30 January 1986

THE MODEL CONCEPT

NON-PROCEDURAL PROGRAMMING FOR NON-PROGRAMMERS

January 1986

Table of Contents

PART I: INTRODUCTION TO MODEL

1. INTRODUCTION AND OVERVIEW

1.1. INTRODUCTION

This text contains description of the syntax and semantics of the language, MODEL, and techniques for its use. MODEL is a fifth-generation computer language. It is *equational* and *non-procedural*. (What these terms mean will be made clear to you as you begin to get involved in the MODEL system.) In this chapter, we shall discuss the value of the language.

MODEL is a tool for systems and program design and development. Like most computer languages, the MODEL system comes with a compiler which is used to receive and analyze the language statements. But as we shall see later on, the whole process of analysis and coding is radically different in the MODEL system from programming methods in current practice.

Using today's conventional technology, it is necessary for the analyst to have knowledge of how a computer works internally. Otherwise, the analysis is often unusable by the programmers. To express data processing requirements that are translatable into a procedural language requires knowledge of how a computer executes the solution of the problem. Non-procedural languages are problem-oriented and independent of knowledge of how the computer works.

MODEL eliminates the need for transfer of information from analyst to programmer. The MODEL system uses the computer to perform program design and coding automatically. In traditional systems design, after the requirements and analysis phases are completed, the programming task begins. Specifications are given to programmers who first perform the program design and then write and debug the programs. In MODEL, once a specification is completed, the "programming" task is also done as a byproduct. The specification itself is entered into the computer; submitted to the MODEL compiler. A PL/I program is generated, as well as a series of reports about the newly generated program.

MODEL has facilities for automating all program development phases: design, coding and testing. It reduces the analyst's involvement with computer execution through having the compiler interface with the computer and its environment. The analyst writes a specification which is entered into the computer. The specification is transformed into a PL/I program by the MODEL compiler. As soon as the specification is completed the system is ready for testing. In short, MODEL is an outgrowth of a widespread need to make programming more natural and more accessible to non-programmers.

Welcome to the realm of non-procedurality. You are going to learn a new methodology for systems analysis. It will help you to complete complex projects and enable you to conceptualize problems in a clear, precise manner, without having to at the same time worry about its implementation in a computer.

1.2. HOW TO USE THIS TEXT

Because MODEL is intended to be easy to use, there is no need for a high level of computer skills in order to learn it. This text is written for a range of people, from those who have had no previous experience with computers, to seasoned programmers looking for a new tool. We will explain all computer-related terms that we use, but we assume familiarity with algebra and basic mathematics. We also include comments written for people who know more about computers. (In planning this manual we set a goal of providing useful information about the MODEL system to people of all levels of computer experience who have a familiarity with algebra and basic mathematics.)

Here is a summary of the chapters. There are two parts. The first part provides an overall review of MODEL. It consists of three chapters as follows.

The first chapter explains MODEL in general terms. It gives suggestions for using the rest of the text and presents the components of a specification.

The second chapter reviews the basic concepts of data processing needed to program in MODEL. Some of this material may be a review for readers who have a computer background. The chapter should be reviewed before reading subsequent ones.

The third chapter discusses the process of developing a specification and the stages encountered in a specific application.

The fourth chapter presents the syntax and semantics of the language elements, the basic building blocks of the MODEL specification. The second part provides more detailed information on the statements of MODEL and how to use it. This part contains eight chapters as follows.

The fifth chapter presents the syntax and semantics for representing data to be described in MODEL specifications.

The sixth chapter describes how equations are composed.

The seventh chapter discusses use of subscripts in equations.

The eighth chapter describes the use of MODEL control variables.

The ninth chapter discusses techniques employed in debugging of a MODEL specification.

Chapter ten describes considerations in composing a MODEL specification of efficiency of the produced program.

Finally, chapter eleven contains a description of functions available in the MODEL system.

First skim each chapter to get an idea of what it's about, then read it for understanding. Flip back to earlier sections to find information that is referenced. Refer to the Index and Table of Contents when additional information on any specific topic is desired. Try also to follow the explanations accompanying the examples. This will help reinforce the important points.

