SELECT(B(I),cond(I));
```

Consider the following example involving the files:

```
1 F IS FILE,
  2 G(*) IS RECORD,
    3 K IS FIELD (CHAR(4)),
    3 X IS FIELD (NUM(4));

1 E IS FILE,
  2 H(*) IS RECORD,
    3 KEY IS FIELD (CHAR(4)),
    3 Y IS FIELD (NUM(4));
```

The SELECT function is used to define E as containing only those records in F whose X field is positive:

```
H(*) = SELECT(G(I),X(I)>0);
```

A more powerful application of this function is to use more complex conditional expression involving the subscript of H. For example, the following assertion selects only the first of every subsequence of G which have the same value in field K:

```
H(L) = SELECT(G(I),KEY(L-1)^=K(I));
```

The resulting size is automatically defined by the SELECT function.

3.2.3.2 The MERGE Function

Given two arrays of structures, MERGE defines a new array by interleaving the elements of the two source arrays. An assertion using the MERGE function has the following format:

```
A(L) = MERGE(B(I),C(J),cond(I,J,L));
```

The assertion states that the L-th element of array A is defined as the I-th element of B if cond(I,J,L) is true, otherwise it is defined as the J-th element of C.

Consider the data structures:

```
1 F IS GROUP,
  2 P(*) IS FIELD(NUM);

1 G IS GROUP,
  2 Q(*) IS FIELD(NUM);

1 H IS GROUP,
  2 R(*) IS FIELD(NUM);
```

Assume that G and H are sorted in ascending order, then the assertion

```
P(*) = MERGE(Q(I),R(J),Q(I)<=R(J));
```

defines the array P, remaining sorted in ascending order, by merging elements in arrays Q and R.

3.2.3.3 The SORT Function

The SORT function defines a one-dimensional array of structures by sorting the elements of another one-dimensional array of structures. The format of using the SORT function is:

```
A = SORT(B,key,order);
```

where 'key' is a field variable in B, and 'order' is the key word ASC (for ascending) or DSC (for descending). The assertion states that all the elements of the array B are sorted according to 'key' in the order 'order'.

As an example, consider the following data:

```
1 P IS GROUP,
  2 Q(*) IS RECORD,
    3 R IS FIELD,
    3 S IS FIELD;
```

```
1 X IS GROUP,
  2 Y(*) IS RECORD,
    3 U IS FIELD,
    3 V IS FIELD;
```

The assertion

```
P = SORT(X,U,ASC);
```

sorts the one-dimensional array Y(*) of records, according the field U in ascending order, resulting in the one-dimensional array Q(*).

3.2.3.4 The COLLECT Function

The COLLECT function converts a one-dimensional array of structures into a two-dimensional jagged-edge array of structures. The format of using this function is:

```
A(I,J) = COLLECT(B(K),cond(I,J,K));
```

The one-dimensional array B(*) is divided into array of arrays A(*,*) in such a way that A(1,1) is defined as B(1), A(1,2) as B(2), etc. And, when cond(I,J,K) is true, J is the range of array A(I,*).

As an example, let B(*) be a one-dimensional array containing the integers 1 through 15.

```
{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}
```

The assertion

```
A(I,J) = COLLECT(B(K),I=J & K<10);
```

defines a two-dimensional array A(*,*):

```
Row 1:  {1}
Row 2:  {2,3}
Row 3:  {4,5,6}
Row 4:  {7,8,9,10,11,12,13,14,15}
```

3.2.3.5 The FUSE Function

The FUSE function is the inverse of COLLECT. It defines a one-dimensional array from a two-dimensional jagged-edge array. It has the following format:

```
A(*) = FUSE(B(*,*));
```

There is no conditional expression involved in the definition of A(*). It is simply the concatenation of all rows in B(*,*).

3.2.3.6 The CONCAT Function

The CONCAT function concatenates two one-dimensional arrays of compatible structures. The format of using this function is:

```
A=CONCAT(B,C);
```

The concatenation is done in such a way that the first element of array C follows the last element of array B.

Consider the following data:

```
1 A IS GROUP,
  2 AF(*) IS FIELD;

1 B IS GROUP,
  2 BF(*) IS FIELD;

1 C IS GROUP,
  2 CF(*) IS FIELD;
```

The assertion A=CONCAT(B,C) defines A as an array of size SIZE.BF+SIZE.CF, whose elements are obtained by appending array C to array B. The relative orders of the elements within their original arrays are maintained.

3.2.3.7 The UNIQUE Function

The UNIQUE function eliminates duplication in a one-dimensional array. The format of using this function is:

```
A=UNIQUE(B,key);
```

where 'key' is a field in B or the keyword ALL. When a field name is used as the key, elements in B with the same value in that field are considered duplicated, disregarding contents of other fields in the structure. For example, consider the structure:

```
1 A IS GROUP,
  2 AG(*) IS GROUP,
    3 AF1 IS FIELD,
    3 AF2 IS FIELD,
    3 AF3(*) IS FIELD;

1 B IS GROUP,
  2 BG(*) IS GROUP,
    3 BF1 IS FIELD,
    3 BF2 IS FIELD,
    3 BF3(*) IS FIELD;
```

The assertion

```
A=UNIQUE(B,AF1);
```

defines A as containing elements in B, omitting those elements whose AF1 values are duplicated.

If 'ALL' is used as the key instead of a field name, fields with the same dimensionality as those of B's elements will be used in determining duplication. Therefore, assuming the same data structures defined above, the assertion

```
A=UNIQUE(B,ALL);
```

eliminates those elements of B whose combined values of AF1 and AF2 are duplicated.

3.2.3.8 The UNION Function

When two one-dimensional arrays, say A and B, are viewed as sets, their union can be obtained by using the UNION function as follows:

```
C=UNION(A,B);
```

The resulting array C contains all elements in A and B, with duplicated elements eliminated. The order of elements in C

is undefined.

There is a restriction on array structures in using UNION. The elements of the arrays must contain single fields only. Two elements in the array are considered duplicated if corresponding fields in both elements contain the same value. For example, consider the data structures:

```
1 A IS GROUP,
  2 AR(3) IS GROUP,
    3 AF1 IS FIELD(CHAR(1)),
    3 AF2 IS FIELD(CHAR(1)),
    3 AF3 IS FIELD(CHAR(1));

1 B IS GROUP,
  2 BR(2) IS GROUP,
    3 BF1 IS FIELD(CHAR(1)),
    3 BF2 IS FIELD(CHAR(1)),
    3 BF3 IS FIELD(CHAR(1));

1 C IS GROUP,
  2 CR(*) IS GROUP,
    3 CF1 IS FIELD(CHAR(1)),
    3 CF2 IS FIELD(CHAR(1)),
    3 CF3 IS FIELD(CHAR(1));
```

and assume the actual data are:

```
AR(1):  {C,A,R}
AR(2):  {C,A,T}
AR(3):  {C,A,B}

BR(1):  {C,A,N}
BR(2):  {C,A,R}
```

then C=UNION(A,B) defines C as:

```
CR(1):  {C,A,R}
CR(2):  {C,A,T}
CR(3):  {C,A,B}
CR(4):  {C,A,N}
```

3.2.3.9 The DIFF Function

When two one-dimensional arrays, A and B, are viewed as sets, their difference can be obtained by using the DIFF function as follows:

```
C=DIFF(A,B);
```

The resulting array C contains all elements which are in A but not in B. The DIFF function has the same restriction on its arguments like in using the UNION function. Both A and B must contain single fields only. As an example, assume A and B have the same structures and values as shown in the last section (the UNION function example), then the assertion C=DIFF(A,B) defines array C as:

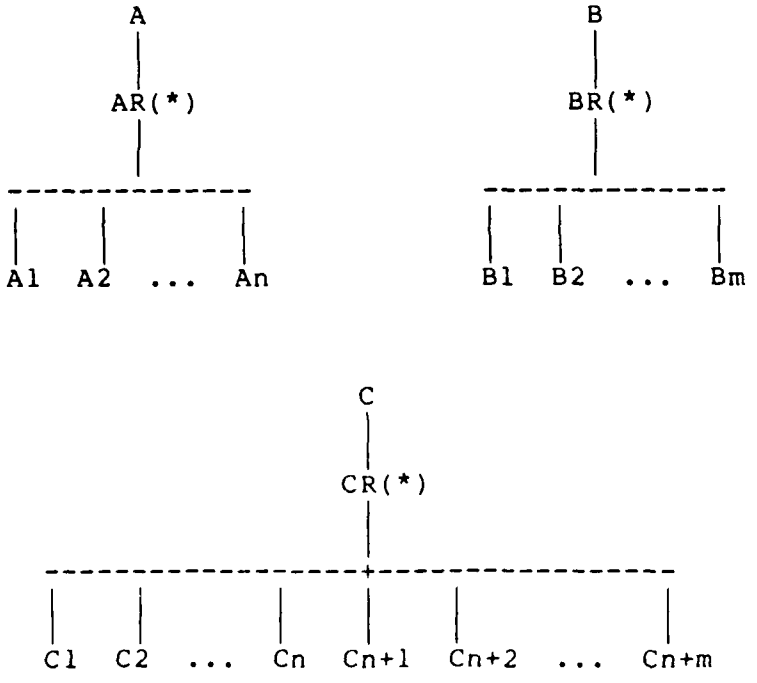
```
CR(1):  {C,A,T}
CR(2):  {C,A,B}
```

3.2.3.10 The PRODUCT Function

Let arrays A and B be one-dimensional arrays of structures containing n and m single fields respectively. The PRODUCT function defines a one-dimensional array of structures containing n+m single fields, whose first n fields are taken from A and the last m fields are from B. The format of using the PRODUCT function is:

```
C=PRODUCT(A,B);
```

The data structures of the source and target arrays are illustrated as follows:



As an example, consider the following structures of A, B and C:

```
1 A IS GROUP,
  2 AR(3) IS GROUP,
    3 AF1 IS FIELD(CHAR(3)),
    3 AF2 IS FIELD(NUM(2)),

1 B IS GROUP,
  2 BR(2) IS GROUP,
    3 BF1 IS FIELD(NUM(4));

1 C IS GROUP,
  2 CR(*) IS GROUP,
    3 CF1 IS FIELD(CHAR(3)),
    3 CF2 IS FIELD(NUM(2)),
    3 CF3 IS FIELD(NUM(4));
```

assuming the actual value of A and B are:

```
AR(1):  {JAN,15}
AR(2):  {MAY,31}
AR(3):  {DEC,1}

BR(1):  {1983}
BR(2):  {1984}
```

Then C=PRODUCT(A,B) defines C as:

```
CR(1):  {JAN,15,1983}
CR(2):  {JAN,15,1984}
CR(3):  {MAY,31,1983}
CR(4):  {MAY,31,1984}
CR(5):  {DEC,1,1983}
CR(6):  {DEC,1,1984}
```

### 3.2.4 AN EXAMPLE OF A SPECIFICATION USING HIGH-LEVEL OPERATIONS

Figure 3.3 shows an example of a MODEL specification using the high-level function SORT.

The first three lines specify names of the specification - S, the source file - SB, and the target file - SA. Lines 5 through 12 describe the source file structure. It consists of records B, which are many-leveled structures. Lines 14 through 21 describes the structure of the target file, which is compatible with the source file. The size of the source file is specified in line 23. Line 25 defines the target file SA as obtained from sorting elements of SB in ascending order (ASC). The key used in the sorting is the field B1 in input record B. By treating the whole structures as high-level data objects, only one assertion - SA=SORT(SB,B1,ASC) - is sufficient in specifying the desired sorting.

```
 1   MODULE: S;
 2   SOURCE: SB;
 3   TARGET: SA;
 4
 5   1 SB IS FILE,
 6     2 B(*) IS RECORD,
 7       3 B1 IS FIELD(CHAR(3)),
 8       3 B2(3) IS GROUP,
 9         4 B21(2) IS GROUP,
10           5 B211 IS FIELD(CHAR(5)),
11           5 B212 IS FIELD(CHAR(5)),
12         4 B22 IS FIELD(CHAR(4));
13
14   1 SA IS FILE,
15     2 A(*) IS RECORD,
16       3 A1 IS FIELD(CHAR(3)),
17       3 A2(3) IS GROUP,
18         4 A21(2) IS GROUP,
19           5 A211 IS FIELD(CHAR(5)),
20           5 A212 IS FIELD(CHAR(5)),
21         4 A22 IS FIELD(CHAR(4));
22
23   SIZE.B = 30;
24
25   SA=SORT(SB,B1,ASC);
```

Figure 3.3 An Example of a Specification using High-level Operations

# CHAPTER 4

# EFFICIENCY CONSIDERATIONS

The major problem in the use of high-level operations has been in the inefficiency of their implementations, especially when the data structures on which they operate are large files. Efficiency issues therefore have become be the major concern in incorporating high-level operations into a programming language. This chapter discusses the efficiency issues, reviews the various levels of optimization in the MODEL processor, and shows how the approach via decomposition achieves an efficient implementation.

## 4.1 THE EFFICIENCY ISSUES

From a user's point of view, high-level operations operate on structures. Both the operations and the operands are viewed as indivisible units. The user needs not be

concerned with the sequence of operations on components which comprise the data structures. In actual implementation, however, straightforward interpretation of this view is often unnecessary and undesirable, mainly because of excessive storage demand. The efficiency problem becomes serious when the variables referenced represent large amount of data. Optimization can be done in several areas as follows:

(a) At the implementation level, high-level operands should not be treated as indivisible units. Individual components within a data structure should be analyzed independently. The advantage of this is that operations on a data component can be performed as soon as its predecessor operands are available, disregarding whether those for other components are available or not. The result of this is that each component of a structure can have its optimal storage allocation scheme, without regard to whether other data component that belong to the same structure can be operated on. For parallel machines, this view is even more important as it facilitates parallel execution.

(b) The second area of potential improvement is the sharing of storage space by different instances of variables. The objective is to retain in main storage only as many

variable instances as necessary. Depending on the nature of the operations involved, the definition of a variable (generation) and its use in defining others (consumption) may be in sequence. Very often storage space for only one or just a few instances is enough to be shared by the entire array. Note that this area of optimization does not conflict with item(a) above.

(c) For implementation on sequential machines, the computations of array elements are enclosed in loops. Arrays of the same range can be put in the same loop and share the loop control. From storage efficiency's point of view, an even more important aspect is that the variables defined and only referenced in a loop do not require simultaneous storage. It is therefore desirable to enlarge loop scopes by putting more variables, even those with different but related ranges, in a loop in order to reduce storage requirements.

(d) In code generation, there is another possibility for optimization. Some unnecessary copy operations can be recognized and avoided. This results also in the sharing of the same memory space by the variable instances involving the copy operation.

The approach adapted to provide efficient high-level operations is via decomposition of those operations at the source level. References to structures in high-level operations are transformed into references to components of the structures. This allows independent analysis of individual components, as required in item(a) above. Decomposition also facilities storage reduction described in item(b). As will be discussed in the following section, the reduction in storage to represent an array depends on the subscript expressions used in referring to the respective array. High-level operations can be decomposed into a set ot basic operations with the use of special subscript expressions for achieving the storage reduction.

The decompositions of high-level operations results in a specification entirely at the elementary operations level. Therefore a uniform approach can be employed in verifying the entire user specification. Also the optimization, as described above in items b, c and d is performed uniformly at the elementary operations level.

The following sections describe the optimization techniques used, including loop scope enlargement and the use of storage allocation schemes.

## 4.2 LOOP SCOPE ENLARGEMENT

When loops in a procedural language program have the same range, they often can be combined to achieve better efficiency. For example, consider the task of incrementing the value of each element in an integer array by a constant k. Assuming the array resides in an external device, one possible way of performing the task is:

```
Do for the range of the array;
  Read one element of the array;
End;
Do for I from 1 to the range of array;
  Increment the I-th element of the array by k;
End;
Do for the range of the array;
  Write one element of array;
End;
```

Since the above three loops have the same range, they may be combined as:

```
Do for I from 1 to range of array;
  Read the I-th element of the array;
  Increment the I-th element of the array;
  Write the I-th element of the array;
End;
```

The advantage of combining loops together as shown above is the saving in storage space for the entire array.

Loops of different ranges may also be merged if the ranges are related. Consider the specification shown in Figure 4.1.

```
 1  MODULE: G;
 2  SOURCE: NFILE;
 3  TARGET: LFILE;
 4
 5  1 NFILE IS FILE,
 6    2 NR(*) IS RECORD,
 7      3 NAME IS FIELD(CHAR(30)),
 8      3 SNO IS FIELD(NUM(4));
 9
10  1 LFILE IS FILE,
11    2 LR(*) IS RECORD,
12      3 NAME IS FIELD(CHAR(30));
13
14  I IS SUBSCRIPT;
15
16  INX IS FIELD(NUM(5));
17
18  INX(I)= IF I=1
19          THEN IF SNO(I)>181
20               THEN 1
21               ELSE 0
22          ELSE IF SNO(I)>181
23               THEN INX(I-1)+1
24               ELSE INX(I-1);
25
26  LFILE.NAME(INX(I)) = IF I=1 & INX(I)=1 |
27                             INX(I)>INX(I-1)
28                       THEN NFILE.NAME(I);
```

Figure 4.1 A Specification Example for Merging Loops with Different Ranges

The specification has one source file - NFILE, and one target file - LFILE (lines 2 and 3). Lines 5 to 8 describe the structure of the source file. It contains records of two fields: a name (NAME) and its associated serial number (SNO). Lines 10 to 12 describe the structure of the target file, which is simply a list of names. The specification defines the target file as a list of names whose associated serial number is greater than 181. The idea in defining this file is to use an intermediate integer array which serves as a link between the indices to the names in the source file and the indices to the names in the target file. This indexing array is INX, as declared in line 16 and defined with the assertion in lines 18 to 24. The array value is monotonically incremented by 1 whenever the corresponding instance of SNO is greater than 181. The assertion in lines 26 to 28 uses INX to relate the instances of the NAMEs in the source and the target files.

Because the value of the array used to subscript LFILE.NAME, INX(I), increases as I does, and at a rate slower than I, the definition of LFILE.NAME can be put in the scope of I as follows:

```
Do for I from 1 to range of NR;
  Read one instance of record NR;
  Define INX(I);
  If I=1 & INX(I)=1 or INX(I)>INX(I-1)
  then
    Do;
      LFILE.NAME(INX(I))=NFILE.NAME(I);
      Write one instance of record LR;
    End;
End;
```

Loops of different ranges can be merged together depending on how the two loop indices, I and INX(I), are related. The indexing array INX must satisfy the following conditions:

a) monotonically increasing: INX(i)>=INX(j) for i>j, and

b) increases slower than the indices: INX(i)<=i for all i.

An integer array satisfying the above conditions is referred to as being <u>sublinear</u>. Syntactically, the indexing array must have the following format:

```
INX(I) = IF I=1
         THEN [1|0]
         ELSE IF any condition
              THEN [INX(I-1)+1 | INX(I-1)]
              ELSE [INX(I-1)+1 | INX(I-1)]];
```

The values of the elements of a sublinear array can be viewed as an orderly enumeration (with repetition) of the indices along the dimension it subscripts. The dimension

indexed by the subscript I is then called the major dimension having a major range, as related to the dimension indexed by INX(I), which has its subrange. A subrange relative to a major range may be the major range of some other subranges. Therefore, these sublinear relationships may form a tree with the maximal major range at the root. The scheduling process will attempt to place all variables involving the tree into one loop which iterates for all the instances of the maximal major range. The scope of the loop will also contain conditions that will check that only a single instance within the major range is evaluated for each of the variables with the subranges.

The sublinearity can be generalized for the merging of more than one ranges into another. Consider the assertion:

```
B(Z1(I),Z2(I),...,Zn(I))=A(I);
```

where B is an n-dimensional array and A is one-dimensional. The source variable A, the assertion, and the target variable B can all be scheduled in a loop of range I as follows:

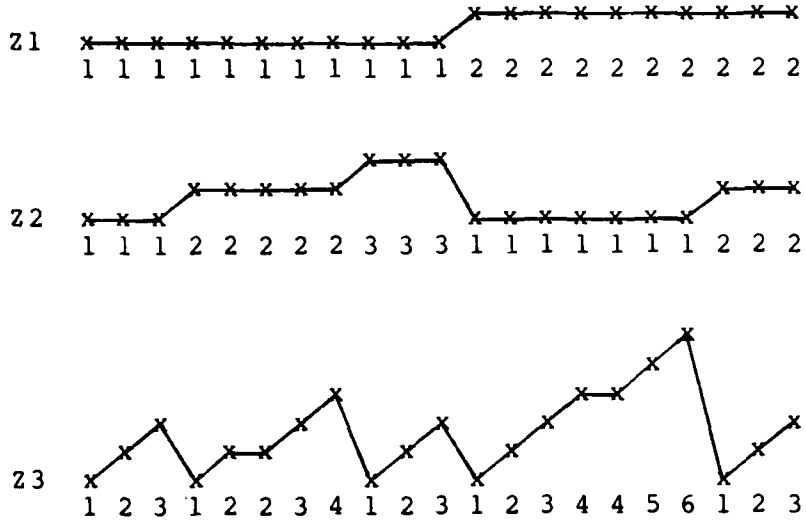



Figure 4.2 An Example of Sawtooth Index Sequence

Syntactically, except for the first array in the sawtooth array sequence, which must conform with the sublinear array syntax as stated above, the rest of them must have the following format:

```
Zk(I) = IF I=1
        THEN Zk-1(I)
        ELSE IF Zk-1(I)^=Zk-1(I-1)
             THEN 1
             ELSE [ Zk(I-1)+1 |
                    IF any_condition
                    THEN [Zk(I-1)+1 | Zk(I-1)]
                    ELSE [Zk(I-1)+1 | Zk(I-1)]];
```

## 4.3 STORAGE ALLOCATION SCHEMES IN MODEL

A program generated by the MODEL processor incorporates loops where the value of each loop control variable is stepped from one to the range of the loop. The storage allocation requirements for a variable in the generated program depend mainly on how instances of the variable are referenced.

Three storage allocation schemes are used in the MODEL system for various requirements. They are described in the following paragraphs. Specification examples are given to illustrate the circumstances in which they are used. Note that these schemes concern the storage associated with each dimension of a variable. If a variable is multi-dimensional, each dimension of the variable may have its own different storage allocation scheme, independent of those for other dimensions.

### 4.3.1 VIRTUAL STORAGE ALLOCATION SCHEME

Virtual storage allocation scheme for a variable dimension refers to the allocation of one single element space for an entire dimension. When an instance of a variable dimension is defined and referenced in a monotonically sequential manner, the dimension can be virtual. Consider the specification example in Figure 4.3:

```
 1   MODULE: D;
 2   SOURCE: PF;
 3   TARGET: DF;
 4
 5   1 PF IS FILE,
 6     2 PREC(*) IS RECORD,
 7       3 TIME IS FIELD(DEC FLOAT),
 8       3 XPOS IS FIELD(DEC FLOAT),
 9       3 YPOS IS FIELD(DEC FLOAT);
10
11   1 DF IS FILE,
12     2 DREC(*) IS RECORD,
13       3 TIME IS FIELD(DEC FLOAT),
14       3 DIST IS FIELD(DEC FLOAT);
15
16   I IS SUBSCRIPT;
17
18   DF.TIME(I)=PF.TIME(I);
19   DIST(I) =
20      SQRT(XPOS(I)*XPOS(I)+YPOS(I)*YPOS(I));
```

Figure 4.3 An Example of Virtual Storage Allocation Scheme

The specification defines a distance vs. time file (DF) from a file containing X- and Y-positions and respective times (PF). It calculates the distances of points from their X- and Y-positions. The assertion in lines 19 and 20 shows this definition, where SQRT is the square root function. In this assertion the i-th instance of DIST is obtained from the i-th instances of both XPOS and YPOS. The program needed to be generated for this specification can therefore be outlined as:

```
Do for the range of I;
  Read one instance of record PREC;
  Define one instance of DF.TIME;
  Define one instance of DIST;
  Write one instance of record DREC;
end;
```

As a result, only one storage space is needed for the entire array of each variable. In this example, the dimensions of both DIST and TIME can be assigned virtual storage allocation scheme.

### 4.3.2 WINDOW STORAGE ALLOCATION SCHEME

For cases where a variable instance does not strictly depend on the same corresponding instance of another variable, as it does in the case of Figure 4.3, the allocation of one storage space for the entire defining array instances is not enough. However, if the dependant

variable instances referenced are limited to those which precede the current instance, then it is possible to allocate a fixed number of storage spaces for the entire array instances. This type of storage allocation scheme is referred to as window, meaning that a window of fixed storage spaces can be viewed as shifting over the array instances as the computation goes along. To illustrate this situation, consider the specification in Figure 4.4.