1.3. USING THE MODEL SYSTEM

A program can be described as a means of taking certain data (e.g., lists of numbers or words) and producing new data as its output. The input can come from one or more external devices such as tape, disk, cards, or a keyboard. The output can appear on a report or go to a screen, tape, disk, or cards as well.

A conventional computer program (e.g. COBOL, PL/I, etc.) consists of a list of instructions that tell a computer what to do. If the program was correctly developed, the computer will carry out the planned sequence of events needed to achieve the system designer's goals.

The MODEL compiler is what is known as an *automatic program generator*. The user gives the compiler a set of requirements, (called a *specification*), and the system automatically produces a program in PL/I, which you can then

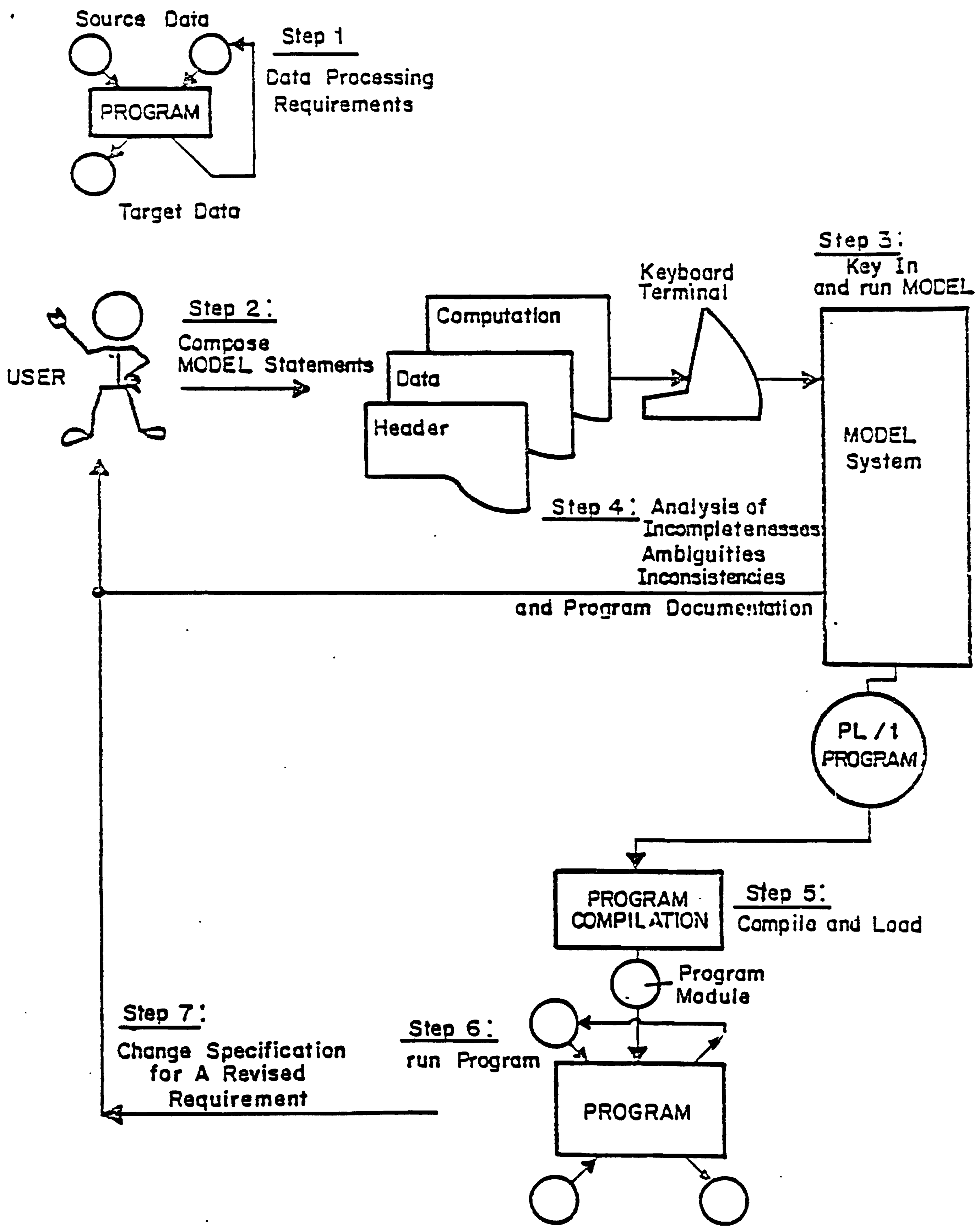


Figure 1.1 The Overall Procedure for Using MODEL

use as you would any other program. It is not necessary for you to know how to program in PL/I in order to use MODEL.

A MODEL specification consists of three main parts each with its own purpose: the *header* names the specification and any input and output files; the *data declaration* describes the organization of the data; the *equations* state the relationships among data (the problem statement).

The writing and using of a MODEL specification takes place in several stages. These are illustrated in Figure 1.1.

The process starts with a need to solve a problem through potential data processing applications. It may involve a few or several programs that communicate among themselves, and that read in and produce data. We call individual executable programs *modules.*

In the second step you compose a specification for each module in an attempt to state the problem in detail. Most users will probably enter their specifications through a terminal using a text editor.

In step 3, the MODEL compiler checks the input specification for incompleteness, ambiguities and inconsistencies, and reports any that it finds. It makes decisions in an attempt to fill in what is missing. If the specification compiles cleanly it produces a program in PL/I.

In step 4, the system generates documentation from its compile of your specification. You can select various report listings which will help you improve upon your original attempt at the problem statement. Possible report options include a reformatted listing of the specification, a variable cross-reference report, a subscript-range report that gives the dimensions and sizes of all arrays, a flowchart and listing of the generated program and an error report.

The error report contains both warning and error messages. Warnings indicate that aspects of your specification may have been reinterpreted because of incomplete information. Error messages refer to problems which prevented the MODEL compiler from successfully converting your specification into a PL/I program. Error messages describe the type of error along with the line in your listing where the error is located.

At this point, using the documentation of your specification, you may wish to rewrite and correct parts of it. When you resubmit a specification after correcting the errors, the compiler may present a new set of other errors which were previously hidden. (For example, the compiler does not discover semantic errors until errors in syntax are corrected.) This process usually takes place over a few stages.

In step 5, after you have successfully compiled your specification into a PL/I program, you can submit it to the PL/I compiler and load it in preparation for execution. Before you can run the program, you must have your input (Source) data in a form available to the computer and stored on some physical device such as cards, tape or disk.

In step 6, you run the program which the MODEL compiler has written from your specification. After examining your output (Target) data, if you are satisfied with the results, then you can stop. Otherwise, from step 7 you can alter the specification and try again. In general, it is recommended that you only make alterations in the MODEL specification and not in the PL/I program it produces, though this is sometimes a useful debugging technique. Because changes are easy, MODEL is well suited to the maintenance of programs as well as their creation.

1.4. WHY USE MODEL: A SUMMARY

We hope you will find MODEL simpler to use than other languages, because writing a MODEL specification only requires the ability to express the relationships (equations) that characterize your problem and frees you from the details of how the computer executes the solution. Writing a MODEL specification is conceptually allied to setting up equations for an algebra problem.

In MODEL, tasks such as input, output, program timing, memory allocation, and loops (repeated sets of operations) are all set up by the compiler based on your specification. To use MODEL, you don't need to know what a loop is. You can learn to use it effectively even if you have not had exposure to computer concepts. Your task is further simplified because the MODEL system conducts a thorough analysis of your specification and prompts you to correct any incompletenesses, inconsistencies, or ambiguities that it may have discovered. Finally, the MODEL compiler will produce a *highly efficient* program, which would otherwise require more detailed analysis.

MODEL will also be attractive to you if you are already a proficient programmer with sophisticated business or scientific needs. For example, the equations format supports simultaneous equations, making MODEL an ideal language in which to perform system modelling.