```
 1  MODULE: V;
 2  SOURCE: PF;
 3  TARGET: VF;
 4
 5  1 PF IS FILE,
 6    2 PR(*) IS RECORD,
 7      3 TIME IS FIELD(DEC FLOAT),
 8      3 XPOS IS FIELD(DEC FLOAT),
 9      3 YPOS IS FIELD(DEC FLOAT);
10
11  1 VF IS FILE,
12    2 VR(*) IS RECORD,
13      3 TIME IS FIELD(DEC FLOAT),
14      3 VX IS FIELD(DEC FLOAT),
15      3 VY IS FIELD(DEC FLOAT);
16
17  I IS SUBSCRIPT;
18
19  DF.TIME(I)=VF.TIME(I);
20
21  VX(I) = IF I=1
22          THEN 0
23          ELSE (XPOS(I)-XPOS(I-1)) /
24                  (TIME(I)-TIME(I-1));
25
26  VY(I) = IF I=1
27          THEN 0
28          ELSE (YPOS(I)-YPOS(I-1)) /
29                  (TIME(I)-TIME(I-1));
```

Figure 4.4 An Example of Window Storage Allocation Scheme

The specification computes the velocities in the X and Y directions based on the X- and Y-positions and the corresponding time. The velocity is calculated by dividing the difference in positions by the difference in times. The source file (PF) is defined in lines 5 to 9. Its records

consists of three fields - TIME, XPOS and YPOS. The target file (VF) structure is defined in lines 11 to 15. Its records contains the fields TIME, the velocity in the X direction - VX, and the velocity in the Y direction - VY. The TIME in PF is the same as the TIME in VF (line 19). The assertion in lines 21 to 24 and the one in lines 26 to 29 defines the velocities in the X and Y directions respectively. The important thing here is the use of subscript I-1. It indicates that the current and the previous instances of XPOS, YPOS and TIME are needed in defining the current instance of VX and VY. Therefore two storage spaces are needed each for PF.TIME, XPOS and YPOS. The computation can be carried out with the following program logic:

```
Do for the range of I;
  Read one instance of record PR;
  Define one instance of VF.TIME;
  Define one instance of VX;
  Define one instance of VY;
  Write one instance of record VR;
end;
```

### 4.3.3 PHYSICAL STORAGE ALLOCATION SCHEME

There are cases where main storage space for an entire array dimension is needed because references to the elements along that dimension are not in any order. This storage

allocation scheme is referred to as physical. This is illustrated in Figure 4.5.

```
 1  MODULE: P;
 2  SOURCE: MF,NF;
 3  TARGET: RF;
 4
 5  1 MF IS FILE,
 6    2 MR(*) IS RECORD,
 7      3 NAME IS FIELD(CHAR(30));
 8
 9  1 NF IS FILE,
10    2 NR(*) IS RECORD,
11      3 SNO IS FIELD(NUM(4));
12
13  1 RF IS FILE,
14    2 RR(*) IS RECORD,
15      3 NAME IS FIELD(CHAR(30));
16
17  I IS SUBSCRIPT;
18
19  RF.NAME(I) = MF.NAME(SNO(I));
```

Figure 4.5 An Example of Physical Storage Allocation Scheme

Two source files, MF and NF, are used in the specification (line 2). MF is a list of names (lines 5 to 7) and NF is a list of serial numbers (lines 9 to 11). The specification defines a report file (RF) as containing the names from MF in the order according to the numbers in NF.

The assertion in line 19 defines this order. It states that the I-th name on the report file is the SNO(I)-th name on the source file. Since the relationship between SNO(I) and I is unknown, the definition (input from an external device) of MF.NAME instances and references to them will have to be scheduled in separated loops as:

```
Do for the range of I;
  Read one instance of record in MF;
end;
Do for the range of I;
  Read one instance of record in NF;
  Define one instance of RF.NAME;
  Write one instance of record in RF;
end;
```

Furthermore, all instances of MF.NAME have to reside in main storage in order to be used in defining an instance of RF.NAME, because we don't know which instance of MF.NAME is to be used. The storage allocation scheme for MF.NAME is therefore physical.

## 4.4 SUBSCRIPT EXPRESSIONS THAT SUPPORT VIRTUAL AND WINDOW ALLOCATIONS

The storage allocation scheme used for a dimension of a variable determines 1) the type of subscript expression used for that dimension, and 2) whether the variable is used in defining other variables with different ranges. From the examples in the previous sections, it can be seen that if

the subscript expression is of the form I, the storage allocation scheme can be virtual. If it is of the form I-k, where k is a positive integer, then a window of k+1 is sufficient. The MODEL processor recognizes the following subscript expressions for virtual and window allocation:

1. I
2. I-1
3. I-k
4. X(I), X is sublinear
5. X(I)-1, X is sublinear
6. X(I)-k, X is sublinear
7. Sawtooth subscript expressions

If the subscript expression is of the form I or X(I), and the definition and all references can be placed in one loop, only the storage space for one element is needed. For I-1 or X(I-1)-1, a window of 2 suffices. I-k and X(I)-k are generalized forms of the previous case. They indicate the need for k+1 elements. Sawtooth subscript expressions are used in merging a set of ranges into another. For subscript expressions having formats other than those mentioned above, the system quits further analysis and physical allocation schemes are used.

## 4.5 OPTIMIZATION IN CODE GENERATION

In code generation, the MODEL processor performs another level of optimization [Szym82]. It analyzes statements inside a loop, checks whether the window storage may be further reduced, and removes unnecessary copying operations.

```
 1  MODULE: STACKOP;
 2  SOURCE: DATA;
 3  TARGET: ST;
 4
 5  1 DATA IS FILE,
 6    2 INREC(*) IS RECORD,
 7      3 SEQNUM IS FIELD(DEC FIX(4)),
 8      4 VALUE IS FIELD(DEC FLOAT);
 9
10  1 ST IS FILE,
11    2 VERSION(*) IS RECORD,
12      3 S(*) IS FIELD(DEC FLOAT);
13
14  (I,J) ARE SUBSCRIPT;
15  SIZE.S(I) = IF I=1 THEN 1
16              ELSE IF SEQNUM(I)>SEQNUM(I-1)
17                   THEN SIZE.S(I-1)+1
18                   ELSE SIZE.S(I-1);
19  IF I=1
20  THEN S(I,J)=VALUE(I);
21  ELSE IF SEQNUM(I)=SEQNUM(I-1)
22       THEN S(I,J) = IF J=SIZE.S(I)
23                     THEN S(I-1,J)+VALUE(I)
24                     ELSE S(I-1,J);
25       ELSE S(I,J) = IF J=SIZE.S(I)
26                     THEN VALUE(I)
27                     ELSE S(I-1,J);
```

Figure 4.6 An Example of Optimization for Stack-like Structure

This optimization is illustrated by the example shown in Figure 4.6. The source is a file of records (INREC) consisting of a sequence number SEQNUM and an associated value VALUE. The target is a two-dimensional array S of sums obtained by adding up those values in the source

records which contain the same sequence number. Figure 4.7 illustrates the organization of the source and target data. The definition of the target data, the elements of each row with the exception of the first element in the row, are obtained by copying the data from the previous row. These copy operations can be saved if the last same storage is used for the row, adding only the new element. The saving is not just the time spent in the copy operation, the reduction in memory space is even more significant.

| | Source | Target | | | |
|---|---|---|---|---|---|
| 1 | #1,V1 | V1 | | | |
| 2 | #1,V2 | V1+V2 | | | |
| 3 | #2,V3 | V1+V2 | V3 | | |
| 4 | #2,V4 | V1+V2 | V3+V4 | | |
| 5 | #2,V5 | V1+V2 | V3+V4+V5 | | |
| 6 | #3,V6 | V1+V2 | V3+V4+V5 | V6 | |
| 7 | #4,V7 | V1+V2 | V3+V4+V5 | V6 | V7 |
| 8 | #4,V8 | V1+V2 | V3+V4+V5 | V6 | V7+V8 |

Figure 4.7 Illustration of Source and Target File in the Specification in Figure 4.6.

This optimization consists of three steps. First, a list of structures which are possibly eligible for window size reduction is produced. The program is scanned to see whether there are any unnecessary assignments on the structures. The result of this step is a list of structures

whose dimensions can be reduced. The second step is to mark all unnecessary copy operations as removable. And the final step is to modify the data structures and the object code involved based on the results from the first two steps, resulting in an equivalent, but more efficient, object program.

# CHAPTER 5

# OVERVIEW OF THE EXTENDED MODEL PROCESSOR

The MODEL language processor accepts a MODEL specification as input, verifies its correctness, and generates a program in the target language. The sequence of processing is illustrated in Figure 5.1. Boxes in the figure represent major phases of the processor. The arrows indicate the flow of execution. Between boxes are various representations of the user specification. Starting with the source specification, the MODEL processor performs a series of transformations until a program in the target languages is obtained.

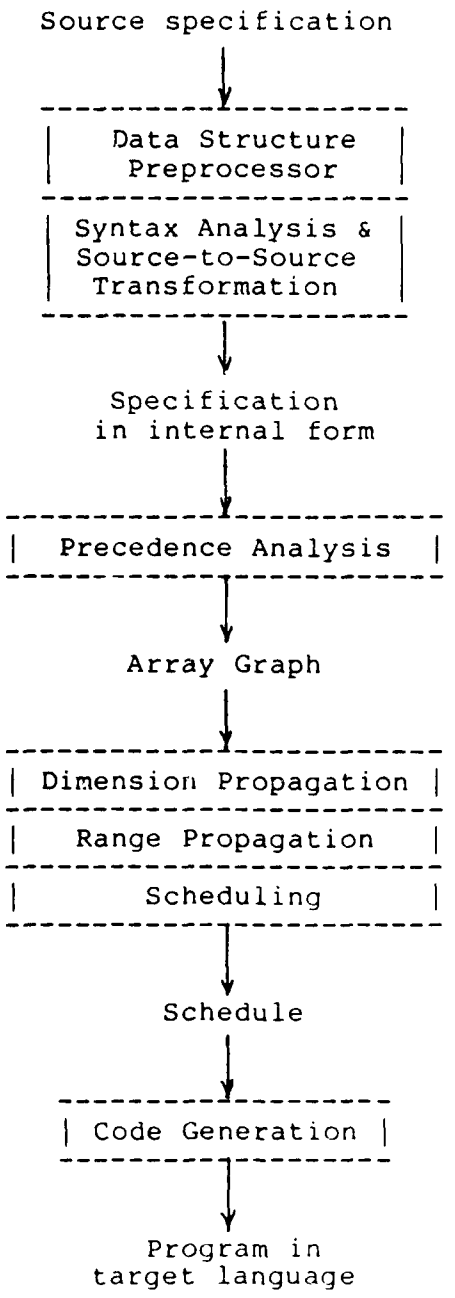


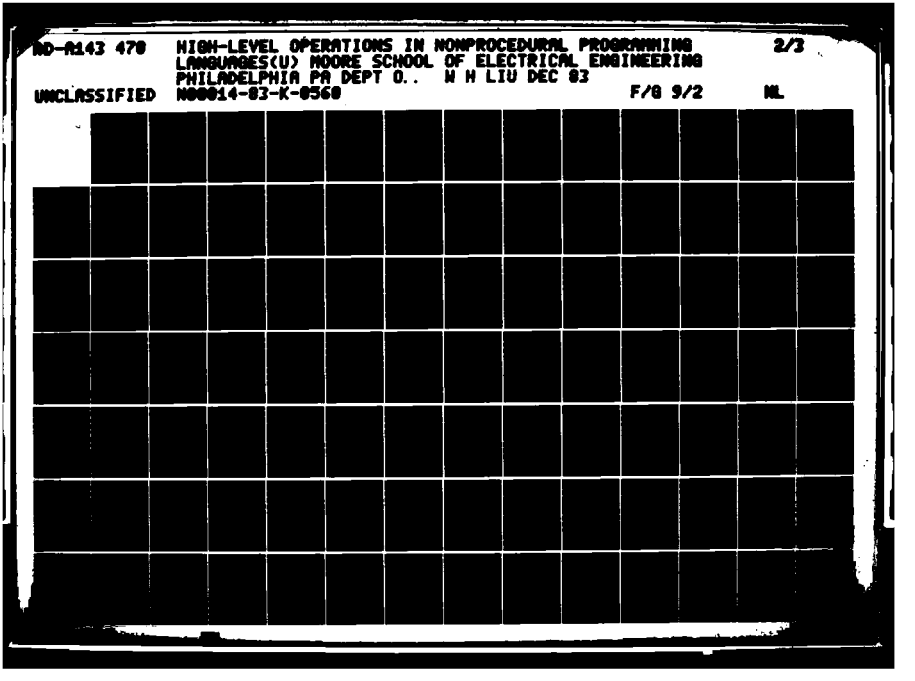

Figure 5.1 The MODEL language processor

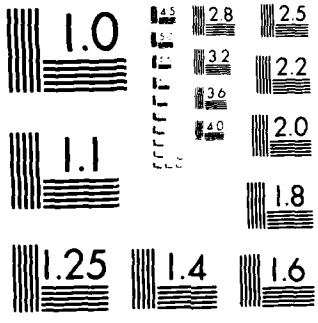
The first two boxes show a source MODEL specification scanned in two passes. First pass is by the Data Structure Preprocessor for the extraction of data structure information needed in the transformation of high-level operations. The second pass is by the Syntax Analysis and Source-to-Source Transformation module which parses the specification and transforms the high-level operations into elementary operations. After being analyzed syntactically, the source statements are stored in an internal form for efficient retrieval.

The next box analyzes precedence among statements and creats a data-flow network, called the array graph. It is a directed graph where nodes represent variables or assertions, and edges show dependency relationships between nodes.

The next series of boxes perform consistency checks by evaluating consistency of attributes along the edges. Attributes such as number of dimensions and dimension size are propagated from node to node to detect and resolve any inconsistencies and incompleteness.

Finally, depending on the type of the target machine (von Neumann or others), the verified array graph is rearranged topologically to obtain a schedule for that particular type of machine. It is then further translated


AD-A143 470
HIGH-LEVEL OPERATIONS IN NONPROCEDURAL PROGRAMMING LANGUAGES(U) MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA PA DEPT O.. W H LIU DEC 83
2/3
UNCLASSIFIED
N00014-83-K-0560
F/G 9/2
NL




MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS

into the desired target language.

The array graph representation is universal in the sense that it does not depend on any particular type of machine or language. So is the verification process. However, scheduling is machine type dependent, and code generation depends further on the chosen target language. This implementation is based on the conventional von Neumann machine and the PL/I target language.

The overview of the expanded version of the MODEL processor, modified to incorporate high-level operations, is described in this chapter. The processing phases of the modified MODEL processor have been modified to include a new Data Structure Preprocessor as a separate phase. Additionally, the following phases have been modified as follows:

a) Syntax Analysis -

Addition of source-to-source transformation routines.

b) Precedence Analysis -

Recognition of integral operations and the creation of edges for them. The integral operations are implemented by generating a block of PL/I code to be inserted in the object program. This phase also recognizes the various types of subscript expressions (such as the indirect

'sawtooth' indexing).

c) Range Propagation -

Processing of propagating sawtooth subscript expressions has been added. As described in Chapter 3, the sawtooth subscript expression defines index values which increase monotonically and then are reset to the value of 1. This sequence of subscript values needs special handling to obtain efficient implementation.

d) Scheduling -

Extensions for the handling of sawtooth subscript expressions.

e) Code Generation -

Generation of procedure calls for integral operations.

The following sections describe briefly the above mentioned modifications, as well as how various phases of the processor work. Detailed descriptions of the extensions is given separately in the following chapters.

## 5.1 DATA STRUCTURE PREPROCESSOR

The data structure preprocessor performs a preliminary scan ot the user's specification and extracts data structure information. During this first pass, all variable names,

their types, and the hierarchical relationships are saved in a table accessible to later processing phases. The preprocessor thus facilitates the transformation of high-level operations. The source-to-source transformation requires data structure information of variables. Since there is no restriction on statement order within a MODEL specification, the declaration of a variable may or may not appear before the reference to it.

For example, the assertion SA=SORT(SB,B1,ASC) in the specification shown in Figure 3.3 involves the high-level function SORT. The structures SA and SB must be determined for checking their compatibility and the declaration of interim variables, prior to transforming the SORT operation. The information gethering on SA and SB and the transformation of SORT are handled separately in two different passes. The preprocessor saves information about SA and SB in a table in the first pass, and the syntax analyzer (described in the next section) invokes the transformation process in the second pass.

The data structure preprocessor does not modify the source specification at all. It simply scans every source statement and stores the extracted information in a separate area.

## 5.2 SYNTAX ANALYSIS AND SOURCE-TO-SOURCE TRANSFORMATION

The user specification is scanned the second time by the syntax analyzer which, besides parsing and storing the source statements, performs the transformation of high-level operations.

The syntax analyzer scans the MODEL specification statement by statement. Entries are created in the data dictionary for data description statements, to store the variables and their attributes. Source assertion statements are converted into internal assertion trees. When an assertion references a high-level operation, the associated transformation routine is called to replace the assertion with a set of statements according to the transformation rule. The source statements generated as a result of the transformation replace the high-level operations. These system generated statements are processed just like others written by the user. Thus, during this phase of syntax analysis, the input stream may 'grow' in length whenever a source-to-source transformation takes place.

The process of transformation is illustrated as follows, using the SORT example shown in Figure 3.3. The specification defines a file (SA) whose records are those of another file (SB) sorted in ascending order according to the value of the field B1. This is expressed in the assertion

SA=SORT(SB,B1,ASC). When the transformation is being carried out for SORT, this source statement is replaced by system generated statements as follows:

```
 1   O1$CG IS GROUP(O1$C(*));
 2   O1$C IS FIELD(CHAR(3));
 3   O1$C=B1;
 4   O1$IG IS GROUP(O1$I(*));
 5   O1$I IS FIELD(NUM(5));
 6   O1$IG=SORTC(O1$CG,1,SIZE.B);
 7   SIZE.O1$B=SIZE.B;
 8   A1(SUB1)=B1(O1$I(SUB1));
 9   A211(SUB1,SUB2,SUB3)=B211(O1$I(SUB1),SUB2,SUB3);
10   A212(SUB1,SUB2,SUB3)=B212(O1$I(SUB1),SUB2,SUB3);
11   A22(SUB1,SUB2)=B22(O1$I(SUB1),SUB2);
12   SIZE.B=30;
```

These system generated statements specify in detail the desired SORT operation. The idea is to use the interim array variable O1$C (declared in lines 1-2) for the key field B1 (line 3), then define an integer array O1$I which indicates relative order of the elements in O1$C (line 6). SORTC stands for sorting character strings. Array O1$I is then used to express the index correspondences between the source field indices and the target field indices (lines 8 through 11).

## 5.3 THE ARRAY GRAPH REPRESENTATION

The array graph of a MODEL specification is a directed graph which shows precedence relationships among data and assertions. A node in the graph represents either a

variable or an assertion. A variable may have a number of elements and an assertion may repeat for all subscript values within respective ranges. A node in the array graph in general represents either an array of variable occurrences or a multiple applications of an assertion. An edge between two nodes shows their hierarchical (both nodes are variables) or data dependency (one variable and one assertion) relationship.

Associated with each node are a number of attributes: number of dimensions of the node, each dimension range, etc. Edge attributes include the difference in number of dimensions between the source and the target nodes and how respective dimensions correspond to each other. Information associated with nodes can be propagated back and forth along the edges for the detection of possible conflicts and their corrections.

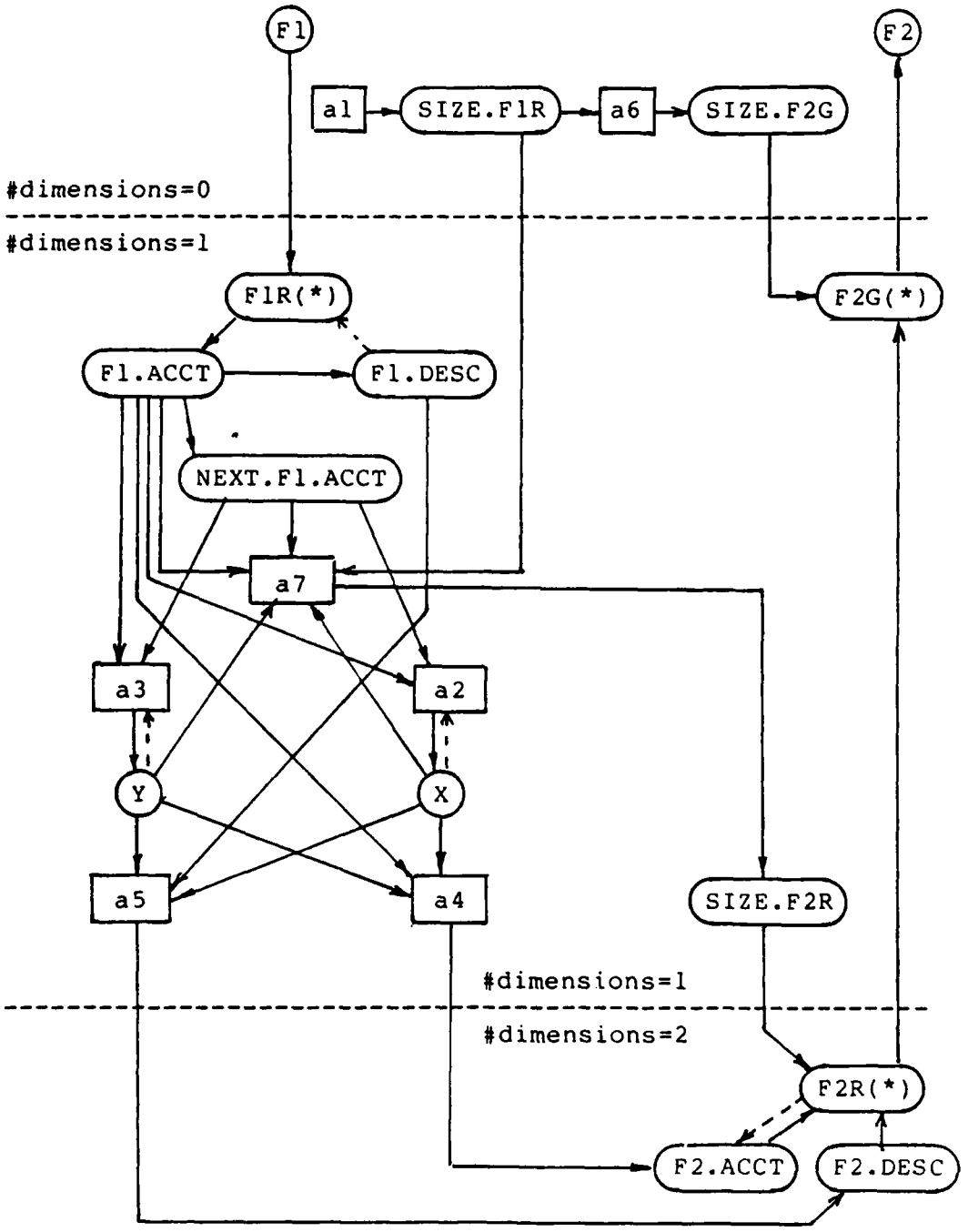



Figure 5.2 The Array Graph of the Specification in Figure 3.2

Figures 5.2 and 5.4 illustrate array graphs for two examples. Figure 5.2 is the array graph of the MODEL specification example shown in Figure 3.2. Both elliptical and rectangular boxes represent nodes, with the former for variables and the latter for assertions. Nodes with the same number of dimensions are grouped together in the same area for clarity. Arrows represent edges. A hierarchical edge, such as the one from F1R(*) to F1.ACCT, indicates that the structure has to be accessed (a F1R record is read) before its components can be available (F1.ACCT field unpacked). A data dependency edge such as the one from F1.ACCT to a3 indicates that the evaluation of the target (assertion a3) depends on the availability of the source (field F1.ACCT). The dashed edges shows that accessing the next structure (e.g., F1R(*)) must follow the current element (F1.DESC).

As far as high-level operations are concerned, array graph represents the specification after source-to-source transformation. For example, the transformation of the specification in Figure 3.3 results in the one shown in Figure 5.3. The array graph is created based on the specification shown in Figure 5.3, not the original one shown in Figure 3.3. This array graph is shown in Figure 5.4.