Unlike other very high level languages, MODEL is domain independent and general in purpose. It uses all the functions and data types available in PL/I, and allows new functions to be defined as well.

2. BASIC CONCEPTS

2.1. MODEL TERMS

In this chapter, we will introduce the basic concepts for developing equations and using data in MODEL.

2.1.1. FILES

A *file* in MODEL is an aggregate of scalar or array variables. A file can be viewed in many different ways by the user. This provides great versatility in approaches for solving problems. MODEL accesses files through its generated PL/I program. You describe the data in your specification in two different ways. One is a structural representation of the data in your source (input) intermediate and Target (output) files; the second is through expressing the relationships between data as indicated in your equations. You can think of the purpose of a program as being one of achieving the goal of realizing target files according to constraints expressed in equations.

Data may be external or internal. External data are data residing on external devices and are represented by Source File or Target File statements in the specificiation. Internal, or Intermediate files are data structures set up by the programmer to hold data, but this data will not be available after the execution of the program is completed.

2.1.2. VARIABLES

In a MODEL specification, the equations express relationships among variables. The concept of a variable in MODEL is different from the concept in a procedural program. In procedural languages, a variable is a slot which is given a name, like a mail slot. This slot can be filled with any one of a range of possible values, and these may change during the program execution.

MODEL is unlike most computer languages which allow you to assign more than one value to a variable in a program. One reason for this restriction in MODEL is that we follow the mathematical notion of an equation. An equation in MODEL defines a *dependent* variable on the lefthand side of an equal sign in terms of *independent* variables on the righthand side. If the independent variables are vectors or arrays, then the equation will be applied across the entire structure throughout its ranges. We consider each element of a multi-dimensional structure to be a separate variable. Therefore, variables contain one and only one value.

Furthermore, when the independent variables are given values, the dependent variable is computed from the single equation. A dependent variable is said to have only one value and that is the value of the equation which expresses the relationships among the variables needed to determine it.

2.1.3. SUBSCRIPTS

As in algebra, in MODEL, you use subscripts in parentheses to distinguish between the elements of array variables. The subscript gives the position of an element along a dimension of the array. Individual instances of data are represented by a variable name as well as subscript values. The subscript starts at 1 (first element) and goes up to the number of elements in the dimension which is also known as the dimension's *range*.

A subscript is placed in parentheses after a variable name, for example, ITEM(1). This represents the first element of the data structure called "ITEM". Subscript values must be integers or variables that have the value of integers.

The real power of MODEL comes through the use of *Subscript variables*. For example, in "ITEM(J)", J is said to be a *subscript variable.* "ITEM(J)" refers to a range of elements as J is allowed to vary. It can take any value from 1 to n, where n is the range of the dimension.

Of course, a variable may have more than one subscript depending on the number of dimensions in its structure. Generally, however, most standard MODEL applications will be limited in their use of multi-dimensional data structures. With just a few dimensions, the MODEL language can solve most every problem.

2.2. PRINCIPLES OF DATA ORGANIZATION

As part of writing a MODEL specification you need to describe how your data are organized. The equations you will write represent a bridge between your source and target files. How you set up your data structures will greatly impact the way you approach the more general task of describing the problem through the equations. (MODEL allows you to describe your problem as a means tc reaching solutions.)

The following discussion provides a theoretical background of representations for multi-dimensional structures as it applies to MODEL data structures. It may be skimmed if you feel comfortable with the concepts of Arrays and Tree structures. Later chapters will explain how these conventions are applied in data declarations.

2.2.1. ARRAYS AND TREES

The phrase "how your data are organized" concerns also the names and subscripts you use in equations to refer to each piece of data. In MODEL, you may think of data as consisting of lists (vectors) of basic elements, lists of lists (matrices), lists of lists of lists (3 dimensional arrays), etc. Each list has a name and a *repetition count,* that is, the number of elements it contains. The use of subscripts is consistent with organizing data. (The use of dimensions of arrays and subscripts in describing arrays will be explained shortly). For example, each row in a table may be thought of as a list, and the whole table may therefore be thought of as a list of lists. Each element in the table can be identified by providing the values of its subscripts. Figure 2.2 shows a 3x4 array of items in shopping lists for three different stores, each list contains four items.