```
MODULE: SORTREC;
SOURCE: SB;
TARGET: SA;

1 SB IS FILE,
  2 B(*) IS RECORD,
    3 B1 IS FIELD(CHAR(3)),
    3 B2(3) IS GROUP,
      4 B21(2) IS GROUP,
        5 B211 IS FIELD(CHAR(5)),
        5 B212 IS FIELD(CHAR(5)),
      4 B22 IS FIELD(CHAR(4));

1 SA IS FILE,
  2 A(*) IS RECORD,
    3 A1 IS FIELD(CHAR(3)),
    3 A2(3) IS GROUP,
      4 A21(2) IS GROUP,
        5 A211 IS FIELD(CHAR(5)),
        5 A212 IS FIELD(CHAR(5)),
      4 A22 IS FIELD(CHAR(4));

    O1$CG IS GROUP(O1$C(*));
    O1$C IS FIELD(CHAR(3));
b1: O1$C=B1;
    O1$IG IS GROUP(O1$I(*));
    O1$I IS FIELD(NUM(5));
b2: O1$IG=SORTC(O1$CG,1,SIZE.B);
b3: SIZE.O1$B=SIZE.B;
b4: A1(SUB1)=B1(O1$I(SUB1));
b5: A211(SUB1,SUB2,SUB3)=B211(O1$I(SUB1),SUB2,SUB3);
b6: A212(SUB1,SUB2,SUB3)=B212(O1$I(SUB1),SUB2,SUB3);
b7: A22(SUB1,SUB2)=B22(O1$I(SUB1),SUB2);
b8: SIZE.B = 30;
```

Figure 5.3 Result Specification of Figure 3.3 after Transformation

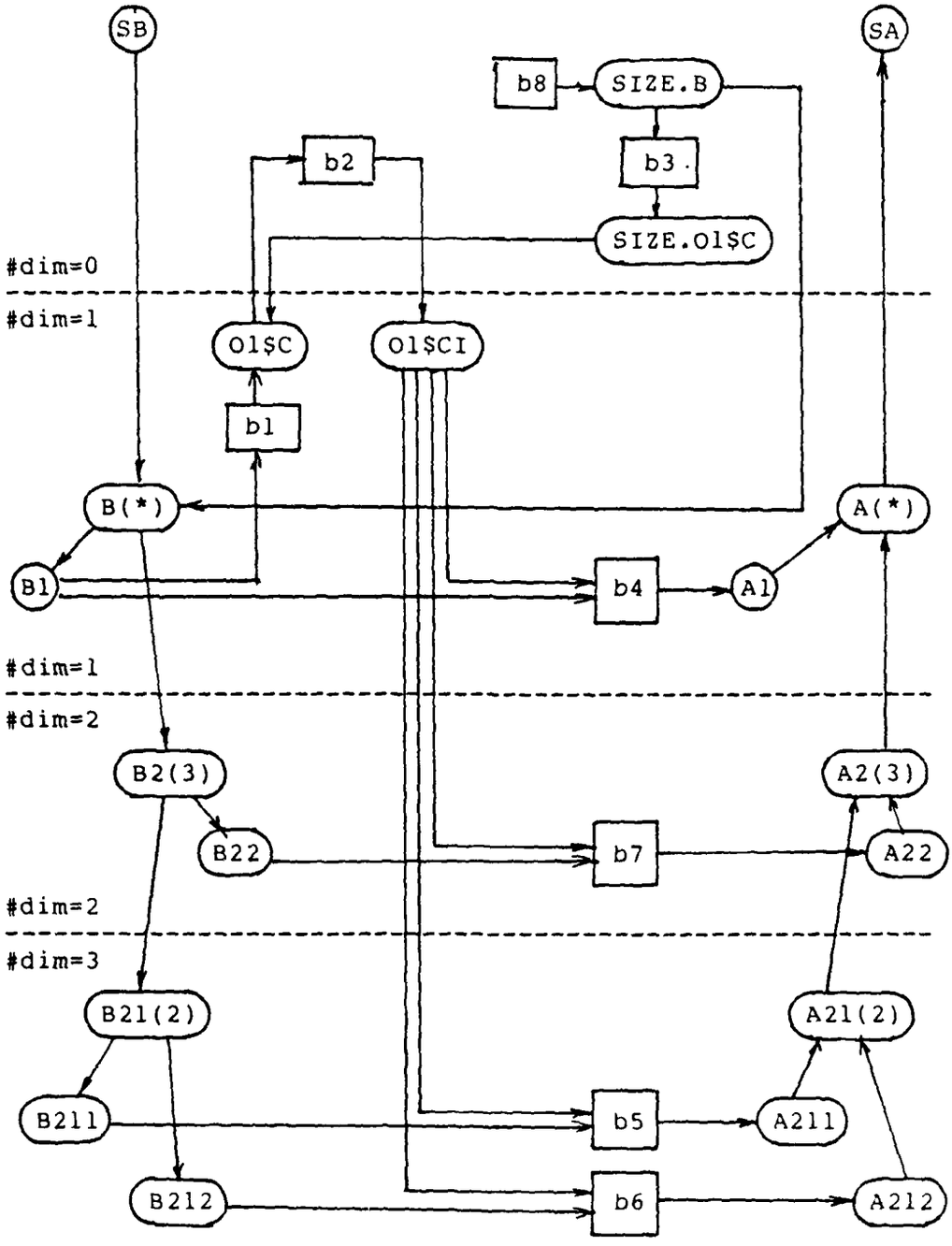



Figure 5.4 Array Graph of the Specification in Figure 5.3

## 5.4 PRECEDENCE ANALYSIS

This section describes how edges are created to form an array graph. The extensions to the precedence analysis phase for the incorporation of high-level operations involve only the handling of integral operations.

Precedence analysis determines various relationships between variables and assertions and enters these relationships in the array graph with edges. There are basically three categories of edges - hierarchical, data dependency, and data parameter. Hierarchical edges are drawn between components of a data structure to show the structure hierarchy. They are created from the data description statements. Data dependency edges are drawn from source variables to an assertion and from the assertion to the target variable, to indicate the order in which variables are accessed and evaluated. Data parameter edges are drawn between a data parameter variable node using keyword prefixes such as END, SIZE or NEXT and the data node in the prefix.

The information associated with an edge includes the following:

1) edge type,
2) the source node of the edge,

3) the target node of the edge,
4) difference in the numbers of dimensions between the source and the target nodes,
5) subscript expressions of the source dimensions.

Hierarchical edges are created by the Enter Hierarchical Relationships program (ENHRREL). The program basically traverses the data definition trees, determines the hierarchical relationships between nodes, and enters these relationships as edges in the array graph. These edges play an important role in the I/O activities of the generated program. They not only show the order of I/O operations (e.g., a record must be read before the unpacking any field), they also indicate how the I/O activities are repeated (e.g., next record can not be read unless the last field of the current record is unpacked).

Data dependency edges are created by the Enter Dependency Relationships program (ENEXDP). For each assertion in the specification, this program scans the expressions on both sides of the equal sign to obtain the target variable and the source variables of the assertion. An edge is created from each source variable to the assertion, and from the assertion to the target variable. They simply mean that the source variable must be available before the assertion can be applied and that the target

variable will be available only after the assertion is evaluated.

For integral operations, which may reference high-level variables, edges are drawn from all fields in each source variable to the assertion, and from the assertion to all fields in the target variable.

The ENEXDP program also creates the data parameter edges for all the control variables used in the specification. Control variables are of the form <control_prefix>.<variable_name>, where <control_prefix> could be POINTER, SIZE, END, FOUND, NEXT, MALDATA, or SUBSET. An data parameter edge is drawn from the control variable to the variable whose parameter is being defined. The control variables define properties associated with the array denoted by <variable_name>. The meanings of individual control variables are given in detail in [LuKS82].

## 5.5 DIMENSION AND RANGE PROPAGATIONS

When the array graph is first constructed, the number of dimensions and the dimension ranges associated with each node in the graph may not be explicitly stated, because subscripts in assertions can be implicit and that the data descriptions for some interim variables may be omitted.

However, because each edge bears the information about the difference in source and target dimensionalities, the incompleteness can be resolved by propagating the dimensions and the ranges along the edges until a solution that satisfies all nodes in the graph is obtained.

The process of dimension propagation is based on the idea that the dimensionality of a node should agree with those of its neighboring nodes, taking into account the dimension differences associated with the connecting edges. The process starts with creating a queue which contains all nodes in the graph. The nodes are taken from the queue one by one and are checked to see whether the neighboring nodes agree with the dimension differences associated with the corresponding edges. For any neighboring node which does not agree, the number of dimensions may be increased. It is then appended to the queue to be checked again. The process may or may not converge. If the process converges, then every node in the graph has a finite number of dimensions. If it diverges, the processor will issue an error message showing those nodes which have inconsistent numbers of dimensions.

Another property of a node which can be obtained by propagation is the range of a dimensions. Range propagation refers to the partitioning of node dimensions into a number

of range sets. All node dimensions in a range set share a common dimension range. The partitioning starts with assuming that every node dimension constitutes a range set. Then based on any one of the following relations, node dimensions of two range sets are merged together to form a larger one:

1) Two node dimension both subscripted by the same global subscript.
2) One of the node dimensions corresponds to a data node and the other corresponds to the same dimension position of the associated control variable.
3) The two node dimensions occur on the same dimension position of two data nodes in the same data structure.
4) One node dimension is associated with an assertion node and the other with a source variable of the assertion.
5) One node dimension is associated with an assertion node and the other with the target variable of the assertion.

Associated with each range set is a termination criterion by which its range is defined. The criteria are derived from the user specification and can be one of the following:

1) The range set has a constant range.
2) The range is specified by a END control variable.

3) The range is specified by a SIZE control variable.
4) The range is implicitly defined by the end-of-file condition at run-time.
5) The range is implicitly defined by the end-of-record condition

When the partitioning of node dimensions into range sets is completed, each range set will have one of the above range termination criteria associated with it. If there is a range set which does not have a termination criterion defined, a message will be issued by the processor to report the error.

For the purpose of optimization, certain indirect subscript patterns associated with variables in the specification are recognized as sawtooth subscript expressions. The association with such a sequence is also a property that can be propagated along edges. The sawtooth subscript expressions are propagated to determine which other nodes in the graph can also be associated with such indirect indices. The result of this propagation is used to improve efficiency.

## 5.6 SCHEDULING THE ARRAY GRAPH

The array graph shows only the data dependency of a computation, not its execution sequence. To realize the computation on a given machine, it is necessary to transform the array graph into another representation, called a schedule, which resembles more closely the execution style of the machine.

A schedule depends on the type of the target machine, but is independent of the target language. This section describes briefly the scheduling for von Neumann type machines. Scheduling data-flow machines is discussed in [Gokh83].

The execution style of a von Neumann machine is sequential in nature. A schedule is essentially a linear rearrangement of the nodes in the array graph according to the partial order imposed by the edges. The general approach to this scheduling consists of creating a component graph, with each component containing an MSCC (maximally strongly connected component) in the array graph, and the edges connecting the MSCC's. The component graph is an acyclic graph and hence can be topologically sorted. When the component graph is sorted, a gross-level representation of the schedule is obtained.

Before sorting an MSCC, it is decomposed first by deleting edges which have an I-k or a X(I)-k type of subscript expression associated with it. This will result in an acyclic subgraph which can be further sorted.

In addition to sorting the nodes as described above, the scheduler also tries possible enlargement of loop scopes as described in Section 4.2. It is based on the information about subscript expressions in order to produce a more efficient program. When a node in the graph is associated with a sawtooth subscript expression, its dimensions covered by the subscript expression can be all scheduled in a single loop, as opposed to many nested ones. This eliminates the unnecessary loop opening end closing, which usually impose additional storage requirements in the generated program.

If an array graph cannot be fully decomposed by the scheduling process, the MSCC is interpreted as representing a set of simultaneous equations. An iterative numerical method is employed for evaluating the equations [Gree81].

The scheduling phase has been extended to handle nodes which are related through the use of sawtooth subscript expressions. Such related nodes are scheduled together as conditional blocks for the generation of more efficient object code. Detailed description is given in Chapter 8.

## 5.7 CODE GENERATION

Code generation is a process of translating the schedule into a program in the target language. There are three types of elements in the schedule: node-element, for-element and cond-element. A node-element corresponds to a basic statement in the target language such as assignment or I/O operation. A for-element corresponds to a loop such as DO-FOR or DO-WHILE. A cond-element represents a conditional block of code which is only to be executed conditionally, not for every loop instance of the enclosing loop. The cond-elements are used for those nodes whose range is different from, but sublinear to, the range of the enclosing loop. These nodes are scheduled within the major loop for optimization purpose.

Code generation is a straightforward translation of elements in the schedule:

1) Scan elements in the schedule. For each element, perform steps 2 through 4.
2) If the element is a node-element, generate the corresponding target language statement for that node.
3) If the element is a for-element, open a loop, generate code for the enclosed elements recursively, and close the loop.
4) If the element is a cond-element, generate an IF

condition and a DO block header, generate code for the enclosed elements recursively, and close the block with an END.

The above procedure sets up the control structure of the generated program. For individual nodes (step 2), the correspondence between node types and the generated program statements are as follows:

| | | |
|---|---|---|
| FILE node | - | OPEN/CLOSE a file |
| input RECORD node | - | READ a record to input buffer |
| output RECORD node | - | WRITE a record from output buffer |
| interim GROUP node | - | nothing |
| input FIELD node | - | unpack a field from input buffer |
| output FIELD node | - | pack a field to output buffer |
| interim FIELD node | - | nothing |
| assertion node | - | an assignment |
| integral operation | - | a procedure call |

Extensions to the code generation phase include 1) the generation of more efficient code for conditional blocks resulting from the use of sawtooth subscript expressions, and 2) the generation of procedure calls for integral operations. They are described in detail in Chapter 8.

# CHAPTER 6

# PREPROCESSING AND SYNTAX ANALYSIS

The MODEL syntax analyzer is automatically generated. It recognizes components of the MODEL language in a top-down, recursive manner. When a component such as a variable or an assertion is recognized, it calls a subroutine specific to that component to do the desired analysis and storing. The automatic generation of the syntax analyzer makes it very straightforward to extend the syntax of the language. To take advantage of this, high-level operations are transformed as soon as they are recognized. This allows the transformation procedures to perform only the transformation related tasks, leaving the syntax related problems to the syntax analyzer. This 'on the spot' transformation process requires that the information needed for the transformation be generated in a prior pass. The module that preprocesses the data description statements is the data structure preprocessor

(PRESAP). It is the very first module executed in processing a MODEL specification. Its relationship with the normal syntax analyzer (SAP) is shown in Figure 6.1.

solid boxes: program modules
dashed boxes: data modules
solid arrows: execution sequence
dashed arrows: data flow

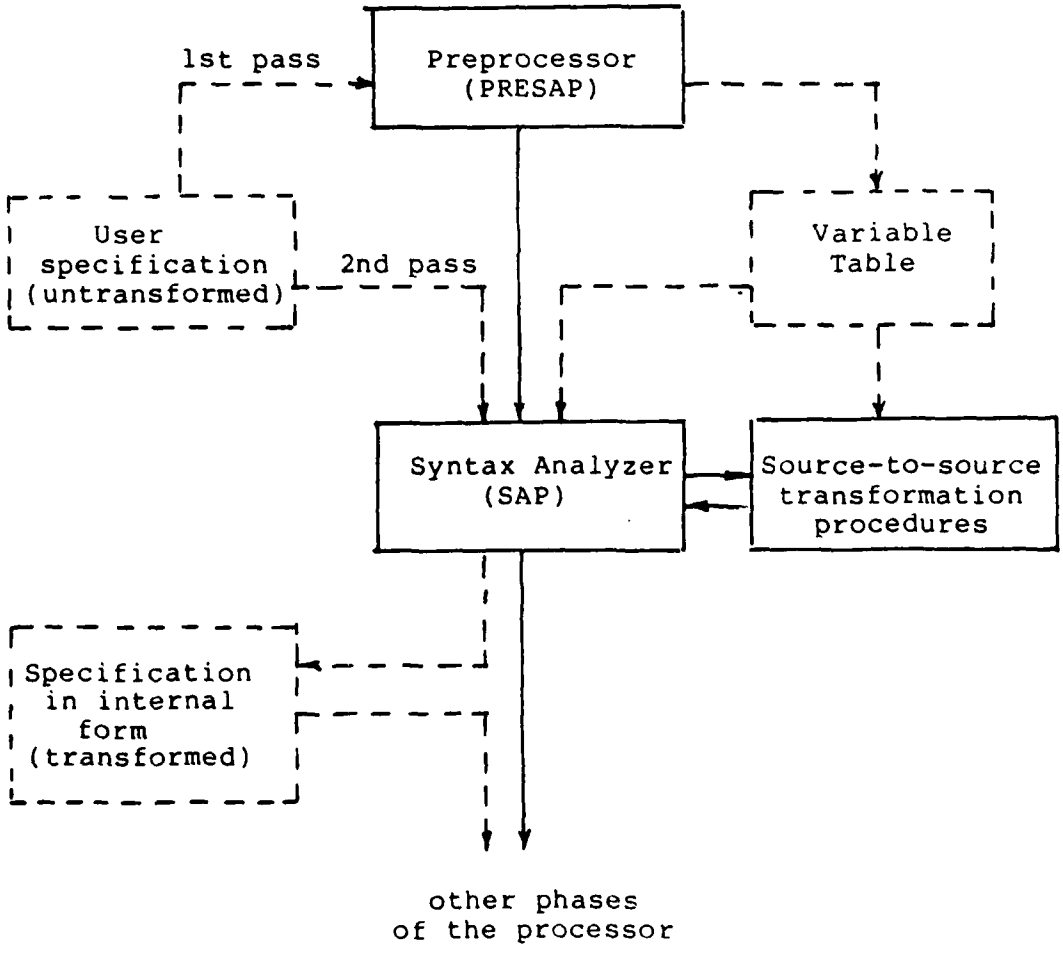


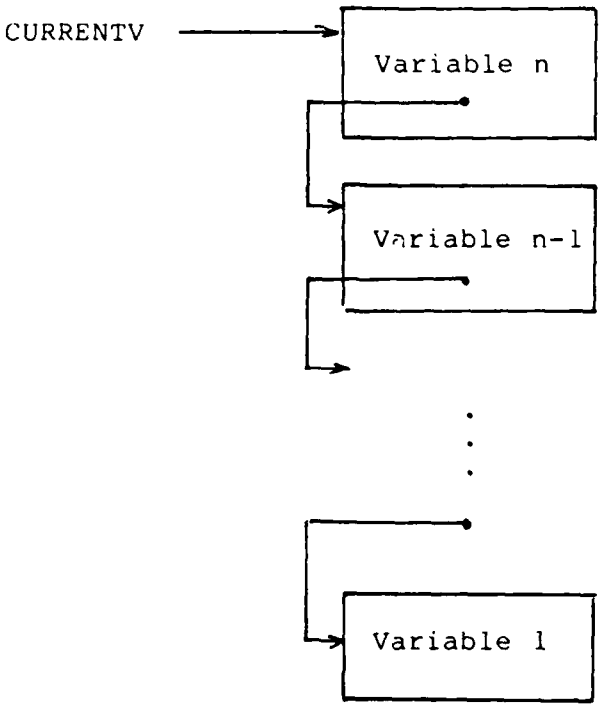

Figure 6.1 Syntax Analysis for Source-to-Source Transformation

which references them. Since the transformation of a high-level operation depends on the attributes of the referenced variables and the hierarchical relationships involved, it is necessary to assemble the required information before the transformation can be carried out.

Data descriptions can be written in two ways in MODEL. The first one follows the PL/I structure declaration format, where level numbers are used to express the hierarchical relationships. The following data description, taken from the data structure example in Figure 3.1, uses this format.

```
1 DEPT IS FILE,
  2 DEPTNO IS FIELD(NUM(4)),
  2 EMPLOYEE(*) IS GROUP,
    3 EMPNO IS FIELD(NUM(6)),
    3 NAME IS FIELD(CHAR(30)),
  2 PROJECT(10) IS GROUP,
    3 PJNO IS FIELD(NUM(5)),
    3 EQUIP(*) IS GROUP,
      4 ITEMNO IS FIELD(NUM(10)),
      4 DESC IS FIELD(CHAR(30));
```

The other format uses parentheses to indicate the immediate descendents of a variable:




Each entry in the table contains:

VNAME - variable name, unqualified
VTYPE - variable type, file, record, group, or field
LEVEL_NO - level number used in data definition
FATHER - father of this variable
SON - the first son of this variable
BROTHER - the immediate younger brother
OCCUR - repetition of the variable
CIMAGE - the line image which contains the data definition statement
LAST_DUP - the next variable in the table with the same name
NEXTV - pointer to next entry

Figure 6.2 Data Structure of the Variable Table

Algorithm 6.1 ADD_F1_ENTRY

Given a line of data description statement of format-1, enter the variable into the variable table.

Input: NM - Variable name, unqualified.

TY - Node type of the variable, file, group, record or field.

LN - Level number of the variable declared by the user.

OC - Occurrence, single, fixed multiple or variable multiple.

Data structure: 1. The Variable Table.

2. The ancestor stack ANCESTOR_STK, initially empty.

Method:

1. Allocate one entry for the input variable. Set the VNAME field to NM, VTYPE to TY, LEVEL_NO to LN, and OCCUR to OC.
2. If the variable table is empty, then push the current variable on ANCESTOR_STK.
3. If there is an variable in the table with the same name as NM, then set up the LAST_DUP link.
4. If LN equals LEVEL_NO of the last entered variable, then

set up the FATHER link of this entry and the BROTHER link of the last entry.

5. If LN is greater than LEVEL_NO of the last entered variable, then set up the SON link of the last entry and the FATHER link of the current entry. Also push current entry in ANCESTOR_STK.
6. If LN is less than LEVEL_NO of the last entered variable, pop ANCESTOR_STK until LN is equal to LEVEL_NO of the top element on stack. Then set the current variable to be the BROTHER of the variable on top of the stack.

Algorithm 6.2 ADD_F2_VENTRY

Given a line of data description statement of format-2, enter the variable into the variable table.

Input: NV - Number of variables appeared in the line.
NM - Array of NV variable names.
OC - Array of NV variable occurrences
TY - Type of NM(1)

Date structure: The Variable Table

Method:

1. If TY is 'FLD', then for each name in NM, search for the entry which has the same name, enter 'FLD' as VTYPE of the entry. Also set the SON link of the entry to null.

2. Search for the entry with VNAME equal to NM(1). If found, set its VTYPE to TY If not found, create an entry for this variable, enter the SON link and save the pointer to this entry as COMMON_FATHER.
3. For every variable in the array except NM(1), create an entry in the table, set their FATHER link to COMMON_FATHER and BROTHER to its next neighboring variable.

Algorithm 6.3 VT_LOC

Locate an entry in the Variable Table with the specified variable name.

Input: NM - variable name, may be qualified with a file name.

Output: Pointer to the entry whose name matches NM, null if not found.

Data structure: The Variable Table.

Method:

1. If NM is qualified (with a dot '.'), extract the two subnames to NAME1 and NAME2 respectively. If NM is not qualified, set NAME1 to empty string and NAME2 to NM.
2. Search the variable table for the entry whose VNAME is the same as NAME2. If not found, return null.

3. If NAME1 is null, return with the pointer obtained in step 2.
4. Check whether any ancestor of NAME2 is NAME1. If so, return with the pointer obtained in step 2. Otherwise return null.

## 6.2 THE SYNTAX ANALYZER

The MODEL syntax analyzer is automatically generated by Syntax Analysis Program Generator (SAPG). The syntax of the MODEL language and the associated processing is specified using the meta language EBNF/WSC (extended BNF with subroutine calls). SAPG accepts the EBNF/WSC and generate a corresponding Syntax Analysis Program (SAP) to parse the MODEL specification in a top-down fashion. The following is an example showing the definition of arithmetic expression in EBNF/WSC:

```
<ARITH_EXP> ::= /E(81)/ /SAVE/ [ <SIGN> /SVOP1/ ] <TERM>
/SVCMP1/
                [ <OPS> /SVNXOP/ <TERM> /SVNXCMP/ ]* /STALL/
```

where nonterminals are enclosed in angle brackets < >, the square brackets [ ] denote an optional component, and the asterisk (*) indicates a zero or more repetitions. It specifies that <ARITH_EXP> consists of one or more <TERM>s separated by <OPS>, possibly prefixed with a <SIGN>. Names inside slashes (/) are subroutines to be invoked at that

specific point during parsing. Thus, for example, SVOP1 is called when <SIGN> is present, and STALL is executed when the whole right hand side is successfully recognized.

To process high-level operations, the EBNF/WSC is extended to recognize both the matrix operations and the high-level functions. For matrix operations, expressions formed via matrix operators are treated as a new component, denoted by <MFACTOR>, with a precedence between those of <TERM> and <FACTOR>. Processing involved includes saving the operators (SVOP1 and SVNXOP), remembering the nonterminal type (SVMFAC), and saving the expressions formed by the operators (SVCMP1 and SVNXCMP). The EBNF/WSC statements for <MFACTOR> is shown in Figure 6.3. The other two new subroutines, BMOPREC and UMOPREC, are used to recognize the binary matrix operators (|* and |/) and unary matrix operators (|/ and |^) respectively.

```
<TERM> ::= /E(87)/ /SVTERM/ <MFACTOR> /SVCMP1/
           [<MOPS> /SVNXOP/ <MFACTOR> /SVNXCMP/]* /STALL/

<MFACTOR> ::= /SVMFAC/ <FACTOR> /SVCMP1/
           [ <BMOP> /SVNXOP/ <FACTOR> /SVNXCMP/ ]* /STALL/

<FACTOR> ::= <UMOP> /SVMFAC/ /SVOP1/ <PRIMARY> /SVCMP1/
                /STALL/
         | /E(85)/ /SVFAC/ [^ /SVOP1/] <PRIMARY> /SVCMP1/
           [<EXPON> /SVNXOP/ <PRIMARY> /SVNXCMP/]* /STALL/

<BMOP> ::= /BMOPREC/

<UMOP> ::= /UMOPREC/
```

Figure 6.3 EBNF/WSC Statements for Matrix Operations

General array manipulation operations are treated as <PRIMARY>, as shown in Figure 6.4. The arguments to the called functions are also treated as <PRIMARY>. These recursive definitions enable the composition of functions to be recognized and processed automatically.

```
<PRIMARY>::= /E(86)/ /SVPRIM/ <IS_PRIM> /SVCMPl/ /STALL/

<IS_PRIM>::= ( <BOOLEAN_EXPRESSION> /E(24)/ )

     | SELECT ( <PRIMARY> /SLARGl/ , <BOOLEAN_EXPRESSION>
        /SLCOND/ ) /SLTRAN/

     | MERGE ( <PRIMARY> /MGARGl/ , <PRIMARY> /MGARG2/ ,
        <BOOLEAN_EXPRESSION> /MGCOND/ )  /MGTRAN/

     | SORT ( <PRIMARY> /SRARGl/ , <PRIMARY> /SRKEY/ ,
        <PRIMARY> /SRORDER/ )  /SRTRAN/

     | FUSE ( <PRIMARY> /FSARGl/ ) /FSTRAN/

     | COLLECT ( <PRIMARY> /CLARGl/ , <BOOLEAN_EXPRESSION>
        /CLCOND/ ) /CLTRAN/

     | CONCAT ( <PRIMARY> /CCARGl/ , <PRIMARY> /CCARG2/ )
        /CCTRAN/

     | UNIQUE ( <PRIMARY> /UQARGl/ ) /UQTRAN/

     | UNION ( <PRIMARY> /UNARGl/ , <PRIMARY> /UNARG2/ )
        /UNTRAN/

     | DIFF ( <PRIMARY> /DFARGl/ , <PRIMARY> /DFARG2/ )
        /DFTRAN/

     | PRODUCT ( <PRIMARY> /PRARGl/ , <PRIMARY> /PRARG2/ )
        /PRTRAN/

     | <NUMBER> /STNUM/

     | <STRING_FORM>

     | <FUNCTION_CALL>

     | <SUB_VARIABLEl>
```

Figure 6.4 EBNF/WSC Statements for General Array Operations

As can be seen in Figure 6.4, three groups of subroutines are added for the syntax analysis of high-level functions. They are named as XXARGn, XXCOND and XXTRAN, where XX is a pair of characters identifying the functions. Subroutines XXARGn store the pointers to the corresponding <PRIMARY> recognized. Subroutines XXCOND store the conditions <BOOLEAN_EXPRESSION>. Subroutines XXTRAN are the ones which perform the transformations based on the arguments saved by XXARGn and XXCOND. These transformation procedures will be explained in more detail in the next section.

A complete listing of the EBNF/WSC statements can be found in Appendix A.1. Note that it is for the second pass processing. Since the first pass involves only the extraction of data structure information, many subroutines called in the second pass are not needed. EBND/WSC statements for the first pass is a simplified version of those for the second one, with most of the subroutine calls (those enclosed with slashes '/') eliminated. Appendix A.2 shows the EBNF/WSC statements for the first pass.

## 6.3 TRANSFORMATION PROCEDURES

When the recognition of a high-level function is completed, the associated transformation procedure is invoked. These procedures have names ending with 'TRAN', such as SLTRAN for SELECT, MGTRAN for MERGE, etc. (shown in Figure 6.4). They are essentially the transformation rules for the corresponding functions written in PL/I. Inputs to each procedure are the arguments to the function, in the form of interim tree representation. Outputs from the procedure are the transformed MODEL statements, including both data declarations and assertion, in the form of source statements. Individual transformation rules are discussed in Chapter 7, this section describes only the conventions and frameworks common to all the transformation procedures.

Because many interim variables are used for the transformation and the same function may be referenced more than once, different tags of character strings are used to prefix interim variables for different activations of a transformation procedure. The tag used for a particular activation is the concatenation of 1) a character identifying the function, 2) an integer for the sequence number of the activation, and 3) the special character '$'. For example, the transformation procedure for the SELECT function uses L1$C for the interim selection array when

called the first time. If called again, it switches to L2$C just to differentiate the two activations. The same is true for declaring interim structured variables compatible to the user's data definition. For example, if a file is declared as

```
1 A IS FILE,
  2 AR(*) IS RECORD,
    3 AF1 IS FIELD(CHAR(3)),
    3 AF2 IS FIELD(NUM(5));
```

and SELECT(A,AF1='ABC') is to be transformed, then the following interim structure is used for the first activation:

```
1 L1$ IS GROUP,
  2 L1$AR(*) IS GROUP,
    3 L1$AF1 IS FIELD(CHAR(3)),
    3 L1$AF2 IS FIELD(NUM(5));
```

In generating the above data definition, the Variable Table (Section 6.1) is searched to obtain the information about the hierarchical structure.

For nested function calls, because inner level calls are processed first, outer level calls can be transformed by simply using the interim result (L1$ in the above example) for all the inner calls which have been processed up to that point. This is done without having to know how the inner calls are composed. The interim result is stored in variable RESULT in the SAP program. It automatically

then the last interim variable used (or intended to be used), O1$, will never appear in the transformed source statements. The overall transformation for the assertion consists of the definitions of the fields in L1$ in terms of those in A, and the fields in B in terms of those in L1$. Note that interim structured variables are always generated compatible to the source structured variable. The compatibility checking is performed only when necessary, i.e., at the very last moment between L1$ and B.

## 6.4 CHECKING OF VARIABLE COMPATIBILITY

The compatibility checking is done either by-name or by-structure, as described in Chapter 3. It is performed by procedures BYNAME and BYSTRU respectively. If the MODEL user does not specifically indicate which criterion to take, then by-name will have a higher priority than by-structure. If the checking is successful, the correspondence between the target field variables and the source field variables is saved in a table for later access. The checking involves traversing the data definition tree, which can be done following the links set up in the variable table described in Section 6.1. The following two algorithms describe the two ways of checking compatibility.

Algorithm 6.4 BYNAME

Algorithm 6.5 BYSTRU

Given two structured variables, T (target) and S (source), check whether they are compatible by structure. In other words, for every field t in the structure of T, there is a field s in the structure of S such that t and s share the same sibling position, same dimensionality, and same data type.

Input: Two pointers P1 and P2 to the two structured variables.

Output: FND=TRUE if P1 is compatible to P2, *FND=FALSE* otherwise.

Method (the procedure is recursive):

1. If the repeating factors of the two variables (P1->OCCUR and P2->OCCUR) are different, return with FND=FALSE.
2. If both P1 and P2 have a son, call BYSTRU(P1->SON,P2->SON,FND). If FND=FALSE, return.
3. If both P1 and P2 are fields, check for same type. If so, add the pair to the compatibility table and return with FND=TRUE. Otherwise return with FND=FALSE.
4. If both P1 and P2 have brothers, call BYSTRU(P1->BROTHER,P2->BROTHER, FND). If FND=FALSE, return.

If structure T is found to be compatible with structure S, then there exists a mapping from fields in T to those in S which satisfies the dimensionality and data type requirements for being compatible. The mapping is stored in the compatibility table by BYNAME or BYSTRU after the check is completed. The compatibility table (CT) is a flat table with two attributes: target field name (TFLDNAME) and source field name (SFLDNAME). Each entry in CT stores a pair of fields which corresponding to each other. The compatibility table is reterenced mainly by the source-to-source transformation procedures when defining target fields in terms of source fields.

# CHAPTER 7

# TRANSFORMATION OF HIGH-LEVEL OPERATIONS

This chapter describes source-to-source transformations for high-level operations, including analysis of the transformation problem, the transformation techniques, and the transformation rules for the high-level operations provided in the MODEL language.

The general transformation problem is presented in Section 7.1. It discusses the analysis conducted for the transformation of all high-level operations. It also shows how a high-level operation can be decomposed into a set of elemental ones. Section 7.2 provides some guidelines for setting up the transformation rules. Arrays with certain properties are suggested as useful tools in establishing indexing correspondences. Sections 7.3 and 7.4 describe the transformation rules for the matrix operations and the array manipulation functions respectively.

## 7.1 ANALYSIS OF THE TRANSFORMATION PROBLEM

Elemental assertions define variables which are fields of target data structures. Any assertion that references structured variables has to be transformed into a group of elemental assertions which define and reference fields in those structures. The transformation is guided by the data declarations of the structured variable. The analysis that identifies the field is conducted on two levels: structural and indexing. The structural analysis consists of matching the trees of the referenced structures. It shows which field in a referenced structure is used to define a field in the target structure. The indexing analysis defines the subscript expressions to be used with the referenced variable. The establishment of these two correspondences is discussed in Sections 7.1.1 and 7.1.2, respectively. The reduction of the high-level operation transformation problem to the problem of establishing structural and indexing correspondences is discussed here first.

A high-level operation may be classified as belonging to one, or a combination, of the following classes:

a) reordering or reshaping,

b) field projection,

c) algebraic computation.

Reordering refers to the rearrangement of array elements such as merging or selection. Reshaping refers to changing array dimensionality such as 'flattening' a two-dimensional matrix into a one-dimensional vector. Most of the high-level operations discussed here involve reordering or reshaping. The transformation of operation of this class employs mainly indexing correspondence. Subscript expressions can be used to indicate the relationships between indices of the defined array and those of the referenced arrays. Arrays of integers are frequently used as subscripts for indirect indexing.

Field projection refers to the removal of fields from a structure or the rearrangement of the fields inside the structure. The PROJECTION operation in Relational Algebra belongs to this class. An application example of field projection is the creation of a report of employee names and addresses from a file of employee records. The employee record might contain many other fields besides name and address. In this case, not every source field is selected for output, and the order in which selected fields appear in the report might be different from that in the source record. Field projection is accomplished by structural correspondence, or matching the source structure and the target structure for finding out the correspondence between the fields.

A example of computational operations is matrix inversion, where a set of data (two-dimensional matrix elements) is used as the operand of the operation to produce another defined set of data. This class of operations can be handled more efficiently by computing the high-level operations as integral units. They are transformed into special built-in functions referred to as integral operations. In code generation, integral operations are translated into calls to precoded procedures.

To summarize the above discussion about source-to-source transformation, consider a high-level operation F which defines structured variable B from structured variable A:

```
B=F(A);                                                    (7.1)
```

The assertion can be decomposed into a group of simple ones of the form:

```
FBi(exp1,exp2,...,expn) =
  [IF cond THEN] FAi(exp1',exp2',...,expm');    (7.2)
```

where: (1) FAi and FBi are fields in structures A and B respectively,

(2) m and n are the numbers of dimensions of FAi and FBi, and

(3) The exp's are subscript expressions.

(4) The condition enclosed in brackets may not be required, depending on individual operations.

The task of transforming assertion (7.1) is then equivalent to:

1) Structural correspondence problem: For each field FBi in target structure B, find a corresponding field FAi in source structure A, and
2) Indexing correspondence problem: Use subscript expressions to express the rearrangement of the field occurrences.

The following two subsections further discuss these two problems.

### 7.1.1 STRUCTURE COMPATIBILITY

The compatibility problem between structured variables has been described earlier in Section 3.2, where the high-level operations in MODEL are presented. Given a structured target variable T and a structured source variable S, the compatibility can be established in two ways:

*by name*: for every field t in the struct of T, there is a field s in the structure of S such that t and s share the same name, same data type and same dimensionality.

by structure: for every field t in the structure of T, there is a field s in the structure of S such that t and s share the same sibling position, same dimensionality, and same data type.

If the correspondence exists, T is said to be compatible to S. The total injective mapping from the set of all fields in T to those of S is then used in the transformation of high-level operations of the form:

```
T = S;                                          (7.3)
```

or

```
T = IF cond1 THEN S1
    ELSE IF cond2 THEN S2                       (7.4)
              .
              .
              .
```

or

```
T = F(S)                                        (7.5)
```

For the first two cases (eqs. 7.3 and 7.4), the transformations are simply

```
t = s;
```

or

```
t = IF cond1 THEN s1
    ELSE IF cond2 THEN s2
              .
              .
              .
```

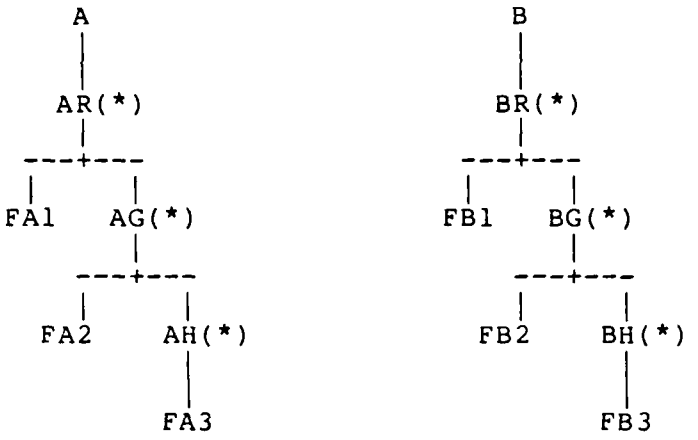
for all the t and s pairs. As to the third case (equation 7.5), the equation T=F(S) is transformed into a set of assertions like equation 7.2 after the indexing correspondence is established.

### 7.1.2 EXPRESSING REORDERING AND RESHAPING

Consider the equation B=F(A) again, with the fields in B and A denoted by FBi and FAi respectively, as in eqs. 7.1 and 7.2. If the argument of the function is an Ns-dimensional array of structures and the result is an Nt-dimensional array of structures (Ns and Nt are usually 1 or 2), since FBi and FAi are the fields corresponding to each other, their numbers of dimensions are the same. Therefore, except for the first Ns dimensions of FAi and the first Nt dimensions of FBi, the rest of the subscript expressions should be the same, i.e., there exists an integer k such that Ns+k=m and Nt+k=n, and assertion (5.1) becomes

```
  FBi(exp1,exp2,...,expNt,I1,I2,...,Ik)
= FAi(exp1',exp2',...,expNs',I1,I2,...,Ik);
```

To illustrate the point, let the structures of the two variables, A and B, be




where A is a one-dimensional (Ns=1) array of record AR and B is a one-dimensional (Nt=1) array of record BR. Let the function be the selection of every even number record in A to form the new array B. We can express the target fields in terms of the source fields as:

```
FB1(I)     = FA1(X(I));
FB2(I,J)   = FA2(X(I),J);
FB3(I,J,K) = FA3(X(I),J,K);
```

with the use of an interim array X, whose value is

```
{2,4,6,8,...}
```

This interim indexing array has to be defined separately as part of the transformation. But assume its existence for this illustration.

The above example shows that as long as a 'suitable' indexing array X is found, the corresponding subscript expressions on the left hand side and the right hand side could be as simple as I vs. X(I). Therefore, aside from the checking of variable compatibility, the problem of transforming an operation on structures can be reduced to the problem of transforming the same operation on elemental data items. E.g., sorting an array of structures can be reduced to sorting an array of integers.

## 7.2 TECHNIQUES FOR ESTABLISHING INDEXING CORRESPONDENCES

Unlike the problem of structural correspondence, which depends only on the structures of the operands, the problem of indexing correspondence depends very much on the nature of individual operations. Some may be straightforward, while others may be very complicated. This section introduces some useful tools in obtaining the desired indexing relationships.

### 7.2.1 SUBSCRIPT MANIPULATION

Subscript manipulation can be used in simple reordering of array elements. For example, the transposition of a matrix A (of elements AF) to obtain a new matrix B (of elements BF) can be written as

```
BF(I,J) = AF(J,I);
```

And the concatenation of two arrays A (of AF) and B (of BF) to form a new array C (of CF) can be written as

```
CF(I) = IF I<=SIZE.A
        THEN AF(I)
        ELSE BF(I-SIZE.A);
```

### 7.2.2 SELECTION ARRAYS

A selection array is an array of 0's and 1's indicating whether the corresponding elements in another array are selected. In many cases, it is easier to convert a given selection condition to a selection array and then reference the later afterwards. The size of the resulting array after the selection can also be obtained by counting the 1's in the selection array.

As an example, let F be a function which selects from array A those elements satisfying the condition 'cond':

```
B=F(A,cond);
```

The selection array C based on the condition 'cond' can be defined as

```
C(I)=IF cond THEN 1 ELSE 0;
```

From this array, the size of the resulting array B can be

expressed simply as the sum of all the elements in C:

```
SIZE.B=SUM(C(I),I);
```

### 7.2.3 SUBLINEAR ARRAYS

As introduced in the Chapter 4, a sublinear array X is an array of integers satisfying the following conditions:

a) monotonically increasing: X(i)>=X(j) for i>j, and
b) increases slower than the indices: X(i)<=i for all i.

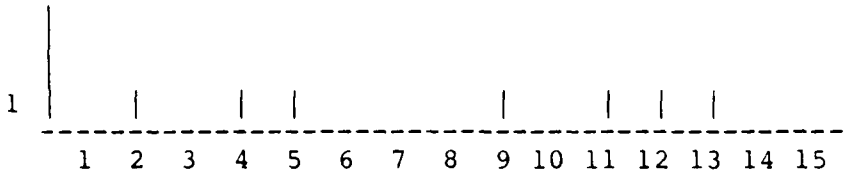
An important thing about sublinear arrays is that it serves as a link between a selection array and the final field definition in the transformation. For example, let C be an selection array, then the sublinear array X can be defined as the integration of C:

```
X(I) = IF I=1
       THEN IF C(I)=1
            THEN 1
            ELSE 0
       ELSE IF C(I)=1
            THEN X(I-1)+1
            ELSE X(I-1);
```

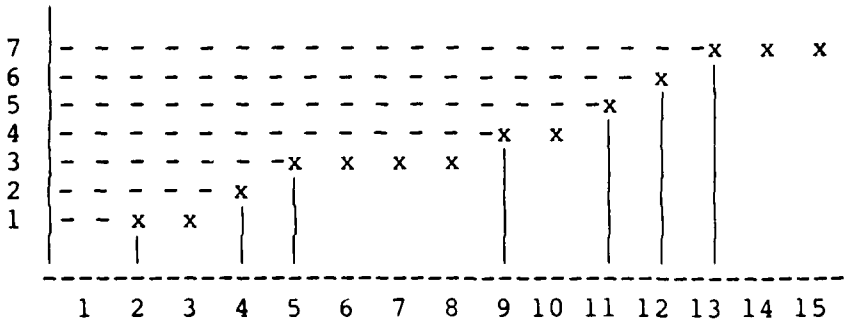
and a target field can be expressed in terms of the corresponding source field with the assertion (using the same variables as in eqs. 7.1 and 7.2):

```
FBi(X(I),I1,I2,...,In)=FAi(I,I1,I2,...,In);
```

Figure 7.1 shows an example of how the sublinear array relates the two sets of indices of the source and target arrays.




selection array : {0,1,0,1,1,0,0,0,1,0,1,1,1,0,0}




sublinear array : {0,1,1,2,3,3,3,3,4,4,5,6,7,7,7}

Figure 7.1 An example of using sublinear arrays

Note that sublinear arrays are usually used in the cases where the resulting array is smaller than that of the source array. In the example shown above, the source array size is 15 and that of the resulting array is 7.

### 7.2.4 SAWTOOTH ARRAYS

When the number of dimensions of the defining and the defined arrays are different, the sublinear array alone can not be used to establish the indexing correspondence. In this case, the sawtooth array can be used together with the sublinear array to accomplish setting up the correspondence. As an example, consider the function COLLECT(B,cond) converting a one-dimensional array B to a two-dimensional array A. If the condition 'cond' is satisfied, a new row of A is formed. To transform the assertion

```
A = COLLECT(B,cond);
```

with the following operand structures:

```
1 A IS GROUP,
  2 AG(*) IS GROUP,
    3 AR(*) IS RECORD,
      4 AF IS FIELD;
```

```
1 B IS GROUP,
  2 BR(*) IS RECORD,
    3 BF IS FIELD;
```

we can use a sublinear array Z1 and a sawtooth array Z2, defined as

```
Z1(I) = IF I=1 THEN 1
        ELSE IF cond THEN Z1(I-1)+1
                     ELSE Z1(I-1);

Z2(I) = IF I=1 THEN Z1(I)
        ELSE IF Z1(I)>Z1(I-1) THEN 1
                              ELSE Z2(I-1)+1;
```

to establish the indexing correspondence as:

```
AF(Z1(I),Z2(I))=BF(I);
```

### 7.2.5 INTEGRAL OPERATIONS

In many cases a desired index array may not be constructed from other arrays easily, especially when the definitions are procedural in nature. These situations can be more efficiently handled with integral operations. As mentioned earlier, an integral operation defines an high-level variable, including arrays, from other variables, with the flexibility provided by the object language. An example of its usage is in sorting a given array and returning the indices as an index array to show the rearrangement needed. Thus, if A is an array of integers:

{8,2,3,2,7}

an integral operation, say SORTN for sorting an array of integers, can be used to construct an index array X:

```
X=SORTN(A);
```

yielding

```
{2,4,3,5,1}
```

The desired sorted array B can then be expressed as

```
B(I)=A(X(I));
```

which is

```
{2,2,3,7,8}.
```

## 7.3 TRANSFORMATION OF MATRIX OPERATIONS

This category of operations includes

1) matrix transposition (denoted by the unary operator |^),
2) matrix multiplication (denoted by binary operator |*),
3) matrix inversion (denoted by the unary operator |/), and
4) elementary arithmetic operations (+, -, * and /).

These operations are grouped together because of the common properties of their operands: rectangular shape and homogeneous components. They can be easily expressed in basic MODEL expressions with a little subscript manipulations.

Consider the following data descriptions:

```
1 A IS RECORD,
  2 AG(*) IS GROUP,
    3 AF(*) IS FIELD;
```

```
1 B IS RECORD,
  2 BG(*) IS GROUP,
    3 BF(*) IS FIELD;

1 C IS RECORD,
  2 CG(*) IS GROUP,
    3 CF(*) IS FIELD;
```

The matrix transposition A=|^B and matrix multiplication A=B|*C are transformed to

```
AF(I,J) = BF(J,I)
```

and

```
AF(I,J) = SUM(BF(I,K)*CF(K,J),K)
```

respectively. For matrix inversion A=|/B, since the computation is much more efficient if implemented as a procedure in the target language, it is transformed into a function assertion A=MATINV(B), which will later be interpreted as procedure call to a matrix inversion subroutine. In case where the inversion is applied to an expression such as

```
A = |/ (B |* C),
```

an interim variable

```
1 T IS GRP,
  2 TG(*) IS GRP,
    3 TF(*) IS FLD;
```

is needed to replace whatever is inside the parentheses:

```
TF(I,J) = SUM(BF(I,K)*CF(K,J),K);
A = MATINV(T);
```

Basic arithmetic operators applied on high-level operands such as

```
C=A*B;
```

are transformed with subscripts to reflect the piecewise application of the operations:

```
CF(I,J)=AF(I,J)*BF(I,J);
```

Since matrix operations can be combined together to form expressions in the same way as that with ordinary basic operations, when transforming an assertion containing compositions of matrix operations, one has to take into account operator precedence and variable dimensionalities. This is done recursively by a procedure called MATRIFY. For each assertion, this procedure checks to see whether it involves matrix operations. If so, expressions containing matrix operators and high-level operands are transformed into another one containing only elemental arithmetical operators and field variables. The procedure has three input parameters:

a) EXP - The expression (in the form of a character string) to be translated.

b) SUB1 - Subscript variable to be used as the subscript of the first (or row) dimension in the translated

4) The expression is a built-in function

5) The expression is a whole assertion

The following subsections describe in detail each of the five cases. Note that at the very top level, MATRIFY accepts the given assertion and returns the transformed one.

### 7.3.1 VARIABLES OR CONSTANTS

There is no transformation for constants. The expression to be returned, TEXP, is exactly the same as the given expression, EXP. And SHAPE is always 'scalar'.

For variables, since they may appear in the assertion with or without subscripts, it is processed based on the way the subscripts are written. Let A be an m-dimensional field variable, and the J's be subscript variables in the given expression EXP.

V1. EXP = A

If m=0, return TEXP=A and SHAPE='scalar'.

If m=1, return TEXP=A(SUB2) and SHAPE='row'.

If m>1, return TEXP=A(SUB1,SUB2) and SHAPE='matrix'.

V2. EXP = A(J1,...,Jm)

Return TEXP=EXP and SHAPE='scalar'.

V3. EXP = A(J1,...,*)

The processing for this class of expression depends on whether OP is |^, |/, or simply a non-matrix operation.

U1. OP = a non-matrix unary operator

Call MATRIFY(EXP1,SUB1,SUB2,TEXP1,SHAPE1)

Return TEXP = OP EXP1, and SHAPE=SHAPE1.

U2. OP = |^, matrix transposition

Call MATRIFY(EXP1,SUB2,SUB1,TEXP1,SHAPE1)

Return TEXP=TEXP1 and

```
SHAPE = IF SHAPE1='scalar' or 'matrix'
        THEN SHAPE1
        ELSE IF SHAPE1='column'
             THEN 'row'
             ELSE 'column'
```

U3. OP = |/, matrix inversion

Call MATRIFY(EXP1,SUB1,SUB2,TEXP1,SHAPE1)

If SHAPE1 is not 'matrix', report an error condition.

If it is an matrix, two auxiliary matrices are declared:

```
G IS GROUP (R(*))
R IS RECORD (AUX(*),INV(*))
(AUX,INV) ARE FIELDS (NUM)
```

Also generate the assertions to define them:

```
AUX(SUB1,SUB2) = TEXP1
INV = MATINV(AUX,SIZE.AUX)
```

Then return TEXP=INV(SUB1,SUB2) and SHAPE='matrix'.

MATINV will be a built-in function for computing the

inverse of a square matrix.

### 7.3.3 BINARY OPERATIONS

Let the given expression be

EXP = EXP1 OP EXP2

where EXP1 and EXP2 are two subexpressions. The processing of this type of expression is also depending on the type of the binary operator OP.

B1. OP = a non-matrix binary operator

```
Call MATRIFY(EXP1,SUB1,SUB2,TEXP1,SHAPE1)

Call MATRIFY(EXP2,SUB1,SUB2,TEXP2,SHAPE2)

Return TEXP = TEXP1 OP TEXP2 and
SHAPE = IF SHAPE1=SHAPE2
        THEN SHAPE1
        ELSE IF SHAPE1='scalar'
             THEN SHAPE2
             ELSE IF SHAPE2='scalar'
                  THEN SHAPE1
                  ELSE 'matrix'
```

B2. OP = |*, matrix multiplication

Let K be a system generated subscript variable.