		ITEM #			
		1	2	3	4
	LIST OF STORES				
1	SUPERMARKET	apples	milk	bread	cheese
2	DRUG STORE	band-aids	cough drops	comb	soap
3	SCHOOL BOOKSTORE	MODEL text	calendar	pens	notebook

Figure 2.2
Shopping List Array

In MODEL this organization is described as a 3 level hierarchial structure as follows:

```
1  SHOPPING_LISTS IS GROUP,
    2 STORE (3) IS GROUP
       3 ITEM (4) IS FIELD (CHAR(12));
```

This table has a rectangular structure; it consists of three STORE lists of each four *items* with 12 characters allowed for each item. You can do this because each list has the same number of items. In a rectangular array each row should have the same number of entries as every other row. The same thing is true for the columns.

Suppose after checking your refrigerator, you realize that you need to add chicken to your supermarket shopping list. To keep track of a set of lists where you can potentially have a different number of items on each list, you need to allow also non-rectangular arrays with different number of elements in each row. Another alternative is to structure data in the form of a tree.

```
                         SHOPPING  LIST
                                  |
               ___________________|__________________
               |                  |                  |
               |                  |                  |
          SUPERMARKET        DRUG STORE       SCHOOL BOOKSTORE
        _______|_______     _____|______      ______|______
        |   |   |   |   |   |     |    | |     |    |    |   |
 apples | bread | chicken   |cough drops |soap | calendar |  pens
        |       |           |            |     |          |
      milk    cheese    band-aids      comb  MODEL text   notebook
```

Figure 2.3

Shopping List Tree

Figure 2.3 shows the shopping list array rearranged into a tree. Trees consist of *nodes* and the *branches* between them. Nodes are the parts of the tree that are given distinct names. In this case, the name of each store is a node in the tree. Different levels in a data tree are connected by branches. These branches show how the nodes are interrelated. This kind of data organization is called *hierarchical*, because nodes representing lists that contains other lists are higher up in the tree than the nodes representing the lists they contain.

For any pair of nodes connected by a branch, the upper node is called the *parent*, and the lower node is called the *child*. One node can be the parent for any number of child nodes. Also the same node can be the parent of the one immediately below it in the tree and the child of the one immediately above it. The node SHOPPING_LISTS, representing the whole tree, is parent for the nodes representing shopping lists for individual stores. Similarly, store lists are parents for individual items. The variability of number of elements of the items for each store can be expressed by giving the maximum and minimum number of repetitions of items for each store.

```
1  SHOPPING_LISTS IS GROUP,
   2  STORES (3) IS GROUP,
      3  ITEM (4:5) IS FIELD (CHAR 12);
```

Such an array will be called a *ragged edge* array. The actual number of elements will have to be defined by an equation, as described later.

2.2.2. DESCRIBING ARRAY STRUCTURE

The following data structures use numbers as the basic elements. This is a matter of convenience. Data structures may be constructed using character elements. There can be data structures of any size containing letters, names, binary bits, etc. There can also be data structures which include both numbers and character strings as basic elements.

A scalar or zero-dimensional array is a single variable, like X in algebra. See Figure 2.4(a) for examples. Giving a variable name, such as X or LUCY, to a scalar is enough to identify it uniquely. This means using the variable name, either you or the computer can refer to a scalar without getting it confused with any other scalar.

.05	6.02E23	11
LUCY	ETHEL	LITTLE_RICKY

(a) **Examples of Scalars**

I	1	2	3	4	5
FELIX(I)	.05	.76	.31	.40	.72

(b) **Example of a Vector**

ROW(I)				
1	7	4	11	3
2	5	6	4	2
3	1	3	8	12
	1	2	3	4
	COLUMN(J)			

(c) **Example of Matrix MOE(I,J)**

MATRIX(I)	1				2			
ROW(J) 1	7	4	1	3	8	9	9	1
2	5	6	4	2	4	7	3	5
3	1	3	8	2	2	6	5	6
	1	2	3	4	1	2	3	4
	COLUMN(K)				COLUMN(K)			

(d) **Example of Three-Dimensional Array CURLY(I,J,K)**

Figure 2.4 Arrays of Different Sizes

A vector or a one-dimensional array is a single row of element variables, each of which can take on one value. See Figure 2.4(b) for an example of a vector. The number of elements (numbers) in a vector is called its *range* or *size*. Giving a name, such as FELIX, to a vector will not enable you to specifically refer to any of its elements. For a vector of range M, where M >= 1, any member of that vector can be uniquely specified with a subscript I, where 1 <= I <= M. From the example in Figure 2.4(b), FELIX(1) = .05, FELIX(2) = .76, FELIX(3) = .31, and so on. Each element of the vector has a different subscript. The whole vector is called FELIX, and its range is 5. Any data structure, such as FELIX, whose elements are distinguished with subscript values is also called a repeating variable, or a subscripted variable.

An ordinary table of numbers is a two-dimensional array or a matrix. A matrix consists of a certain number of vectors laid down next to each other. Each vector is a row in the table, or if you prefer, each can be a column. Each element of a matrix must have two subscripts, (I,J), to specify it uniquely, where one of the subscripts refers to the row number and the other refers to the column number. For example, in Figure 2.4(c), with I being row number and J being column number, MOE(2,3) = 4, while MOE(3,2) = 3. The range (or size) of the row dimension of MOE(I,J) is 3, while the range of the column dimension is 4.

A three-dimensional array consists of a series of matrices arranged one above the other to form a rectangular prism, like a stack of blackboards. See Figure 2.4(d) for an example (presented two-dimensionally). Each element of a three dimensional array must have three subscripts (I,J,K), where one of the subscripts refers to the number of the matrix, a second refers to row number, and a third refers to the column number. For example, in Figure 2.4(d), with I being matrix number, J being row number, and K being column number, CURLY(1,2,3) = 4, while CURLY(3,2,1) does not exist. The ranges of I, J, and K are 2, 3, and 4, respectively.

A four-dimensional array would consist of a series of rectangular prisms of the same size and shape, like several stacks of blackboards. The reader can probably infer that an element within it would require four subscripts (I,J,K,L) for unique specification. The general rule is that an array of n dimensions requires n subscripts in order for each element to be uniquely identified.

2.3. EQUATIONS

2.3.1. EXPRESSING REQUIREMENTS AS EQUATIONS

The form for representing a problem is expressed through equations. These are statements that define the value of one variable in terms of one or more other variables, constants or functions. For example,

```
FRED = 5;
```

defines the value of the variable FRED to be 5. (FRED could not be given another value in the same specification.)

```
ITEM(2) = ITEM(1);
```

defines the value of ITEM(2) as equal to the value of ITEM(1). This equation could appear at any point in the specification, even before the value of ITEM(1) is defined. The value of ITEM(1) would be defined in another assertion, such as

```
ITEM(1) = 6;
```

In writing equations, it is common (and efficient) to use subscript variables, instead of subscript numbers. In this way, all the elements are defined in one equation. This is illustrated with the following example: Suppose we have two source vector variables, DIVIDEND, DIVISOR and a target vector variable, QUOTIENT. (By t[illegible]rm "Vector" variable, we mean a one dimensional data structure.) Essentially, we have a list of dividends and a list of divisors, and we want a list of their respective quotients. The values of <u>all</u> the quotients are defined in a <u>single</u> equation:

```
QUOTIENT(J) = DIVIDEND(J) / DIVISOR(J);
```

In words, "any Jth quotient is defined as the Jth dividend divided by the Jth divisor." This would apply to all values of J, which will generally vary from 1 up to the range of the source file dimension.