```
Call MATRIFY(EXP1,SUB1,K,TEXP1,SHAPE1)

Call MATRIFY(EXP1,K,SUB2,TEXP1,SHAPE2)
```

The following cases are error conditions due to uncomformable shapes of the operands:

a) SHAPE1 = 'scalar'

b) SHAPE2 = 'scalar'

c) SHAPE1 = 'column' and SHAPE2 = 'matrix'

d) SHAPE1 = 'matrix' and SHAPE2 = 'row'

e) SHAPE1 = SHAPE2 but not 'matrix'

It no error, return TEXP=SUM(TEXP1*TEXP2,K), and

```
SHAPE = IF SHAPE2='matrix'
        THEN SHAPE1
        ELSE IF SHAPE1='matrix'
             THEN SHAPE2
             ELSE IF SHAPE1='row'
                  THEN 'scalar'
                  ELSE 'matrix'
```

B3. OP = |/, matrix division

As mentioned before the expression EXP1 |/ EXP2 is equivalent to EXP1 |* |/ EXP2. Let E be EXP1 |* |/ EXP2,

Call MATRIFY(E,SUB1,SUB2,TEXP1,SHAPE1)

Return TEXP=TEXP1 and SHAPE=SHAPE1.

### 7.3.4 BUILT-IN FUNCTIONS

The built-in functions here refer to the existing MODEL functions of field variables such as SUM, MAX or MIN. Let EXP be F(ARG1,...,ARGr), where ARGi is the i-th argument to the function F. For each ARGi, i = 1 to r,

Call MATRIFY(ARGi,SUB1,SUB2,TEXPi,SHAPEi)

Return TEXP=F(TEXP1,...,TEXPr). SHAPE is a value depending on F.

### 7.3.5 ASSERTIONS

Let EXP be TARGET=SOURCE, where TARGET is the variable to be defined, and SOURCE is the expression on the right hand side of the assertion, then

```
Call MATRIFY(TARGET,SUB1,SUB2,TEXP1,TSHAPE)
Call MATRIFY(SOURCE,SUB1,SUB2,TEXP2,SSHAPE)
```

If TSHAPE=SSHAPE or SSHAPE='scalar', return TEXP1=TEXP2 as the translated assertion. For other cases where the shapes of the target and the source do not conform to each other, report an error condition.

## 7.4 TRANSFORMATION OF ARRAY MANIPULATION FUNCTIONS

This section describes the transformation for the following array manipulation functions:

SELECT, MERGE, SORT, COLLECT, FUSE,
CONCAT, UNIQUE, UNION, DIFF, PRODUCT

They all have described in detail in Chapter 3. In this section, the transformation rule for each of them are given.

Unless specifically mentioned, array variables used in the presentation are assumed to have general structures as

```
1 A IS GROUP
  2 AR(*) IS GROUP,
    3  .
       .
       .
     AFi IS FIELD;
       .
       .
       . ;

1 B IS GROUP,
  2 AR(*) IS GROUP,
    3  .
       .
       .
     BFi IS FIELD;
       .
       .
       .;

1 C IS GROUP,
  2 CR(*) IS GROUP,
    3  .
       .
       .
     CFi IS FIELD,
       .
       .
       .;
```

Root variable A, B or C refers to the entire array. Field variables such as AFi are used with subscripts for referencing particular field occurrences such as

AFi(I0,I1,...,Ik),

where k is the difference in the numbers of dimensions between AR and AFi. Note that for special cases like arrays of integers, there will be only one field variable in the structure and the value of k is 0.

### 7.4.1 THE SELECT FUNCTION

Let A be an array of structures, SELECT(A,cond) defines an array by selecting those elements from A which satisfy the condition 'cond'.

The transformation of assertions containing the SELECT function such as

```
B=SELECT(A,cond);
```

consists of the following:

First convert the conditional expression to a condition array:

```
D(I)=IF cond THEN 1 ELSE 0;
```

From this condition array, a sublinear array X is defined to express the relationship between the indices of A and those of B:

```
X(I)=IF I=1
     THEN IF D(I)=1
          THEN 1
          ELSE 0
     ELSE IF D(I)=1
          THEN X(I-1)+1
          ELSE X(I-1);
```

Then express the resulting array in terms of source array via indirect indexing:

```
BFi(X(I0),I1,...,Ik)=AFi(I0,I1,...,Ik);
```

And define the size of the resulting array as

```
SIZE.BR=SUM(D(I),I);
```

### 7.4.2 THE MERGE FUNCTION

Given two arrays of structures A and B, MERGE(A,B,cond) defines a new array by interleaving the elements of arrays A and B in an order according to the specified condition 'cond'.

To transform an assertion containing the MERGE function such as

```
C=MERGE(A,B,cond);
```

the following interim variables are used:

SEL : Indicates whether the next element to be selected for C is from A (SEL=1) or B (SEL=0).

ADONE : Indicates whether elements in A have all been selected (ADONE=1).

BDONE : Indicates whether elements in B have all been selected (BDONE=1).

X : A sublinear array for subscripting array A whenever A is selected.

Y : A sublinear array for subscripting array B whenever B is selected.

The assertions tor defining the above arrays are:

```
SEL(I)=ADONE(I)|(^ADONE(I)&cond);

ADONE(I)=IF I=1
         THEN '0'B
         ELSE ADONE(I-1) | (X(I-1)=SIZE.AR & SEL(I-1));

BDONE(I)=IF I=1
         THEN '0'B
         ELSE BDONE(I-1) | (Y(I-1)=SIZE.BR & ^SEL(I-1));

X(I)=IF I=1
     THEN 1
     ELSE IF SEL(I-1) & ^ADONE(I)
          THEN X(I-1)+1
          ELSE X(I-1);

Y(I)=IF I=1
     THEN 1
     ELSE IF SEL(I-1)|BDONE(I)
          THEN Y(I-1)
          ELSE Y(I-1)+1;
```

and the assertions for defining the target array C are:

```
CFi(I0,I1,...,Ik) = IF SEL(I0)
                    THEN AFi(X(I0),I1,...,Ik)
                    ELSE BFi(Y(I0),I1,...,Ik);

SIZE.CR=SIZE.AR+SIZE.BR;
```

### 7.4.3 THE SORT FUNCTION

Let A and B be arrays of structures, the assertion

```
B=SORT(A,key,order);
```

defines array B by sorting array A according to 'key' in ascending (order='ASC') or descending (order='DES') order.

The problem of sorting an array of structures can be reduced to a simpler one of sorting an array of integers or character strings by first defining an array from the given keys and then sort the keys to obtain an index array which shows the sorted order. The actual sorting is performed by a procedure, SORTC (for sort character strings) or SORTN (tor sort numbers), in object language PL/I. They correspond to integral operations at the source level. The transformation is therefore as follows.

First define an array for all keys, assuming they are character strings:

```
EG IS GROUP(E(*));
E IS FIELD(CHAR(10));
E(I)=key(I);
```

and declare an array of integers for the sorted array:

```
XG IS GROUP(X(*));
X IS FIELD(NUM(5));
```

Then use an function assertion to sort the keys:

```
XG=SORTC(EG,order,SIZE.AR);
```

And the sorted array is defined as

```
BFi(I0,I1,...,Ik)=AFi(X(I0),I1,...,Ik);
```

Also define the size of the resulting array:

```
SIZE.BR=SIZE.AR;
```

### 7.4.4 THE COLLECT FUNCTION

The COLLECT function is used to convert a one-dimensional array to a two-dimensional one. The function enables the user to perform some grouping on elements of arrays. Let A be a one-dimensional array of structures and B be a two-dimensional one, then

```
B=COLLECT(A,cond);
```

breaks A in such a way that whenever an element in A satisfies the condition 'cond', a new row is formed with that element as the first one of the new row.

The transformation of the COLLECT function needs two interim indexing arrays X and Y for subscripting B:

```
X(I)=IF I=1
     THEN 1
     ELSE IF 'cond'
          THEN X(I-1)+1
          ELSE X(I-1);

Y(I)=IF I=1
     THEN 1
     ELSE IF 'cond'
          THEN 1
          ELSE Y(I-1)+1;
```

The assertions for defining the target array elements are:

```
BFi(X(I0),Y(I0),I1,...,Ik)=AFi(I0,I1,...,Ik);
```

### 7.4.5 THE FUSE FUNCTION

The FUSE function is the inverse of the COLLECT function in the sense that it defines a one-dimensional array from a two-dimensional one. It has the following format:

```
B=FUSE(A);
```

There is no conditions involved in using the function. The rows in the two-dimensional array A are simply concatenated together to form B.

Two indexing arrays are needed to do the transformation:

```
X(I)=IF I=1
     THEN 1
     ELSE IF Y(I-1)=SIZE.AR(I-1)
          THEN X(I-1)+1
          ELSE X(I-1);

Y(I)=IF I=1
     THEN 1
     ELSE IF Y(I-1)=SIZE.AR(I-1)
          THEN 1
          ELSE Y(I-1)+1;
```

And the new one-dimensional array B is defined as:

```
BFi(I0,I1,...,Ik)=AFi(X(I0),Y(I0),I1,...,Ik);
```

The size of the resulting one-dimensional array is:

```
SIZE.BR=SUM(SIZE.AR(I),I);
```

### 7.4.6 THE CONCAT FUNCTION

Let A and B be arrays of structures, CONCAT(A,B) defines an array C with size equal to the sum of the two, SIZE.AR + SIZE.BR, whose first SIZE.AR elements is taken from A and last SIZE.BR elements from B. The original relative orders of the elements in A and B are unchanged.

The transformation of the CONCAT function is simply the manipulation of subscripts:

```
SIZE.CR=SIZE.AR+SIZE.BR;
```

and

```
CFi(I)=IF I<=SIZE.AR
       THEN AFi(I)
       ELSE BFi(I-SIZE.AR);
```

### 7.4.7 THE UNIQUE FUNCTION

Let A be an array of structures, UNIQUE(A) defines an array containing all elements in A without any duplication.

To transform the assertion

```
B=UNIQUE(A);
```

First, an interim matrix M(I,J) is used to mark whether the I-th and the J-th elements of A is the same:

```
M IS FIELD(NUM1));

M(I,J) = IF I>J
         THEN IF AF1(I)=AF1(J)
              & AF2(I)=AF2(J)
              &      .
                     .
                     .
              & AFn(I)=AFn(J)
             THEN 1
             ELSE 0
         ELSE 0;
```

A vector V is then used to sum up values of the elements in the rows of matrix M:

```
V IS FIELD(NUM(3));
V(I) = SUM(M(I,J),J);
```

Since V(I)=0 means that the I-th element is not a duplicate, the transformation is now equivalent to selecting array A with V(I)=0 as the condition. The rest of the transformation is therefore the same as that for the SELECT function, with Q as the selection array and X as the sublinear indexing array:

```
Q IS FIELD(NUM(2));
Q(I) = IF V(I)=0 THEN 1 ELSE 0;

X IS FIELD(NUM(5));
X(I) = IF I=1
       THEN IF Q(I)^=0
            THEN 1
            ELSE 0
       ELSE IF Q(I)^=0
            THEN X(I-1)+1
            ELSE X(I-1);

WG IS GROUP(IX(*));
```

```
W IS FIELD(NUM(5));

W(X(I)) = IF Q(I)^=0 THEN I;

BF1(J)=AF1(W(J));
BF2(J)=AF2(W(J));
      .
      .
      .
BFn(J)=AFn(W(J));
```

### 7.4.8 THE UNION FUNCTION

The set union of two arrays is obtained by using the UNION function. Let A and B be two such arrays, then UNION(A,B) defines another array C, where every element in C appears either in A or in B.

The composition of CONCAT and UNIQUE is equivalent to UNION. The transformation of UNION(A,B) is therefore the combination of those for CONCAT and UNIQUE:

### 7.4.9 THE DIFF FUNCTION

The DIFF function is used to obtain the set difference between two arrays. Let A and B be two such arrays, then DIFF(A,B) defines a set C where every element in C is contained in A but not in B.

The transformation of the DIFF function, as in the following assertion:

```
C = DIFF(A,B);
```

is similar to the one for the UNIQUE function. First, an interim matrix M(I,J) is used to mark the duplications:

```
M IS FIELD(NUM1));

M(I,J) = IF AF1(I)=BF1(I)
          & AF2(I)=BF2(I)
                .
                .
                .
          & AFn(I)=AFn(J)
         THEN 1
         ELSE 0;
```

And a vector V is used to sum up values of the elements in the rows of the matrix M:

```
V IS FIELD(NUM(3));
V(I) = SUM(M(I,J),J);
```

Then use a selection array D and a sublinear indexing array X to define the selection:

```
D IS FIELD(NUM(2));
D(I) = IF V(I)=0 THEN 1 ELSE 0;

X IS FIELD(NUM(5));
```

```
X(I) = IF I=1
       THEN IF D(I)^=0
            THEN 1
            ELSE 0
       ELSE IF D(I)^=0
            THEN X(I-1)+1
            ELSE X(I-1);

WG IS GROUP(IX(*));
W IS FIELD(NUM(5));

W(X(I)) = IF D(I)^=0 THEN I;

BF1(J)=AF1(W(J));
BF2(J)=AF2(W(J));
      .
      .
      .
BFn(J)=AFn(W(J));
```

### 7.4.10 THE PRODUCT FUNCTION

Let A and B be arrays of records whose constituents are all single fields.

```
1 A IS GROUP,
  2 AR(*) IS GROUP,
    3 A1 IS FIELD,
    3 A2 IS FIELD,
       .
       .
       .
    3 An IS FIELD;
```

```
1 B IS GROUP,
  2 BR(*) IS GROUP,
    3 B1 IS FIELD,
    3 B2 IS FIELD,
       .
       .
       .
    3 Bm IS FIELD;
```

then the Cartesian product of the two arrays PRODUCT(A,B) is an array of the following structure:

```
1 C IS GROUP,
  2 CR(*) IS GROUP,
    3 C1 IS FIELD,
    3 C2 IS FIELD,
       .
       .
       .
    3 Cn IS FIELD,
    3 Cn+1 IS FIELD,
    3 Cn+2 IS FIELD,
       .
       .
       .
    3 Cn+m IS FIELD;
```

whose first n fields are from A and the last m fields are from B. The transformation of PRODUCT(A,B) is accomplished by manipulating the subscripts as follows.

```
C1(I)=A1(1+(I-1)/SIZE.B);
C2(I)=A2(1+(I-1)/SIZE.B);
     .
     .
     .
Cn(I)=An(1+(I-1)/SIZE.B);
```

# CHAPTER 8

# ANALYSIS, CHECKING AND CODE GENERATION

This chapter describes the processing performed for high-level operations in the analysis and code generation phases, subsequent to syntax analysis. During precedence analysis, sawtooth arrays are identified, and subscript expressions involving sawtooth arrays are collected and saved. Edges between structured operands and assertions using integral operations are also created during the precedence analysis phase. These two topics are presented in Sections 8.1 and 8.2. Section 8.3 describes the extension to the range propagation phase, where sawtooth subscript expressions are propagated. Section 8.4 describes the scheduling of different ranges which are related via their sawtooth subscripting relationships. This results in the creation of conditional blocks in the schedule. The handling of the conditional blocks is presented in Section 8.5. Section 8.6 describes the translation of integral

(SWTHREL). The following algorithm describes the recognition by SWTHINX:

Algorithm 8.1 SWTHINX

Given an assertion in internal tree format, determines whether the assertion defines a sawtooth array.

Input: Pointer to the assertion tree.

Output: An entry in the sawtooth array table, if the assertion defines a sawtooth array.

Data structure: Sawtooth array table SWTHREL.

*Method:*

The recognition of a sawtooth array is based on the syntax of the assertion. The following functions, each of which recognizes a small part of the right hand side of equation 8.1, are used:

SWEXP1: The boolean expression I=1.

SWEXP2: The subscript variable Z(I).

SWEXP31: The expression Z(I)^=Z(I-1).

SAME_NUM: The constant 1.

SWEXP33: The expression X(I-1) or X(I-1)+1.

SWEXP3: The IF clause (by calling SWEXP31, SAME_NUM, and SWEXP33):

```
IF Z(I)^=Z(I-1)
THEN 1 ELSE [ X(I-1)+1 |
          IF any_condition
          THEN [X(I-1)+1 | X(I-1)]
          ELSE [X(I-1)+1 | X(I-1)]];
```

The whole recognition process is:
If the right hand side of the equal sign is an IF-THEN-ELSE conditional expression and the IF condition is recognized by SWEXP1, the THEN part by SWEXP2, and the ELSE part by SWEXP3, then enter the X and Z pair in the sawtooth array table SWTHREL.

Table SWTHREL is used in detecting sawtooth expression sequences. Consider a variable, A, subscripted indirectly as:

$$A(Z1(I),Z2(I),\ldots,Zn(I)) \qquad (8.2)$$

If Z1 is sublinear and Zi, for $1<i<=n$, is a sawtooth array based on Zi-1, then Z1,Z2,...,Zn constitute a sawtooth expression sequence. The recognition of a sawtooth expression sequence is actually a process of checking whether all neighboring subscripting array pairs appear in the sawtooth array table SWTHREL. When a sawtooth expression sequence is recognized, it is saved in another table, SWSEQ, as shown in Figure 8.1. Each entry in the table contains the following fields:

NODENUM - The node number of the subscripted variable, i.e., A in equation 8.2.

HEADPOS - The subscript position where the first element of the sequence appears. E.g., n in equation 8.2.

NELEMNT - The length of the sequence. This field is not necessary the same as HEADPOS, since a sawtooth expression may be just part of all the subscript expressions in the parentheses such as A(...,Z1(I),Z2(I),...,Zn(I),...).

INDRARYi - The variable name of the i-th sawtooth array in the sequence.

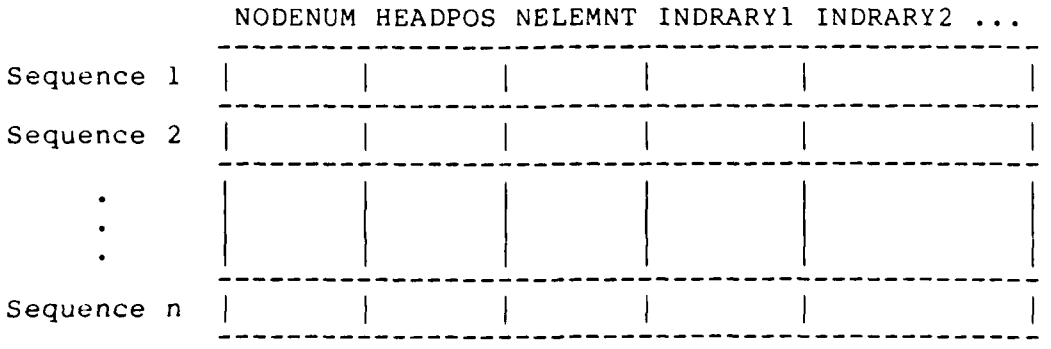



Figure 8.1 Data Structure of the Sawtooth Subscript Expression Sequence Table

The recognition and storing of the sequences are accomplished by procedure STTHSEQ in INDRINX, as described in algorithm 8.2.

Algorithm 8.2 STTHSEQ

Given a subscripted variable, determine whether a sequence of consecutive dimensions are subscripted indirectly by sawtooth expressions.

Input: Pointer to the tree structure representing the variable.

Output: Addition of an entry to the sawtooth expression sequence table.

Data structure: The sawtooth array table SWTHREL and the sawtooth subscript expression sequence table SWSEQ.

Method:

1. From the most significant position, examine the subscript expressions one by one. If a sublinear array is found, mark it as a possible head of a sawtooth expression sequence. If a sublinear array could not be found, return.
2. Starting from the subscript next to the head found in step 1, check each subscript to see whether it is of the

form X(I) and whether X is a sawtooth array defined based on its immediate left neighboring indexing array. If so, append X to the expression sequence. Repeat until the last (the least significant) subscript is examined or until a non-sawtooth array subscript is encountered.

3. If the sequence obtained in step 2 is of length 1, ignore the sequence. Otherwise add the sequence to the sequence table.

## 8.2 CREATING EDGES FOR INTEGRAL OPERATIONS

Assertions using integral operations are in the form of

```
T = P(S1,S2,...Sn);
```

where T is the target variable, P is the name of the integral operation, and the S's are the source variables. These type of assertions are different from other assertions in that their source and target variables are potentially structured, though not necessary so. They are to be recognized by every phase of the processor, from syntax analysis to code generation.

The recognition of the procedure name P is handled the same way as the MODEL built-in functions such as SIN or LOG. The names of procedures to be used are entered in the form of a table (FCNINFO) by program INITIAL of the MODEL

processor. INITIAL also sets some properties of the procedures, such as whether a procedure involves structured variables or whether it is a PL/I built-in function. The information entered into the table by INITIAL not only is used in recognizing integral operations, it is also used in building dependency edges for the assertions, and in generating the corresponding target codes. Besides INITIAL, the names of procedures also have to be entered in the reserved identifier table (in program RESERVED).

In array graph, just like ordinary assertions, an assertion using a integral operation is also represented by a single node. However, in building up edges for the assertion, if a source variable is structured, there will be an edge drawn from every field to the assertion, with dimension difference appropriately set. The same is true for a structured target variable, edges are drawn from the assertion to all fields of the target variable. The creation of these edges are done in procedures SCANRGT and SCANLFT of ENEXDP respectively, the same as the creation of edges for ordinary assertions.

## 8.3 SAWTOOTH EXPRESSION SEQUENCE PROPAGATION

The sawtooth expression sequence associated with a node can be propagated along edges to other nodes in the array graph, if certain conditions are satisfied. To explain this, consider the assertion