Often, it is not possible to define target variables directly in terms of source variables. Instead, we may have to define *interim* variables which will be given values based upon the source variables, and still other variables defined in terms of these, and so on. For example, QUOTIENT(J) might be an interim variable instead of a target variable, and there might be another equation defining some other variable LOGQUO(J), such as

```
LOGQUO(J)  =  LOG(QUOTIENT(J));
```

2.3.2. DEFINING SINGLE VALUES FOR VARIABLES

Non-procedurality requires that a variable be defined uniquely. Contradictory equations which give a variable different values are not allowed. Each target and interim variable is defined by one equation in which that variable appears on the lefthand side of the equal sign. (It is possible to use more than one equation to define a variable provided each applies when a different mutually exclusive condition is met). The values of source variables are obtained from data in the source files.

Non-procedurality may at first create some problems, especially for a person who is used to conventional computer languages where a variable is given different values at different places in a program. It requires a different way of conceptualizing.

The following example illustrates the difference between procedural and non-procedural. Suppose we wish to find the factorial of M. (Factorial of M, M!, is the product of all the integers from 1 to M, i.e. 1*2*3*...*M). In a procedural language we would write a loop program to repeatedly multiply a previous product by the next bigger integer until the number M is reached. (In this example, the value for M! will be stored in the variable, N after the loop is completed execution).

```
N=1;
DO I 2 TO M;
 N=N*I;
END
```

In MODEL the factorial specification would be defined using a subscripted variable N(I). Using a repetition count of M, or a SIZE statement, the subscript variable, I, will automatically vary from 1 to M. Each element of N(I) corresponds to a partial product. This equation would have the following form:

```
N(I) = IF I = 1 THEN 1 ELSE I * N(I-1);
```

In non-procedural terms, the following has occurred: We have defined a vector called N with a range of M. This vector is a list of numbers the first of which would be represented by N(1), the second by N(2), etc. I is a subscript variable which covers the range of the vector N.

The above equation says the following: When I is 1, (i.e. N(1), the first element in vector N), then N(I) has a value of 1 (this is the meaning of "THEN 1"). When I is a value other than 1, (the "ELSE" clause), then N(I) has the value of I*N(I-1). (The symbol * indicates multiplication.) Expressed otherwise, this simply says that a factorial is the product of the previous factorial, N(I-1) and the next number in the series. In this case N(5) would have a value of 120.

2.4. SAMPLE SPECIFICATION

In this section, we have selected an example of a MODEL specification. It solves a simple problem and has been chosen for illustrative purposes. Review this section carefully because the concepts discussed can be applied to the writing of specifications in general. In later sections of the manual we shall give sophisticated examples which will explore more advanced techniques in MODEL.

The following specification solves a problem involving summing groups of five numbers. The source file consists of records each containing the numbers. The target file is to consist of records each with the sum of the numbers in the corresponding source records.

The name of this specification is SUMMARIES. It is shown in Figure 2.1.

```
00100   Module: Summaries;
00200   Source: Figures;
00300   Target: Sums;
00400
00500   1  Figures is file,
00600      4  Fig_record (*) is record,
00700         7  Number_group (5) is group,
00800            10  Number is field (pic '99999');
00900
01000   1  Sums is file,
01100      6  Sum_record (*) is record,
01200         12  Totals is field (pic '999999');
01300
01400   1  Work_fld  is field (pic '999999');
01500
01600   (x,y) are subscripts;
01700
01800   Work_fld(x,y) = if y=1 then Number(x,1)    else
01900                                Work_fld(x,y-1) + Number(x,y);
02000
02100   Totals(x)=work_fld(x,5);
```

The [illegible]

Figure 2.1

The Specification SUMMARIES

2.4.1. THE SECTIONS OF THE SPECIFICATION 'SUMMARIES'

Look over this specification. Read through the statements. We are going to discuss each of them in detail. (Note, the line numbers have been added to assist in clarity and are not really part of the specification).

The module name and the source and target files (lines 100-300) comprise the header section. In the case of SUMMARIES, there is one source file, FIGURES, and one target file, SUMS. The next part (lines 500-1600) contains the layouts of the data structures. Data structures are the statements which define the shape and elements of the source, target and interim files. The last part of the specification are the equations. Note that this program has only two, (lines 1800-2100). As you will see, with these two equations the designer of this specification is able to accomplish the required goal.