```
a1:   A(I,J,K) = B(I,J,K);
```

Since every subscript appears on both sides of the assertion and at the same dimension position, the shapes and sizes of A, B and a1 are identical. If there is a sawtooth expression sequence associated with B, then the same sequence can also be used to enumerate all instances of A. The association of the sequence with A can be derived by propagating the sequence along the edge from B to a1 and the one from a1 to A. More specifically, the conditions for propagating a sawtooth expression sequence along an edge are:

1) the APR_MODE field of every subscript must be 1, i.e., the subscript expressions must be simply I, J or K.
2) the LOCAL_SUB$ fields of all subscripts must be in sequence, e.g., subscripts I,J and K appear on both sides consecutively in the same relative order.

The propagation is done by procedure SSEQPROP, called by the range propagation module RNGPROP after both dimension and range propagations are completed. The following algorithm describes SSEQPROP.

Algorithm 8.3 SSEQPROP

Propagate sawtooth expression sequences along edges.

Input: Sawtooth expression sequence table.

Output: Updated sawtooth expression sequence table.

Method:

1. Starting from the first entry in the sawtooth expression sequence table, get a sequence from the table.
2. Get the list of successor edges (XSUCC_LIST) of the node associated with the sequence.
3. For each successor edge, and for each dimension covered by the sequence, if APR_MODE=1 and LOCAL_SUB$ is greater than the previous one by 1, then the current position of the sequence can be propagated. If any dimension position covered by the sequence can not be propagated, the whole sequence can not be propagated along the successor edge.
4. If the sequence can be propagated along the edge, add a new entry to the sequence table with

NODENUM = the target node.

HEADPOS = the corresponding LOCAL_SUB$ of the edge.

NELEMNT = same as that of the source node.

INDARYi = same as those of the source node.

5. If the sequence can not be propagated along any of the edge on the successor edge list, get next sequence from the sequence table and go to step 2. Terminate it all sequences in the table have been examined.

## 8.4 SCHEDULING FOR CONDITIONAL BLOCKS

In scheduling the array graph, nodes belonging to different ranges can be scheduled in the same loop if they have the same range or subrange of the same range of one of their dimensions. The major range vs. subrange relationship is established from the recognition of sublinear arrays and from the way they are used in indexing. The scheduling a subrange inside the scope of an iteration on its major range requires also the creation of conditional blocks in the scope of the iteration. Since an indexing sawtooth expression always depends on a sublinear array, the conditional blocks need to include the node of an indexing expression. As a matter of fact, a single sublinear array is a special case of sawtooth expression sequence of length 1.

During the scheduling, when a set of nodes are determined as schedulable in the scope of an iteration on a certain major range, it not only means that all node dimensions belonging to the major range can be put in the loop, it also means that those node dimensions belong to any of the subranges (the major range may have more than one subrange) can also be put in the same loop, as long as they are enclosed in a conditional block with a condition which detects the value change of the sublinear array. This can be illustrated using the following assertion, assuming Z1 is sublinear:

```
a1:  A(Z1(I))=B(I);
```

Since Z1 is sublinear, the range of array A is a subrange of that of B. Therefore A can be scheduled inside the loop of the range of B as:

```
Do i=1 to the range of B;
  .
  .
  .
  Define B(i);
  If Z(i)>Z(i-1)
  then define A(Z(i));
  .
  .
  .
end;
```

This can be extended further for cases like:

```
a2:  A(Z1(I),Z2(I),...,Zn(I)) = B(I);
```

where Z1,Z2,...,Zn constitute a sawtooth expression sequence. Because the combined index of the sequence enumerates all instances of A as I increases from 1 to the range of B, the definition of array A can be included in the loop for B as:

```
Do i=1 to the range of B;
  .
  .
  .
  Define B(i);
  It not Zj(i)=Zj(i-1) for all 1<=j<=n
  then define A(Z(i));
  .
  .
  .
end;
```

It is possible to have more than one sequences involved in the above situation, with one of them as a subsequence of others. Consider the following assertions:

```
A(Z1(I),...,Zn(I)) = B(I);
C(Z1(I),...,Zm(I)) = B(I);
```

where 1<=m<n. These two assertions still can be put in the loop of major range B as follows:

```
Do i=1 to the range of B;
   .
   .
   .
   Define B(i);
   If not Zj(i)=Zj(i-1) for all 1<=j<=n
   then Do;
        .
        .
        .
        If not Zj(i)=Zj(i-1) for all 1<=j<=m
        then define C;
        Define A;
        .
        .
        .
        end;
   .
   .
   .
end;
```

Note that there are two places where conditions are used to control whether a node associated with a sawtooth array is to be executed in the generated program. The first place is the entry point to the conditional block. The condition at this point allows the execution of the block only if there is a change in the indexing provided by the sawtooth arrays. Another place for the condition is inside the block, for checking whether the combined index of a subsequence changes. If so, the associated node will then be executed.

## 8.5 CODE GENERATION FOR CONDITIONAL BLOCKS

The main program for code generation is a recursive procedure called GENERATE. When a conditional block (NLMN_TYPE=4) is passed from the scheduler to be translated into PL/I, it generates a conditional block header, calls itself recursively to generate code for the block body, and then generates an END statement to close the block.

The block header is generated by the procedure COND_BLK. It first produces an IF statement to control whether the block is to be executed. The block will be executed only when the value of the combined index changes. Let the sawtooth arrays associated with the block be Z1,Z2,...,Zn, then for each array Zj in the sequence, there is a boolean variable $B_INTERIM.Zj declared in the generated program. The boolean variable is set to TRUE if the index changes, i.e., Zj(I)^=Zj(I-1). The IF statement for detecting changes in the combined index is then

```
IF $B_INTERIM.Z1 |
   $B_INTERIM.Z2 |
         .
         .
         .
   $B_INTERIM.Zn
THEN
  DO;
   .
   . (block body)
   .
  END;
```

In other words, if any index changes ($B_INTERIM.Zj is TRUE), the combined index changes, and the conditional block is executed.

Beside the IF statement described above, the block header also includes the definitions of new subscripts:

```
$X1 = Z1(I);
$X2 = Z2(I);
    .
    .
    .
$Xn = Zn(I);
```

which are to replace all indirect subscripts occurred inside the conditional block.

## 8.6 INTEGRAL OPERATIONS

Integral operations are used as building blocks in setting up the transformation rules for high-level operations. They appear in the array graph and are analyzed exactly the same as assertions. The important capability they provide is the definition of a structured variable using procedures, not just expressions.

In code generation, integral operations are translated into procedure calls. In program GENASSR of the code generation phase, assertions are normally handled by PRINT, which simply rewrites the assertions as assignments in the

target language, with appropriate subscripts inserted. For integral operations, all the source variables and the target variable are written as arguments to the procedure calls. As an example, let matrix C be defined as the inverse of matrix A in the source specification as:

```
C = |/ A;
```

This high-level operation is transformed using the integral operation MATINV as follows:

```
1 GA IS GROUP,
  2 GRA(*) IS GROUP,
    3 AUX(*) IS FIELD (DEC FLOAT);

1 GI IS GROUP,
  2 GRI(*) IS GROUP,
    3 INV(*) IS FIELD (DEC FLOAT);

AUX = A;
GI = MATINV(GA);
C = INV;
```

In code generation, the assertion GI=MATINV(GA) is translated into:

```
CALL MATINV(INV,AUX,ALARM$);

IF ALARM$ THEN PUT SKIP LIST
     ('Inversion failed for matrix ',AUX);
```

As can be seen from the above example, besides translating integral operations into procedure calls, the code generator also generates code for handling abnormal conditions.

The precoded procedures are stored in the MODEL system file UFCNLIB.DAT. In order for the system to recognize these procedures, their names are entered to the system function and procedure table FCNINFO by the initialization program INITIAL.

# CHAPTER 9

# CONCLUSION

## 9.1 SUMMARY

This dissertation presented the source-to-source transformation approach for efficient implementation of high-level operations in nonprocedural programming languages.

Etficiency issues were discussed and justification for achieving efficiency via source-to-source transformation was given. The main idea has been not to treat high-level operands as indivisible entities in implementation, thus allowing global considerations to be applied to individual constituents of high-level structures.

The problem of source-to-source transformation has been analyzed and, as a result, guidelines and tools have been suggested for the decomposition of high-level operations

into elemental ones.

The methodology has been applied in incorporating high-level operations in MODEL. Transformation rules for the provided high-level operations are derived and examples of their use are shown. The implementation of high-level operations in MODEL has demonstrated the feasibility of the methodology.

The contributions of this research can be summarized as follows:

a) A source-to-source transformation scheme for efficient implementation of high-level operations,
b) Operations are made more powerful by allowing more generally structured operands,
c) Special indexing patterns are identified and used for the application of efficient storage allocation schemes,
d) New building blocks for the underlying array graph model are suggested, and
e) The expressive power of the MODEL language is increased substantially through the incorporation of high-level operations.

## 9.2 FUTURE RESEARCH

Several areas can be further investigated to improve or generalize the current results. They include:

a) To make the use of high-level operations more flexible, it is desirable that the user be able to define his own operations, in addition to the ones provided by the system. The immediate question then is how flexible should these operations be. Is the user expected to participate in the source-to-source transformation, or just to define the high-level operations from the system provided ones? There seems to be a conflict between efficiency and flexibility in this area. Since the efficiency considerations built in the transformation vary from operation to operation, more efficient customized transformation rules can always be derived for a given high-level operation.

b) The scheduling and code generation processes described in this dissertation are basically for sequential von Neumann machines. Since array graph is machine type independent, the effect of assuming different types of machines remains to be investigated. Research in this area for data flow machines may have more promising result since all the data flow information is already contained in the array graph and the nature of the data

flow machines allows parallel execution.

c) In allocating storage space for variables in the generated programs, when the use of virtual or window storage allocation scheme is not possible, dynamic storage allocation techniques may be employed to minimize the storage demand of the physical storage allocation scheme. Different variables physically allocated may share a common storage area if they are not referenced at the same time.

# APPENDIX A

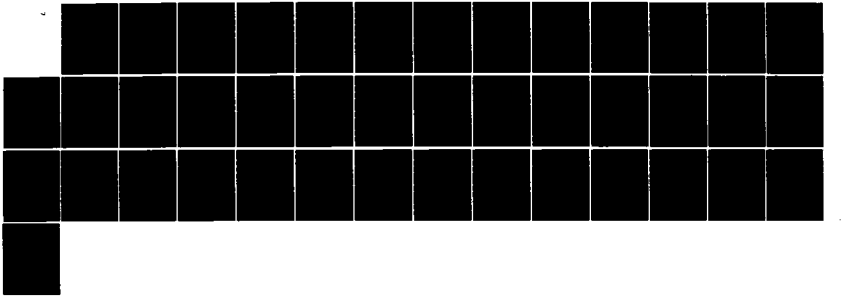
# MODEL EBNF/WSC

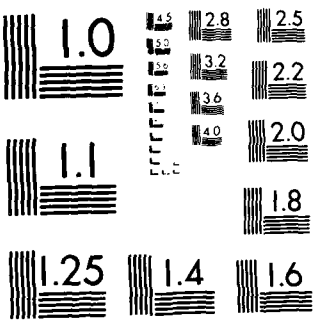
A... [illegible] THE EXTENDED MODEL LANGUAGE

```
<MODEL_SPECIFICATION>:::=[<MODEL_BODY_STMTS> /CLRERRF/ ]*
     /STMT_FL/  <MODEL_SPECIFICATION>

<MODEL_BODY_STMTS>:::=   /E(80)/
   MODULE <MODULE_NAME_STMT>
  |SOURCE <SOURCE_FILES_STMT>
  |TARGET <TARGET_FILES_STMT>
  | @#_END#@ /ENDINP/
  |  <DCL_DESCRIPTION>
  |  <BLOCK_BEGIN>
  |  <BLOCK_END>
  |  <OLD_FILE_STMT>
  |  /ASSINIT/ <ASSERTIONS> /STRHS/

<DCL_DESCRIPTION> ::= 1 /INTDCL/ /INTMVAR/ /MEMINIT/ /SVMEM/
                      <DATA_SPEC>  [, /E(108)/ <INTEGER>
                      /CRDCL/ /INTMVAR/ /MEMINIT/ /SVMEM/
                      <DATA_SPEC> ]* /STDCL/ <ENDCHAR>

<DATA_SPEC>    ::= <DCL_MVAR> [( <OCCSPEC> )] [ <IS> ]
                   <ATTR_SPEC> /SVDCL/

<ATTR_SPEC>    ::= <FILE> /SVF/ /SVFLNM/ <FILE_DESC>
                   <STORAGE_DESC> /STDEV/
                   | <RECORD>                /SVR/
                   | <FIELD_STMT> /STDFLD/ /SVD/
                   | [<GROUP>]               /SVG/

<BLOCK_BEGIN> ::= BLOCK /BLKINIT/ [ <NAME> /SVLBL/ ] /E(2)/
                  : [ <BLOCK_SPEC> ]* /SVBLOK/  <ENDCHAR>

<BLOCK_SPEC>   ::= <SOLUTION> | <ITERATION> | <REL_ERROR>

<SOLUTION>     ::= [ SOLUTION ] METHOD [ <IS> ] /E(62)/
                   <METHODS> /SVMETH/ [ , ]

<METHODS>      ::=  NEWTON | GAUSS_SEIDEL | G_S | JACOBI

<ITERATION>    ::=  [ <MAXIMUM> ] <ITER> [ <IS> ] /E(4)/
                    <NUMBER> /SVITER/ [ , ]

<MAXIMUM>      ::=  MAX | MAXIMUM

<ITER>         ::=  ITER | ITERATION | ITERATIONS
```

AD-A143 470 HIGH-LEVEL OPERATIONS IN NONPROCEDURAL PROGRAMMING LANGUAGES(U) MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA PA DEPT O.. W H LIU DEC 83 2/3

UNCLASSIFIED N00014-83-K-0560 F/G 9/2 NL






MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```
<REL_ERROR>    ::= [ RELATIVE ] <ERROR> [ <IS> ] /E(5)/
                   <NUMBER> /SVERR/ [ , ]

<ERROR>        ::=  ERR |  ERROR

<BLOCK_END> ::= <END> /BLKEND/ [ <NAME> /CHKLBL/ ]
                 <ENDCHAR>

<END>        ::= /ENDID/

<ASSERTIONS>::=/E(14)/<CONDITIONAL> |
               /SVASSR/ /INTMVAR/ <MVAR> /STMVAR/ /SVCMP1/
               [<IS>/SVNXOP/]<DDL_OR_RHS>

<CONDITIONAL>::=IF /SVAAS1/ /SVOP1/ /SETBIT/ /E(18)/
               <BOOLEAN_EXPRESSION> /SVCMP1/ /E(38)/
               THEN /SVNXOP/ <SIMPLE_ASSERTION> /SVNXCMP/
               [ELSE /SVNXOP/ <ASSERTION> /SVNXCMP/]
                /RSETIF/ /STALL/

<ASSERTION>::= /E(14)/ <CONDITIONAL> | <SIMPLE_ASSERTION>

<DDL_OR_RHS>::=/INTODDL/ <DATA_DESC_STMT> /FREETMP/
               | /E(33)/ <INTOAS> <ASSERTION_BRANCH>
                 <ENDCHAR>

<ASSERTION_BRANCH>::= <DEF_EXPRESSION>
                     | <BOOLEAN_EXPRESSION>/SVNXCMP/ /STALL/

<DEF_EXPRESSION>::=  { /INTSUB/ <VALUE_LIST> } /FREESUB/

<VALUE_LIST>::= ( /CRSUB/ /DECPP/ <VALUE_LIST>
                 [, <VALUE_LIST> ]* ) /INCPP/
               | [<SIGN> /SVOPP/] <NUMBER> /STNUM/ /STASS/

<INTOAS>::=/INTOASS/

<SIMPLE_ASSERTION>::=/SVASAE1/ /INTMVAR/ <MVAR> /STMVAR/
                     /CKUNIK/ /SVCMP1/ /E(23)/ = /SVNXOP/
                     <BOOLEAN_EXPRESSION> /SVNXCMP/ /STALL/
                     <ENDCHAR>

<SUB_VARIABLE>::= /SETSUBV/ <VAR> /SVCKMV/ /SVCMP1/
          [(/SVNXOP/ /SETBIT/ /E(22)/
           <EXPRORSTAR> /SVNXCMP/ /SVCKSUB/ [,/SVNXOP/
           <EXPRORSTAR> /SVNXCMP/ /SVCKSUB/ ]*
          /E(24)/ ) ] /CLCKSUB/ /STALL/
```

```
<SUB_VARIABLE1>::= /SETSUBV/ <VAR> /SVCMP1/
          [(/SVNXOP/ /SETBIT/ /E(22)/
           <EXPRORSTAR> /SVNXCMP/ [,/SVNXOP/
           <EXPRORSTAR>/SVNXCMP/]*
          /E(24)/ ) ] /STALL/

<EXPRORSTAR>::= <STARSUB> | <BOOLEAN_EXPRESSION>

<STARSUB>::= /STARSUB/

<BOOLEAN_EXPRESSION>::= /E(82)/ /SVBEXP/ <COND_EXP>
                      | <BOOLEAN_TERM> /SVCMP1/
                    [<OR> /SVNXOP/ <BOOLEAN_TERM> /SVNXCMP/]*
                    /STALL/

<COND_EXP>::=IF /SVCOND/ /E(3)/ <BOOLEAN_EXPRESSION>
         /SVCMP1/ /E(79)/ THEN /SVNXOP/ <BOOLEAN_EXPRESSION>
         /SVNXCMP/ [/E(12)/ ELSE /SVNXOP/
         <BOOLEAN_EXPRESSION> /SVNXCMP/] /STALL/

<OR>::= /OR_REC/

<BOOLEAN_TERM>::= /E(83)/ /SVBT1/ <BOOLEAN_FACTOR> /SVCMP1/
           [& /SVNXOP/ <BOOLEAN_FACTOR> /SVNXCMP/]* /STALL/

<BOOLEAN_FACTOR>::= /E(82)/ /SVBF1/ <CONCATENATION> /SVCMP1/
                    [<RELATION> /SVNXOP/ <CONCATENATION>
                    /SVNXCMP/]* /STALL/

<RELATION>::= /RELREC/

<CONCATENATION>::= /E(84)/ /SVCON/ <ARITH_EXP> /SVCMP1/
                 [ <CONCAT> /SVNXOP/ <ARITH_EXP> /SVNXCMP/]*
                 /STALL/

<CONCAT>::= /CATREC/

<ARITH_EXP>::= /E(81)/ /SVAE/ [<SIGN> /SVOP1/]
          <TERM> /SVCMP1/ [<OPS> /SVNXOP/ <TERM> /SVNXCMP/]*
           /STALL/

<TERM>::= /E(87)/ /SVTERM/ <MFACTOR> /SVCMP1/
           [<MOPS> /SVNXOP/ <MFACTOR> /SVNXCMP/]* /STALL/

<MFACTOR> ::= /SVMFAC/ <FACTOR> /SVCMP1/
            [ <BMOP> /SVNXOP/ <FACTOR> /SVNXCMP/ ]* /STALL/
```

```
<FACTOR>::= <UMOP> /SVMFAC/ /SVOP1/ <PRIMARY> /SVCMP1/
            /STALL/
          | /E(85)/ /SVFAC/ [^ /SVOP1/] <PRIMARY> /SVCMP1/
            [<EXPON> /SVNXOP/ <PRIMARY> /SVNXCMP/]* /STALL/

<BMOP>::= /BMOPREC/

<UMOP>::= /UMOPREC/

<EXPON>::= /EXPREC/

<PRIMARY>::= /E(86)/ /SVPRIM/ <IS_PRIM> /SVCMP1/ /STALL/

<IS_PRIM>::= <IS_PRIM1> | <IS_PRIM2>

<IS_PRIM1>::= ( <BOOLEAN_EXPRESSION> /E(24)/ )
            | SELECT ( <PRIMARY> /SLARG1/ ,
              <BOOLEAN_EXPRESSION> /SLCOND/ ) /SLTRAN/
            | MERGE ( <PRIMARY> /MGARG1/ , <PRIMARY>
              /MGARG2/ , <BOOLEAN_EXPRESSION> /MGCOND/ )
              /MGTRAN/
            | SORT ( <PRIMARY> /SRARG1/ , <PRIMARY> /SRKEY/
                , <PRIMARY> /SRORDER/ )  /SRTRAN/
            | FUSE ( <PRIMARY> /FSARG1/ ) /FSTRAN/
            | COLLECT ( <PRIMARY> /CLARG1/ ,
              <BOOLEAN_EXPRESSION> /CLCOND/ ) /CLTRAN/

<IS_PRIM2>::=  CONCAT ( <PRIMARY> /CTARG1/ , <PRIMARY>
              /CTARG2/ ) /CTTRAN/
            | UNIQUE ( <PRIMARY> /UQARG1/ ) /UQTRAN/
            | UNION ( <PRIMARY> /UNARG1/ , <PRIMARY>
              /UNARG2/ ) /UNTRAN/
            | DIFF ( <PRIMARY> /DFARG1/ , <PRIMARY>
              /DFARG2/ ) /DFTRAN/
            | PRODUCT ( <PRIMARY> /PRARG1/ , <PRIMARY>
              /PRARG2/ ) /PRTRAN/
            | <NUMBER> /STNUM/
            | <STRING_FORM>
            | <FUNCTION_CALL>
            | <SUB_VARIABLE1>

<STRING_FORM>::= ' /SETSTRN/ [ <STRING>  /SVSTRNG/]
                 /E(26)/
                 ' /ADLEX/ [B /STBIT/ /E(1)/ <B_SUFX>]
                 /STNUM/

<FUNCTION_CALL>::= <FUNCTION_NAME> /STFUN/
                   /SETFUNC/ [(/SVNXOP/ <BOOLEAN_EXPRESSION>
                   /SVNXCMP/ [,/SVNXOP/ <BOOLEAN_EXPRESSION>
                   /SVNXCMP/ ]* ) ] /STALL/
```

```
<FUNCTION_NAME>::= /FNCHECK/

<MVAR>::=  ( <SUB_VARIABLE> /SVMVAR/
           [, <SUB_VARIABLE> /SVMVAR/ ]* )
         | <SUB_VARIABLE> /SVMVAR/

<VAR>::= /SETVAR/ /INITQNM/ /E(68)/ <NAME> /ADLEX/ /MKQNM/
       [. /ADLEX/ /E(68)/ <NAME> /ADLEX/ /MKQNM/]* /STR_CON/

<DCL_MVAR> ::= ( <VAR> /SVCKMV/ /SVMVAR/ [, <VAR> /SVCKMV/
                 /SVMVAR/ ]* )
             | <VAR> /SVCKMV/ /SVMVAR/

<B_SUFX>::= /BITSTR/

<QNAME>::= /INITQNM/ /E(68)/ <NAME>  /MKQNM/
    [ .  /E(68)/ <NAME>  /MKQNM/ ] *

<STRING>::= <STRING_CONST>

<OPS>::= /OPREC/

<MOPS>::= /MOPREC/

<TEST>::= /TESTBIT/

<MODULE_NAME_STMT>::= /E(63)/: /E(64)/ <NAME>
    /STMOD/ <ENDCHAR>

<SOURCE_FILES_STMT>::= [<FILE_KEYWORD>] /E(75)/ /INITSFL/ :
    <SOURCE_FILELIST> /STSRC/ <ENDCHAR>

<FILE_KEYWORD>::=FILES|FILE

<SOURCE_FILELIST>::= /E(76)/ <NAME> /SVSRC/
     [, /E(76)/ <NAME> /SVSRC/]*

<TARGET_FILES_STMT>::= [<FILE_KEYWORD>] /E(77)/ /INITTFL/ :
    <TARGET_FILELIST>    /STTAR/ <ENDCHAR>

<TARGET_FILELIST>::= /E(78)/ <NAME> /SVTAR/
    [, /E(78)/ <NAME> /SVTAR/ ]*

<DATA_DESC_STMT>::=  <DATA_DESCRIPTION> <ENDCHAR>
```

```
<DATA_DESCRIPTION>::=
  <FILE_STMT>     /STFILE/
  |<RECORD_STMT> /STREC/
  |<GROUP_STMT>  /STGRP/
  |<FIELD_STMT> /STFLD/
  |<SUB_STMT>    /STSUBST/

<SUB_STMT>::=<SUBSCRIPT>/MEMINIT/ /SVMEM/ [( <OCCSPEC> )]

<SUBSCRIPT>::= SUB | SUBSCRIPT | SUBSCRIPTS

<FILE>::= FILE | REPORT | FILES | REPORTS

<RECORD_STMT>::= <RECORD> /MEMINIT/ [(] <ITEM_LIST> [)]

<RECORD>  ::=  REC | RECORD | RECORDS

<ITEM_LIST>::= /E(52)/<ITEM> [[,] <ITEM>]*

<ITEM>::=<NAME> /SVMEM / [ . <NAME> /SVMEM/ ]*
          [( <OCCSPEC> )]

<OCCSPEC>::=  <STAR> /SVSTAR/ | <MINOCC>/SVMNOC/ [<MAXOCC>]

<STAR>::= /STARREC/

<MINOCC>::=<INTEGER>

<MAXOCC>  ::=  [:/E(51)/]<INTEGER> /SVMXOC/ /CKMNMX/
             |  <INTEGER> /SVMXOC/ /CKMNMX/

<GROUP_STMT>::= <GROUP>/MEMINIT/ [(] <ITEM_LIST> [)]

<GROUP>  ::=  GRP | GROUP | GROUPS

<FIELD_STMT>::=  <FIELD> /SVFLD/ <FIELD_ATTR>
                [<ON_CND> : <OPT> /SVOP/ ]

<ON_CND> ::= ON_CNVERR | ON_CERR

<OPT>    ::= STOP | <NUMBER>

<FIELD>  ::=  FLD | FIELD | FIELDS

<FIELD_ATTR>::= [(]  <TYPE> /SVFDTP2/[ <LENG_SPEC>]
                [,]  [<LINE_SPEC>]  [,]  [<COL_SPEC>]
                [)]
```

```
<LENG_SPEC> ::= ( /E(48)/ <MIN_LENGTH> [ <MAX_LENGTH> ]
                  /E(49)/ )
                |<MIN_LENGTH> [<MAX_LENGTH>]

<MIN_LENGTH>::= <INTEGER> /SVMNFLN/

<LINE_SPEC>::= LINE /E(53)/ /E(54)/ /E(55)/ (<INTEGER>
               /SVLINE/)

<COL_SPEC>::= COL /E(90)/ /E(91)/ /E(92)/ (<INTEGER>
              /SVCOL/)

<TYPE>::= /E(47)/ <PIC_DESC> | <STRING_SPEC>  | <NUM_SPEC>

<PIC_DESC>::= <PIC_TYPE> /E(67)/ /SVPIC/ '
              [ <STRING> /SVPICST/ ] ' /STPIC/

<PIC_TYPE>::= PIC | PICTURE

<STRING_SPEC>::= <STRING_TYPE> /SVSTRTP/

<STRING_TYPE>::= CHAR | CHARACTER | BIT | NUM | NUMERIC

<NUM_SPEC>::= <NUM_TYPE> /SVNUMTP/ [ <FIXFLT> /SVMOD/ ]

<NUM_TYPE>::= BIN | BINARY | DEC | DECIMAL

<FIXFLT>::= FIX | FIXED | FL | FLOAT | FLT

<MAX_LENGTH>::= [:] <INTEGER> /SVMXFLN/
                | , /E(46)/ <SINTGR> /SVSCALE/
                | <INTEGER> /SVMXFLN/

<SINTGR>::= - /E(50)/ <INTEGER> /NEGATE/ | <INTEGER>

<NUMBER> ::= /SETNUM/ <INITNUM> /E(65)/ <RECNUM>

<RECNUM>::= /RECNUM/

<INITNUM>::= /INITNUM/

<SIGN>::= + | -

<RECG>::= <RECORD> | <GROUP>

<KEY>::=KEY|SEQUENCE

<CODE>::=EBCDIC|BCD|ASCII

<ANY>::= <NAME>|<INTEGER>
```

```
<NO_TRKS>::= 7|9

<DENSITY>::= 200|556|800|1600|6250

<PARITY>::= ODD|EVEN

<TYPEDSK>::= 2314|2311|3330|2305 | 3330-1

<ORG>::=ORG|ORGANIZATION

<ORG_TYPE>::= /E(7)/ISAM|SEQUENTIAL|SAM|INDEXED_SEQUENTIAL

<ENDCHAR>::= /E(74)/ <END_CHAR> /STMTINC/

<END_CHAR>::= /SVENDC/

<STRING_CONST>::=/CHARSTR/

<NAME>::=/NAMEREC/

<INTEGER>::=/INTREC/

<IS>::= IS | = | ARE

<FILE_STMT>::=  <FILE> /SVFLNM/ /MEMINIT/ <SON_DESC>
                <FILE_DESC> <STORAGE_DESC> /STDEV/

<SON_DESC>::=( <ITEM_LIST> )
             | <RECG> [NAME] [<IS>] [(] <ITEM> [)]

<OLD_FILE_STMT>::= <FILE> [NAME] [<IS>] /E(56)/ /MEMINIT/
                    /INTMVAR/
                   <DCL_MVAR> /SVFLNM/
                   <RECG> [NAME] [<IS>] [(] <ITEM> [)]
                   <FILE_DESC> /STFILE/
                   <STORAGE_DESC> /STDEV/  <ENDCHAR>

<FILE_DESC>::=   [STORAGE [NAME] [<IS>] /E(44)/ <NAME>
                  /SVSTNM/]
                 [<KEY> [NAME] [<IS>] /E(45)/ <NAME>
                   /SVKEY/]
                 [<ORG> [<IS>] <ORG_TYPE> /SVORG3/]

<STORAGE_DESC>  ::= [DEVICE [<IS>] <DEVICE>] /SVDEV/
        [RECORD /E(57)/][FORMAT [<IS>] <REC_FMT>]/SVRECF/
        <BLK_REC_VOL>
        [<TAPE_DESC>] [<DISK_DESC>]
        [HARDWARE]  [SOFTWARE]
```

```
<DEVICE>  ::= /E(61)/ TAPE  | DISK/SETDEVB/
      |  CARD /SETDEVC/     | PRINTER /SETDEVP/
      | PUNCH /SETDEVU/     | TERMINAL /SETDEVT/

<REC_FMT>  ::= /E(69)/ FIXED|VARIABLE|VAR_SPANNED|UNDEFINED

<BLK_REC_VOL>  ::=
        [ [MAX] /E(70)/ /E(71)/ BLOCKSIZE [<IS>] <INTEGER>
          /SVBLK/ ]
        [ [MAX/E(59)/] RECORDSIZE [<IS>] /E(72)/<INTEGER>
          /SVRCSZ/]
        [VOLUME [NAME] [<IS>] /E(60)/ <NAME>/SVVOL/
          [,/E(60)/<NAME>]* ]

<TAPE_DESC>  ::= [<TRACKS> [<IS>] /E(66)/<NO_TRKS>/SVTRK2/]
          [PARITY [<IS>] /E(66)/ <PARITY>/SVPAR2/]
          [DENSITY [<IS>] /E(66)/ <DENSITY> /SVDEN2/]
          [ [TAPE] LABEL [<IS>] <LABEL_TYPE>/SVLAB2/]
          [START [FILE] [<IS>] /E(66)/ <INTEGER> /SVSTFL2/]
          [[CHAR] CODE [<IS>] <CODE> /SVCC/ ]

<TRACKS>  ::=  NO_TRKS | TRACKS

<LABEL_TYPE>  ::= /E(58)/ IBM_STD|ANSI_STD|NONE|BYPASS

<DISK_DESC>  ::=  [UNIT [<IS>] /E(9)/ <TYPEDSK> /SVUNIT2/]
   [<CYLINDERS>/SVUCYL/ [<IS>] /E(66)/ <INTEGER> /SVQTY2/]

<CYLINDERS>  ::=  NO_CYLS | CYLINDERS

<HARDWARE>::= [[COMPUTER] MODEL [<IS>] <ANY>

<SOFTWARE>::= [[OPERATING] SYSTEM [<IS>] <ANY>]
```

## A.2 EBNF/WSC FOR THE PREPROCESSOR

```
<MODEL_SPECIFICATION>::=
     [<MODEL_BODY_STMTS> /STMT_FL/ /CLRERRF/ ]*
     /STMT_FL/  <MODEL_SPECIFICATION>

<MODEL_BODY_STMTS>::=  MODULE <MODULE_NAME_STMT>
                     | SOURCE <SOURCE_FILES_STMT>
                     | TARGET <TARGET_FILES_STMT>
                     | @#_END#@ /ENDPASS/
                     | <DCL_DESCRIPTION>
                     | <OLD_FILE_STMT>
                     | <ASSERTIONS>

<DCL_DESCRIPTION> ::= 1 /LEVEL_1/ <DATA_SPEC> /SVL_F1/
         [, <INTEGER> /LEVEL_N/ <DATA_SPEC> /SVL_F1/]*

<DATA_SPEC>    ::= <DCL_MVAR> [( <OCCSPEC> )] [ <IS> ]
                     <ATTR_SPEC>

<ATTR_SPEC>    ::= <FILE> /SETFLE/ <FILE_DESC>
                   <STORAGE_DESC>
                  | <RECORD> /SETREC/
                  | <FIELD_STMT> /SETFLD/
                  | [ <GROUP> ] /SETGRP/

<ASSERTIONS>::=  <MVAR> /SBOUND/ [<IS>] <DDL_OR_RHS>
                 /SVL_F2/

<DDL_OR_RHS>::= <DATA_DESC_STMT>

<SUB_VARIABLE>::= <VAR>

<SUB_VARIABLE1>::= <VAR>

<MVAR>::=  ( <SUB_VARIABLE> /SAVV1/
           [, <SUB_VARIABLE> /SAVVN/ ]* )
         | <SUB_VARIABLE> /SAVV1/

<VAR>::=  /S_INIT/ <NAME> /S_SET/ [. <NAME> /S_CON/]*

<DCL_MVAR> ::= ( <VAR> /SAVV1/ [, <VAR> /SAVVN/ ]* )
             | <VAR> /SAVV1/

<STRING>::= <STRING_CONST>

<MODULE_NAME_STMT>::=  : <NAME> <ENDCHAR>
```

```
<SOURCE_FILES_STMT>::=  [<FILE_KEYWORD>] :
                        <SOURCE_FILELIST> <ENDCHAR>

<FILE_KEYWORD>::=FILES|FILE

<SOURCE_FILELIST>::=  <NAME> [, <NAME> ]*

<TARGET_FILES_STMT>::= [<FILE_KEYWORD>] : <TARGET_FILELIST>
                       <ENDCHAR>

<TARGET_FILELIST>::=  <NAME> [, <NAME> ]*

<DATA_DESC_STMT>::=  <DATA_DESCRIPTION> <ENDCHAR>

<DATA_DESCRIPTION>::=
  <FILE_STMT>    /SETFLE/
  |<RECORD_STMT> /SETREC/
  |<GROUP_STMT>  /SETGRP/
  |<FIELD_STMT>  /SETFLD/

<FILE>::= FILE | REPORT | FILES | REPORTS

<RECORD_STMT>::= <RECORD> [(] <ITEM_LIST> [)]

<RECORD>  ::=  REC | RECORD | RECORDS

<ITEM_LIST>::= <ITEM> [[,] <ITEM>]*

<ITEM>::= /S_INIT/ <NAME> /S_SET/ [ . <NAME> /S_CON/ ]*
          [( <OCCSPEC> )] /SAVVN/

<OCCSPEC>::=  <STAR> /OC_STAR/
            | <MINOCC> /OC_MIMA/ [<MAXOCC> /OC_MIMA/ ]

<STAR>::= /STARREC/

<MINOCC>::=<INTEGER>

<MAXOCC>  ::=  [:]<INTEGER>  | <INTEGER>

<GROUP_STMT>::= <GROUP> [(] <ITEM_LIST> [)]

<GROUP>  ::=  GRP | GROUP | GROUPS

<FIELD_STMT>::=  <FIELD> <FIELD_ATTR> [<ON_CND> : <OPT> ]

<ON_CND> ::= ON_CNVERR | ON_CERR

<OPT>    ::= STOP | <NUMBER>
```

```
<FIELD>  ::=  FLD | FIELD | FIELDS

<FIELD_ATTR>::= [(]  <TYPE> [ <LENG_SPEC>]
                [,]  [<LINE_SPEC>]  [,]  [<COL_SPEC>]  [)]

<LENG_SPEC> ::= ( <MIN_LENGTH> [ <MAX_LENGTH> ] )
                |<MIN_LENGTH> [<MAX_LENGTH>]

<MIN_LENGTH>::= <INTEGER>

<LINE_SPEC>::= LINE (<INTEGER> )

<COL_SPEC>::= COL (<INTEGER> )

<TYPE>::=  <PIC_DESC> | <STRING_SPEC>  | <NUM_SPEC>

<PIC_DESC>::= <PIC_TYPE> ' [ <STRING> ] '

<PIC_TYPE>::= PIC | PICTURE

<STRING_SPEC>::= <STRING_TYPE>

<STRING_TYPE>::= CHAR | CHARACTER | BIT | NUM | NUMERIC

<NUM_SPEC>::= <NUM_TYPE> [ <FIXFLT> ]

<NUM_TYPE>::= BIN | BINARY | DEC | DECIMAL

<FIXFLT>::= FIX | FIXED | FL | FLOAT | FLT

<MAX_LENGTH>::= [:] <INTEGER>  | , <SINTGR>  | <INTEGER>

<SINTGR>::= - <INTEGER>  | <INTEGER>

<NUMBER> ::= /SETNUM/ <INITNUM> <RECNUM>

<RECNUM>::= /RECNUM/

<INITNUM>::= /INITNUM/

<SIGN>::= + | -

<RECG>::= <RECORD> | <GROUP>

<KEY>::=KEY|SEQUENCE

<CODE>::=EBCDIC|BCD|ASCII

<ANY>::= <NAME>|<INTEGER>
```

```
<NO_TRKS>::= 7|9

<DENSITY>::= 200|556|800|1600|6250

<PARITY>::= ODD|EVEN

<TYPEDSK>::= 2314|2311|3330|2305 | 3330-1

<ORG>::=ORG|ORGANIZATION

<ORG_TYPE>::= ISAM|SEQUENTIAL|SAM|INDEXED_SEQUENTIAL

<ENDCHAR>::=  <END_CHAR>

<END_CHAR>::= /SVENDC/

<STRING_CONST>::=/CHARSTR/

<NAME>::=/NAMEREC/

<INTEGER>::=/INTREC/

<IS>::= IS | = | ARE

<FILE_STMT>::=  <FILE> <SON_DESC>
                <FILE_DESC> <STORAGE_DESC>

<SON_DESC>::=( <ITEM_LIST> )
            | <RECG> [NAME] [<IS>] [(] <ITEM> [)]

<OLD_FILE_STMT>::= <FILE> /SETFLE/ [NAME] [<IS>]
                   <DCL_MVAR> /SBOUND/
                   <RECG> [NAME] [<IS>] [(] <ITEM> [)]
                   /SVL_F2/
                   <FILE_DESC>
                   <STORAGE_DESC> <ENDCHAR>

<FILE_DESC>::=   [STORAGE [NAME] [<IS>] <NAME>]
                 [<KEY> [NAME] [<IS>] <NAME> ]
                 [<ORG> [<IS>] <ORG_TYPE> ]

<STORAGE_DESC>  ::= [DEVICE [<IS>] <DEVICE>]
         [RECORD][FORMAT [<IS>] <REC_FMT>]
         <BLK_REC_VOL>
         [<TAPE_DESC>] [<DISK_DESC>]
         [HARDWARE]  [SOFTWARE]

<DEVICE>  ::=  TAPE | DISK |  CARD | PRINTER | PUNCH |
               TERMINAL
```

```
<REC_FMT>  ::=  FIXED|VARIABLE|VAR_SPANNED|UNDEFINED

<BLK_REC_VOL>  ::=
         [ [MAX] BLOCKSIZE [<IS>] <INTEGER> ]
         [ [MAX] RECORDSIZE [<IS>] <INTEGER>]
      [VOLUME [NAME] [<IS>] <NAME> [,<NAME>]* ]

<TAPE_DESC>  ::=  [<TRACKS> [<IS>] <NO_TRKS> ]
                  [PARITY [<IS>]  <PARITY>]
                  [DENSITY [<IS>]  <DENSITY> ]
                  [ [TAPE] LABEL [<IS>] <LABEL_TYPE>]
                  [START [FILE] [<IS>]  <INTEGER> ]
                  [[CHAR] CODE [<IS>] <CODE> ]

<TRACKS>  ::=  NO_TRKS | TRACKS

<LABEL_TYPE>  ::=  IBM_STD|ANSI_STD|NONE|BYPASS

<DISK_DESC>  ::=  [UNIT [<IS>]  <TYPEDSK>]
                  [<CYLINDERS> [<IS>]  <INTEGER>]

<CYLINDERS>  ::=  NO_CYLS | CYLINDERS

<HARDWARE>::= [[COMPUTER] MODEL [<IS>] <ANY>

<SOFTWARE>::= [[OPERATING] SYSTEM [<IS>] <ANY>]
```

# APPENDIX B

## EXAMPLES OF TRANSFORMATIONS

## B.1 MATRIX OPERATIONS

```
 1      MODULE: MINVSE;
 2      SOURCE: AFILE;
 3      TARGET: CFILE;
 4
 5      1 AFILE IS FILE,
 6        2 A IS RECORD,
 7          3 AG(3) IS GROUP,
 8            4 AF(3) IS FIELD (PIC'S99V99');
 9
10      1 B IS GROUP,
11        2 BG(3) IS GROUP,
12            4 BF(3) IS FIELD (DEC FLOAT);
13
14      1 CFILE IS FILE,
15        2 C(5) IS RECORD,
16          3 CG(3) IS GROUP,
17            4 CF(3) IS FIELD (PIC'SZZ9.V99');
18
19  *** BF={((1,2,3),(2,3,4),(3,2,1))};
20      BF (1 ,1 )=1 ;
21      BF (1 ,2 )=2 ;
22      BF (1 ,3 )=3 ;
23      BF (2 ,1 )=2 ;
24      BF (2 ,2 )=3 ;
25      BF (2 ,3 )=4 ;
26      BF (3 ,1 )=3 ;
27      BF (3 ,2 )=2 ;
28      BF (3 ,3 )=1 ;
29
30  *** C(1) = A |* B ;
31      (ROW$1,COL$1) ARE SUBSCRIPTS;
32      SUM$1 IS FIELD(DECIMAL FLOAT);
33      SUM$1(ROW$1,COL$1)=
          SUM(AF (ROW$1 ,SUB1 )*BF (SUB1 ,COL$1 ),SUB1);
34      CF (1 ,ROW$1 ,COL$1 )=SUM$1 (ROW$1 ,COL$1 );
35  *** C(2) = |^ A;
36      (ROW$2,COL$2) ARE SUBSCRIPTS;
37      CF (2 ,ROW$2 ,COL$2 )=AF (COL$2 ,ROW$2 );
38  *** C(3) = |/A;
39      (ROW$3,COL$3) ARE SUBSCRIPTS;
40      (AUX$1,INV$1) ARE FIELDS(DECIMAL FLOAT);
41      GRA$1 IS GROUP (AUX$1(*));
42      GRI$1 IS GROUP (INV$1(*));
43      GA$1 IS GROUP (GRA$1(*));
```

## B.2 THE SELECT FUNCTION

```
     MODULE: SEL;
     SOURCE: FB;
     TARGET: FA;

     1 FB IS FILE,
       2 B(30) IS RECORD,
         3 B1 IS FIELD(CHAR(3)),
         3 B2(3) IS GROUP,
           4 B21(2) IS GROUP,
             5 B211 IS FIELD(CHAR(5)),
             5 B212 IS FIELD(CHAR(5)),
           4 B22 IS FIELD(CHAR(4));

     1 FA IS FILE,
       2 A(*) IS RECORD,
         3 A1 IS FIELD(CHAR(3)),
         3 A2(3) IS GROUP,
           4 A21(2) IS GROUP,
             5 A211 IS FIELD(CHAR(5)),
             5 A212 IS FIELD(CHAR(5)),
           4 A22 IS FIELD(CHAR(4));

     SIZE.B = 30;

*** FA=SELECT(FB,B1='XXX');

     L1$C IS FIELD(NUM(2));
     L1$C = IF B1='XXX' THEN 1 ELSE 0;
     L1$X IS FIELD(NUM(5));
     L1$X(SUB1)=IF SUB1=1
          THEN IF L1$C(SUB1)^=0
               THEN 1
               ELSE 0
          ELSE IF L1$C(SUB1)^=0
               THEN L1$X(SUB1-1)+1
               ELSE L1$X(SUB1-1);
     L1$SZ IS FIELD(NUM(5));
     L1$SZ=SUM(L1$C(SUB1),SUB1);
     L1$IG IS GROUP(L1$I(*));
     L1$I IS FIELD(NUM(5));
     L1$I(L1$X(SUB1))=IF L1$C(SUB1)^=0 THEN SUB1;
     SIZE.L1$I=L1$SZ;
     A1(SUB1)=B1(L1$I(SUB1));
     A211(SUB1,SUB2,SUB3)=B211(L1$I(SUB1),SUB2,SUB3);
     A212(SUB1,SUB2,SUB3)=B212(L1$I(SUB1),SUB2,SUB3);
     A22(SUB1,SUB2)=B22(L1$I(SUB1),SUB2);
```

## B.3 THE MERGE FUNCTION

```
      MODULE: MERGE;
      SOURCE: MA,MB;
      TARGET: M;

      1 MA IS FILE,
        2 MAREC(*) IS RECORD,
          3 MA1 IS FIELD(CHAR(3)),
          3 MA2(3) IS GROUP,
            4 MA21(2) IS GROUP,
              5 MA211 IS FIELD(CHAR(5)),
              5 MA212 IS FIELD(CHAR(5)),
            4 MA22 IS FIELD(CHAR(4));

      1 MB IS FILE,
        2 MBREC(*) IS RECORD,
          3 MB1 IS FIELD(CHAR(3)),
          3 MB2(3) IS GROUP,
            4 MB21(2) IS GROUP,
              5 MB211 IS FIELD(CHAR(5)),
              5 MB212 IS FIELD(CHAR(5)),
            4 MB22 IS FIELD(CHAR(4));

      1 M IS FILE,
        2 MREC(*) IS RECORD,
          3 M1 IS FIELD(CHAR(3)),
          3 M2(3) IS GROUP,
            4 M21(2) IS GROUP,
              5 M211 IS FIELD(CHAR(5)),
              5 M212 IS FIELD(CHAR(5)),
            4 M22 IS FIELD(CHAR(4));

      SIZE.MAREC=10;
      SIZE.MBREC=15;

***   M=MERGE(MA,MB,MA1<MB1);
      M1$X IS FIELD(NUM(5));
      M1$Y IS FIELD(NUM(5));
      M1$X(SUB1)=IF SUB1=1 THEN 1 ELSE
           IF M1$S(SUB1-1) & ^M1$D(SUB1)
           THEN M1$X(SUB1-1)+1
           ELSE M1$X(SUB1-1);
```

```
42      M1$Y(SUB1)=IF SUB1=1 THEN 1 ELSE
43           IF M1$S(SUB1-1) | M1$E(SUB1)
44           THEN M1$Y(SUB1-1)
45           ELSE M1$Y(SUB1-1)+1;
46      M1$D IS FIELD(BIT(1));
47      M1$E IS FIELD(BIT(1));
48      M1$D(SUB1)=IF SUB1=1 THEN '0'B
49           ELSE M1$D(SUB1-1) |
50           (M1$X(SUB1-1)=SIZE.MAREC & M1$S(SUB1-1));
51      M1$E(SUB1)=IF SUB1=1 THEN '0'B
52           ELSE M1$E(SUB1-1) |
53           (M1$Y(SUB1-1)=SIZE.MBREC & ^M1$S(SUB1-1));
54      M1$S IS FIELD(BIT(1));
55      M1$S(SUB1)=M1$E(SUB1) | ( ^M1$D(SUB1) &
56           (MA1(M1$X(SUB1)) < MB1(M1$Y(SUB1))));
57      M1$SZ IS FIELD(NUM(5));
58      M1$SZ=SIZE.MAREC+SIZE.MBREC;
59      SIZE.MREC=M1$SZ;
60      M1(SUB1)=IF M1$S(SUB1)
61           THEN MA1(M1$X(SUB1))
62           ELSE MB1(M1$Y(SUB1));
63      M211(SUB1,SUB2,SUB3)=IF M1$S(SUB1)
64           THEN MA211(M1$X(SUB1),SUB2,SUB3)
65           ELSE MB211(M1$Y(SUB1),SUB2,SUB3);
66      M212(SUB1,SUB2,SUB3)=IF M1$S(SUB1)
67           THEN MA212(M1$X(SUB1),SUB2,SUB3)
68           ELSE MB212(M1$Y(SUB1),SUB2,SUB3);
69      M22(SUB1,SUB2)=IF M1$S(SUB1)
70           THEN MA22(M1$X(SUB1),SUB2)
71           ELSE MB22(M1$Y(SUB1),SUB2);
```

## B.4 THE SORT FUNCTION

```
 1      MODULE: SORTREC;
 2      SOURCE: SB;
 3      TARGET: SA;
 4
 5      1 SB IS FILE,
 6        2 B(30) IS RECORD,
 7          3 B1 IS FIELD(CHAR(3)),
 8          3 B2(3) IS GROUP,
 9            4 B21(2) IS GROUP,
10              5 B211 IS FIELD(CHAR(5)),
11              5 B212 IS FIELD(CHAR(5)),
12            4 B22 IS FIELD(CHAR(4));
13
14      1 SA IS FILE,
15        2 A(*) IS RECORD,
16          3 A1 IS FIELD(CHAR(3)),
17          3 A2(3) IS GROUP,
18            4 A21(2) IS GROUP,
19              5 A211 IS FIELD(CHAR(5)),
20              5 A212 IS FIELD(CHAR(5)),
21            4 A22 IS FIELD(CHAR(4));
22
23      SIZE.B = 30;
24
25  *** SA=SORT(SB,B1,1);
26
27      O1$CG IS GROUP(O1$C(*));
28      O1$C IS FIELD(CHAR(3));
29      O1$C=B1;
30      O1$IG IS GROUP(O1$I(*));
31      O1$I IS FIELD(NUM(5));
32      O1$IG=SORTC(O1$CG,1,SIZE.B);
33      SIZE.A=SIZE.B;
34      A1(SUB1)=B1(O1$I(SUB1));
35      A211(SUB1,SUB2,SUB3)=B211(O1$I(SUB1),SUB2,SUB3);
36      A212(SUB1,SUB2,SUB3)=B212(O1$I(SUB1),SUB2,SUB3);
37      A22(SUB1,SUB2)=B22(O1$I(SUB1),SUB2);
```

## B.5 THE COLLECT FUNCTION

```
     MODULE: COLT;
     SOURCE: C1;
     TARGET: C2;

     1 C1 IS FILE,
       2 C1R(*) IS RECORD,
         3 A1 IS FIELD(CHAR(3)),
         3 A2(3) IS GROUP,
           4 A21(2) IS GROUP,
             5 A211 IS FIELD(CHAR(5)),
             5 A212 IS FIELD(CHAR(5)),
           4 A22 IS FIELD(CHAR(4));

     1 C2 IS FILE,
       2 C2G(*) IS GROUP,
         3 C2R(*) IS RECORD,
           4 B1 IS FIELD(CHAR(3)),
           4 B2(3) IS GROUP,
             5 B21(2) IS GROUP,
               6 B211 IS FIELD(CHAR(5)),
               6 B212 IS FIELD(CHAR(5)),
             5 B22 IS FIELD(CHAR(4));

     SIZE.C1R=20;

***  C2 = COLLECT(C1,A1='   ');
     C1$C IS FIELD(NUM(2));
     C1$C = IF A1='   ' THEN 1 ELSE 0;
     C1$X IS FIELD(NUM(5));
     C1$X(SUB1)=IF SUB1=1 THEN 1
          ELSE IF C1$C(SUB1)^=0
               THEN C1$X(SUB1-1)+1
               ELSE C1$X(SUB1-1);
     C1$Y IS FIELD(NUM(5));
     C1$Y(SUB1)=IF SUB1=1 THEN 1
          ELSE IF C1$C(SUB1)^=0
               THEN 1 ELSE C1$Y(SUB1-1)+1;
     SIZE.C2G=C1$X(SIZE.C1R);
     SIZE.C2R(C1$X(SUB1))=IF SUB1=SIZE.C1R
          THEN C1$Y(SUB1)
          ELSE IF C1$C(SUB1+1)>0
               THEN C1$Y(SUB1);
     B1(C1$X(SUB1),C1$Y(SUB1))=A1(SUB1);
```

```
44      B211(C1$X(SUB1),C1$Y(SUB1),SUB2,SUB3)=
              A211(SUB1,SUB2,SUB3);
45      B212(C1$X(SUB1),C1$Y(SUB1),SUB2,SUB3)=
              A212(SUB1,SUB2,SUB3);
46      B22(C1$X(SUB1),C1$Y(SUB1),SUB2)=A22(SUB1,SUB2);
```

## B.6 THE FUSE FUNCTION

```
     MODULE: Y;
     SOURCE: Y2;
     TARGET: Y1;

     1 Y2 IS FILE,
       2 Y2G(2) IS GROUP,
         3 Y2R(6) IS RECORD,
           4 B1 IS FIELD(CHAR(3)),
           4 B2(3) IS GROUP,
             5 B21(2) IS GROUP,
               6 B211 IS FIELD(CHAR(5)),
               6 B212 IS FIELD(CHAR(5)),
             5 B22 IS FIELD(CHAR(4));

     1 Y1 IS FILE,
       2 Y1R(*) IS RECORD,
         3 A1 IS FIELD(CHAR(3)),
         3 A2(3) IS GROUP,
           4 A21(2) IS GROUP,
             5 A211 IS FIELD(CHAR(5)),
             5 A212 IS FIELD(CHAR(5)),
           4 A22 IS FIELD(CHAR(4));

     SIZE.Y2G=2;
     SIZE.Y2R=6;

***  Y1 = FUSE(Y2);
     U1$X IS FIELD(NUM(5));
     U1$Y IS FIELD(NUM(5));
     U1$X(SUB1)=IF SUB1=1 THEN 1
          ELSE IF U1$Y(SUB1-1)=SIZE.Y2R(U1$X(SUB1-1))
               THEN U1$X(SUB1-1)+1
               ELSE U1$X(SUB1-1);
     U1$Y(SUB1)=IF SUB1=1 THEN 1
          ELSE IF U1$Y(SUB1-1)=SIZE.Y2R(U1$X(SUB1-1))
               THEN 1
               ELSE U1$Y(SUB1-1)+1;
     SIZE.Y1R=SUM(SIZE.Y2R(SUB1),SUB1);
     A1(SUB1)=B1(U1$X(SUB1),U1$Y(SUB1));
     A211(SUB1,SUB2,SUB3)=
           B211(U1$X(SUB1),U1$Y(SUB1),SUB2,SUB3);
     A212(SUB1,SUB2,SUB3)=
           B212(U1$X(SUB1),U1$Y(SUB1),SUB2,SUB3);
     A22(SUB1,SUB2)=
           B22(U1$X(SUB1),U1$Y(SUB1),SUB2);
```

## B.7 THE CONCAT FUNCTION

```
     MODULE: CONCAT;
     SOURCE: CTMA,CTMB;
     TARGET: CTM;

     1 CTMA IS FILE,
       2 MAREC(*) IS RECORD,
         3 MA1 IS FIELD(CHAR(3)),
         3 MA2(3) IS GROUP,
           4 MA21(2) IS GROUP,
             5 MA211 IS FIELD(CHAR(5)),
             5 MA212 IS FIELD(CHAR(5)),
           4 MA22 IS FIELD(CHAR(4));

     1 CTMB IS FILE,
       2 MBREC(*) IS RECORD,
         3 MB1 IS FIELD(CHAR(3)),
         3 MB2(3) IS GROUP,
           4 MB21(2) IS GROUP,
             5 MB211 IS FIELD(CHAR(5)),
             5 MB212 IS FIELD(CHAR(5)),
           4 MB22 IS FIELD(CHAR(4));

     1 CTM IS FILE,
       2 MREC(*) IS RECORD,
         3 M1 IS FIELD(CHAR(3)),
         3 M2(3) IS GROUP,
           4 M21(2) IS GROUP,
             5 M211 IS FIELD(CHAR(5)),
             5 M212 IS FIELD(CHAR(5)),
           4 M22 IS FIELD(CHAR(4));

     SIZE.MAREC=10;
     SIZE.MBREC=15;

***  CTM=CONCAT(CTMA,CTMB);
     SIZE.MREC=SIZE.MAREC+SIZE.MBREC;
     M1(SUB1)=IF SUB1<=SIZE.MAREC
          THEN MA1(SUB1)
          ELSE MB1(SUB1-SIZE.MAREC);
     M211(SUB1,SUB2,SUB3)=IF SUB1<=SIZE.MAREC
          THEN MA211(SUB1,SUB2,SUB3)
          ELSE MB211(SUB1-SIZE.MAREC,SUB2,SUB3);
```

```
43      M212(SUB1,SUB2,SUB3)=IF SUB1<=SIZE.MAREC
44           THEN MA212(SUB1,SUB2,SUB3)
45           ELSE MB212(SUB1-SIZE.MAREC,SUB2,SUB3);
46      M22(SUB1,SUB2)=IF SUB1<=SIZE.MAREC
47           THEN MA22(SUB1,SUB2)
48           ELSE MB22(SUB1-SIZE.MAREC,SUB2);
```

## B.8 THE UNIQUE FUNCTION

```
     MODULE: Q;
     SOURCE: Q1;
     TARGET: Q2;

     1 Q1 IS FILE,
       2 Q1R(*) IS RECORD,
         3 SF1 IS FIELD(CHAR(1)),
         3 SF2 IS FIELD(CHAR(2)),
         3 SF3 IS FIELD(CHAR(3)),
         3 SF4 IS FIELD(CHAR(4)),
         3 SF5 IS FIELD(CHAR(5));

     1 Q2 IS FILE,
       2 Q2R(*) IS RECORD,
         3 TF1 IS FIELD(CHAR(1)),
         3 TF2 IS FIELD(CHAR(2)),
         3 TF3 IS FIELD(CHAR(3)),
         3 TF4 IS FIELD(CHAR(4)),
         3 TF5 IS FIELD(CHAR(5));

     SIZE.Q1R = 20;

***  Q2 = UNIQUE(Q1);
     Q1$M IS FIELD(NUM(3));
     Q1$VG IS GROUP(Q1$V(*));
     Q1$V IS FIELD(NUM(3));
     Q1$M(SUB1,SUB2)=IF SUB1>SUB2 THEN IF
                       SF1(SUB1)=SF1(SUB2)
                     & SF2(SUB1)=SF2(SUB2)
                     & SF3(SUB1)=SF3(SUB2)
                     & SF4(SUB1)=SF4(SUB2)
                     & SF5(SUB1)=SF5(SUB2)
                     THEN 1 ELSE 0 ELSE 0;
     Q1$V(SUB1)=SUM(Q1$M(SUB1,SUB2),SUB2);
     SIZE.Q1$V=SIZE.Q1R;
     Q1$C IS FIELD(NUM(2));
     Q1$C = IF Q1$V=0 THEN 1 ELSE 0;
     Q1$X IS FIELD(NUM(5));
```

```
39      Q1$X(SUB1)=IF SUB1=1
40           THEN IF Q1$C(SUB1)^=0
41                THEN 1
42                ELSE 0
43           ELSE IF Q1$C(SUB1)^=0
44                THEN Q1$X(SUB1-1)+1
45                ELSE Q1$X(SUB1-1);
46      Q1$SZ IS FIELD(NUM(5));
47      Q1$SZ=SUM(Q1$C(SUB1),SUB1);
48      Q1$IG IS GROUP(Q1$I(*));
49      Q1$I IS FIELD(NUM(5));
50      Q1$I(Q1$X(SUB1))=IF Q1$C(SUB1)^=0 THEN SUB1;
51      SIZE.Q1$I=Q1$SZ;
52      TF1(SUB1)=SF1(Q1$I(SUB1));
53      TF2(SUB1)=SF2(Q1$I(SUB1));
54      TF3(SUB1)=SF3(Q1$I(SUB1));
55      TF4(SUB1)=SF4(Q1$I(SUB1));
56      TF5(SUB1)=SF5(Q1$I(SUB1));
```

## B.9 THE UNION FUNCTION

```
     MODULE: UNION;
     SOURCE: U1,U2;
     TARGET: U3;

     1 U1 IS FILE,
       2 U1R(*) IS RECORD,
         3 SF1 IS FIELD(CHAR(1)),
         3 SF2 IS FIELD(CHAR(2)),
         3 SF3 IS FIELD(CHAR(3)),
         3 SF4 IS FIELD(CHAR(4)),
         3 SF5 IS FIELD(CHAR(5));

     1 U2 IS FILE,
       2 U2R(*) IS RECORD,
         3 RF1 IS FIELD(CHAR(1)),
         3 RF2 IS FIELD(CHAR(2)),
         3 RF3 IS FIELD(CHAR(3)),
         3 RF4 IS FIELD(CHAR(4)),
         3 RF5 IS FIELD(CHAR(5));

     1 U3 IS FILE,
       2 U3R(*) IS RECORD,
         3 TF1 IS FIELD(CHAR(1)),
         3 TF2 IS FIELD(CHAR(2)),
         3 TF3 IS FIELD(CHAR(3)),
         3 TF4 IS FIELD(CHAR(4)),
         3 TF5 IS FIELD(CHAR(5));

     SIZE.U1R = 20;
     SIZE.U2R = 15;

*** U3=UNION(U1,U2);
     1 U1$ IS GROUP,
       2 U1$U1R(*) IS GROUP,
         3 U1$SF1 IS FIELD(CHAR(1)),
         3 U1$SF2 IS FIELD(CHAR(2)),
         3 U1$SF3 IS FIELD(CHAR(3)),
         3 U1$SF4 IS FIELD(CHAR(4)),
         3 U1$SF5 IS FIELD(CHAR(5));
     SIZE.U1$U1R=SIZE.U1R+ISZE.U2R;
     U1$SF1(SUB1)=IF SUB1<=SIZE.U1R
          THEN SF1(SUB1)
          ELSE RF1(SUB1-SIZE.U1R);
```

```
44      U1$SF2(SUB1)=IF SUB1<=SIZE.U1R
45           THEN SF2(SUB1)
46           ELSE RF2(SUB1-SIZE.U1R);
47      U1$SF3(SUB1)=IF SUB1<=SIZE.U1R
48           THEN SF3(SUB1)
49           ELSE RF3(SUB1-SIZE.U1R);
50      U1$SF4(SUB1)=IF SUB1<=SIZE.U1R
51           THEN SF4(SUB1)
52           ELSE RF4(SUB1-SIZE.U1R);
53      U1$SF5(SUB1)=IF SUB1<=SIZE.U1R
54           THEN SF5(SUB1)
55           ELSE RF5(SUB1-SIZE.U1R);
56      U2$M IS FIELD(NUM(3));
57      U2$VG IS GROUP(U2$V(*));
58      U2$V IS FIELD(NUM(3));
59      U2$M(SUB1,SUB2)=IF SUB1>SUB2 THEN IF
60                        U1$SF1(SUB1)=U1$SF1(SUB2)
61                      & U1$SF2(SUB1)=U1$SF2(SUB2)
62                      & U1$SF3(SUB1)=U1$SF3(SUB2)
63                      & U1$SF4(SUB1)=U1$SF4(SUB2)
64                      & U1$SF5(SUB1)=U1$SF5(SUB2)
65                      THEN 1 ELSE 0 ELSE 0;
66      U2$V(SUB1)=SUM(U2$M(SUB1,SUB2),SUB2);
67      SIZE.U2$V=SIZE.U1$U1R;
68      U2$C IS FIELD(NUM(2));
69      U2$C = IF U2$V=0 THEN 1 ELSE 0;
70      U2$X IS FIELD(NUM(5));
71      U2$X(SUB1)=IF SUB1=1
72           THEN IF U2$C(SUB1)^=0
73                THEN 1
74                ELSE 0
75           ELSE IF U2$C(SUB1)^=0
76                THEN U2$X(SUB1-1)+1
77                ELSE U2$X(SUB1-1);
78      U2$SZ IS FIELD(NUM(5));
79      U2$SZ=SUM(U2$C(SUB1),SUB1);
80      U2$IG IS GROUP(U2$I(*));
81      U2$I IS FIELD(NUM(5));
82      U2$I(U2$X(SUB1))=IF U2$C(SUB1)^=0 THEN SUB1;
83      SIZE.U2$I=U2$SZ;
84      TF1(SUB1)=U1$SF1(U2$I(SUB1));
85      TF2(SUB1)=U1$SF2(U2$I(SUB1));
86      TF3(SUB1)=U1$SF3(U2$I(SUB1));
87      TF4(SUB1)=U1$SF4(U2$I(SUB1));
88      TF5(SUB1)=U1$SF5(U2$I(SUB1));
```

## B.10 THE DIFF FUNCTION

```
     MODULE: DIFF;
     SOURCE: D1,D2;
     TARGET: D3;

     1 D1 IS FILE,
       2 D1R(*) IS RECORD,
         3 SF1 IS FIELD(CHAR(1)),
         3 SF2 IS FIELD(CHAR(2)),
         3 SF3 IS FIELD(CHAR(3)),
         3 SF4 IS FIELD(CHAR(4)),
         3 SF5 IS FIELD(CHAR(5));

     1 D2 IS FILE,
       2 D2R(*) IS RECORD,
         3 RF1 IS FIELD(CHAR(1)),
         3 RF2 IS FIELD(CHAR(2)),
         3 RF3 IS FIELD(CHAR(3)),
         3 RF4 IS FIELD(CHAR(4)),
         3 RF5 IS FIELD(CHAR(5));

     1 D3 IS FILE,
       2 D3R(*) IS RECORD,
         3 TF1 IS FIELD(CHAR(1)),
         3 TF2 IS FIELD(CHAR(2)),
         3 TF3 IS FIELD(CHAR(3)),
         3 TF4 IS FIELD(CHAR(4)),
         3 TF5 IS FIELD(CHAR(5));

     SIZE.D1R = 20;
     SIZE.D2R = 15;

***  D3 = DIFF(D1,D2);
     D1$M IS FIELD(NUM(3));
     D1$VG IS GROUP(D1$V(*));
     D1$V IS FIELD(NUM(3));
     D1$M(SUB1,SUB2)=IF SF1(SUB1)=RF1(SUB2)
                        &SF2(SUB1)=RF2(SUB2)
                        &SF3(SUB1)=RF3(SUB2)
                        &SF4(SUB1)=RF4(SUB2)
                        &SF5(SUB1)=RF5(SUB2)
                      THEN 1 ELSE 0;
     D1$V(SUB1)=SUM(D1$M(SUB1,SUB2),SUB2);
     SIZE.D1$V=SIZE.D1R;
     D1$C IS FIELD(NUM(2));
```

```
45      D1$C = IF D1$V=0 THEN 1 ELSE 0;
46      D1$X IS FIELD(NUM(5));
47      D1$X(SUB1)=IF SUB1=1
48           THEN IF D1$C(SUB1)^=0
49                THEN 1
50                ELSE 0
51           ELSE IF D1$C(SUB1)^=0
52                THEN 1+D1$X(SUB1-1)
53                ELSE D1$X(SUB1-1);
54      D1$SZ IS FIELD(NUM(5));
55      D1$SZ=SUM(D1$C(SUB1),SUB1);
56      D1$IG IS GROUP(D1$I(*));
57      D1$I IS FIELD(NUM(5));
58      D1$I(D1$X(SUB1))=IF D1$C(SUB1)^=0 THEN SUB1;
59      SIZE.D1$I=D1$SZ;
60      TF1(SUB1)=SF1(D1$I(SUB1));
61      TF2(SUB1)=SF2(D1$I(SUB1));
62      TF3(SUB1)=SF3(D1$I(SUB1));
63      TF4(SUB1)=SF4(D1$I(SUB1));
64      TF5(SUB1)=SF5(D1$I(SUB1));
```

## B.11 THE PRODUCT FUNCTION

```
      MODULE: PROD;
      SOURCE: P1,P2;
      TARGET: P3;

      1 P1 IS FILE,
        2 P1R(*) IS RECORD,
          3 F1 IS FIELD(CHAR(1)),
          3 F2 IS FIELD(CHAR(1)),
          3 F3 IS FIELD(CHAR(1));

      1 P2 IS FILE,
        2 P2R(*) IS RECORD,
          3 FA IS FIELD(NUM(3)),
          3 FB IS FIELD(NUM(5));

      1 P3 IS FILE,
        2 P3R(*) IS RECORD,
          3 PF1 IS FIELD(CHAR(1)),
          3 PF2 IS FIELD(CHAR(1)),
          3 PF3 IS FIELD(CHAR(1)),
          3 PFA IS FIELD(NUM(3)),
          3 PFB IS FIELD(NUM(5));

      SIZE.P1R=3;
      SIZE.P2R=5;

***   P3=PRODUCT(P1,P2);
      SIZE.P3R=SIZE.P1R*SIZE.P2R;
      PF1(SUB1)=F1(1+(SUB1-1)/SIZE.P2R);
      PF2(SUB1)=F2(1+(SUB1-1)/SIZE.P2R);
      PF3(SUB1)=F3(1+(SUB1-1)/SIZE.P2R);
      PFA(SUB1)=FA(1+MOD(SUB1-1,SIZE.P2R));
      PFB(SUB1)=FB(1+MOD(SUB1-1,SIZE.P2R));
```

# BIBLIOGRAPHY


Alfo76 M. Alfonseca and M.L. Tavera
Extension of APL to Tree-Structured Information
Proc. APL 76, Ottawa, September 1976

Arvi80 Arvind and R.E. Thomas
I-structures: an Efficient Data Type for Functional Languages
MIT Lab. of Computer Sciences Technical Manual, TM-178, September 1980

Ashc77 E.A. Ashcroft and W.W. Wadge
Lucid, a Nonprocedural Language with Iteration
Comm. ACM, 20,7 (July 1977) 519-526

Codd70 E.F. Codd
A Relational Model of Data for Large Shared Data Banks
Comm. ACM, 13,6 (June 1970) 377-387

Codd72 E.F. Codd
Relational Completeness of Data Base Sublanguages
Data Base Systems, Courant Computer Science Symposium Series, Vol.6, Prentice-Hall, 1972

Dewa79 R.B.K. Dewar, A. Grand, S.C. Liu, J.T. Schwartz, and E. Schonberg
Programming by Refinement, as Exemplified by the SETL Representation Sublanguage
ACM Trans. Program. Lang. Syst., 1,1 (July 1979) 27-49

Fong77 A.C. Fong
Generalized Common Subexpressions in Very High Level Languages
Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, January 1977

Freu83 S.M. Freudenberger, J.T. Schwartz, and M. Sharir
Experience with the SETL Optimizer
ACM Trans. Program. Lang. Syst., 5,1 (January 1983) 26-45

Ghan73 Z. Ghandour and J. Mezei
General Arrays, Operators and Functions
IBM J. Res. Develop., 17,4 (July 1973) 335-352

Gokh83 M. B. Gokhale
Generating Data Flow Programs from Nonprocedural Specifications
Ph.D. Dissertation, Department of Computer and Information Science, Univ. of Pennsylvania, 1983

Gree81 R. G. Greenberg
Simultaneous Equations in the MODEL System with an Application to Econometric Modelling
Technical Report, The Moore School, Univ. of Pennsylvania, 1981

Gull79 W.E. Gull and M.A. Jenkins
Recursive Data Structures in APL
Comm. ACM, 22,1 (January 1979) 79-96

Hend76 P. Henderson and J.H. Morris, Jr.
A Lazy Evaluator
Conference Record of the Third ACM Symposium on Principles of Programming Languages, January 1976, 95-103

Iver62 K. Iverson
A Programming Language, Wiley, 1962

Leav74 B.M. Leavenworth and J.E. Sammet
Overview of Nonprocedural Languages
ACM SIGPLAN Symposium on Very High Level languages, March 1974, 1-12

Lisk77 B. Liskow, A. Snyder, R. Atkinson, and C. Schaffert
Abstraction Mechanisms in CLU
Comm. ACM, 20,8 (August 1977) 564-576

Lown81 P.G. Lowney
Carrier Arrays: An Idiom Preserving Extension to APL
Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages, January 1981, 1-13

LuKS81 K.S. Lu
Program Optimization Based on a Nonprocedural Specification
Ph.D. Dissertation, Dept. of Computer and Information Science, Univ. of Pennsylvania, 1981

LuKS82 K.S. Lu
MODEL Program Generator: System and Programming Documentation, Fall 1982 Version
Technical Report, The Moore School, Univ. of Pennsylvania, 1982

Pryw79 N.S. Prywes, A. Pnueli, and S. Shastry
Use of a Nonprocedural Specification Language and Associated Program Generator in Software Development
ACM Trans. Progrm. Lang. Syst., 1,2 (October 1979) 196-217

Pryw83 N.S. Prywes and A. Pnueli
Compilation of Nonprocedural Specifications into Computer Programs
IEEE Trans. on Software Engineering, SE-9,3 (May 1983) 267-279

Sang80 R. Sangal
Modularity in Non-procedual Languages through Abstract Data Types
Ph.D. Dissertation, Dept. of Computer and Information Science, Univ. of Pennsylvania, August 1980

Scho81 E. Schonberg, J.T. Schwartz, and M. Sharir
An Automatic Technique for Selection of Data Representations in SETL Programs
ACM Trans. Program. Lang. Syst., 3,2 (April 1981) 126-143

Schw83 S. Schwartz
The MODEL Concept: Nonprocedural Programming for Nonprogrammer
Technical Report, The Moore School, University of Pennsylvania

Shaw77 M. Shaw, W.A. Wulf, and R.L. London
Abstraction and Verification in Alphard : Defining and Specifying Iteration and Generators
Comm. ACM, 20,8 (August 1977) 553-564

Szym82 B. Szymanski
An Optimization of Stack Operations in the MODEL System
Technical Report, The Moore School, University of Pennsylvania, 1982

Vass73 J.P. Vasseur
Extension of APL Operators to Tree-Like Data Structures
Proc. of the APL Congress 73, Copenhagen, 1973, 457-464

INDEX