In SUMMARIES there is one source file, FIGURES. The statements describing this file are in lines 500-800. The FILE statement (line 500) must be present in order for the MODEL system to recognize the external file indicated in the header section (line 200). In combination with a SOURCE or TARGET statement, the FILE statement tells the system that the statements which follow represent the data structure of external files which will be read and/or written in the generated program. FIGURES, which is a source file, will be supplying data to the program which will be used to define other variables in the specification.

2.4.2. THE DATA STRUCTURES

There are four types of data statements, FILE, RECORD, GROUP and FIELD. The statements describing FIGURES contain all four. In the description for FIGURES, the statements each have similar formats beginning with a level number, (1,4,7 and 10), followed by the variable name, (FIGURES, FIG_RECORD, etc.) and a keyword, (FILE, RECORD, GROUP, etc.). In addition, the RECORD and GROUP statements, (lines 600 and 700), contain what are known in MODEL as repetition counts, (this is the asterisk and the number 5 which appear in parentheses in lines 600 and 700 respectively).

Level numbers are used to indicate the hierarchical relationship among the elements of the data structure. You will notice in FIGURES that there are four levels, the highest represented by the FILE statement which has a level number of 1 (all structures must begin with this number) and the lowest is the FIELD with a level number of 10. Level numbers do not have to be sequential as you can see from the example.

As mentioned earlier, MODEL allows you to define your data as multi-dimensional structures. The number of dimensions may be determined by counting how many repetition counts exist between a given FIELD statement and the FILE statement at the top. A repetition count follows the name of the RECORD, GROUP or FIELD to which it will apply and usually takes the form of either a single number, an asterisk or two numbers seperated by a colon. In the case of FIGURES, you can tell that the data structure is two dimensional because there are two repetition counts, one for the RECORD, (line 600) and the other for the GROUP, (line 700).

We get information regarding the *ranges* of the dimensions from the repetition counts. The (*), indicates that there are an unknown (or variable) number of repetitions in the first dimension, while the (5) indicates that there are a fixed number of repetitions of the second dimension.

2.4.3. RECORD AND GROUP STATEMENTS

All external files, (source or target), must have a RECORD level in their data declaration representation. This is the unit the computer uses for input or output of the external file. Although both RECORD and GROUP statements contribute to defining the dimensionality of a structure, it is only the RECORD statements which are used for external files.

In source files, you may use multi-dimensional data structures to describe data which are in some prescribed format which you are unable to change. In this case, it is important that you understand the connection between the representation as it is created in the MODEL specification and the actual data in the external file.

In the case of FIGURES, the record contains 5*5 or 25 bytes (a PIC '99999' is a five byte field and this is multiplied by the repetition count of NUMBER_GROUP) and there are an unknown number of data records.

2.4.4. THE TARGET FILE, 'SUMS'

In SUMMARIES there is one target file, SUMS. The statements describing this file are in lines 1000-1200. The FILE statement (line 1000) tells the system that the following statements represent the data structure of the external file which will be written out in the generated program. The statements describing SUMS contain the FILE statement, a RECORD statement and a FIELD statement. Since all external files, (source or target), must have a

RECORD level in their data declaration representation, we have one for SUMS as well. We can see that the units the computer will use for output in the external file will be 6 byte fields, (we have no group repetition this time).

The RECORD statement, (line 1100), contains a repetition count of (*) indicating that SUM_RECORD will be written an unknown number of times and since this is the only repetition count in the structure, the structure is one dimensional.

2.4.5. THE INTERIM FILE

It is sometimes impossible to define target file variables directly in terms of source file variables. Instead, we may have to define *interim* variables which will be given values based upon the source file variables and be used in turn to provide values to target file variables. This is the case in SUMMARIES. In order to define the target file field, TOTALS, it was necessary to define an interim which would have the same structure as the source file, but which could be used to generate the target field. This variable is defined in line 1400 and is called WORK_FLD.

It is also possible to have FILE statements for an interim file structure, but as can be seen from this example, it is not always necessary to code this statement.

If you look at the equation in which the interim field is defined, (lines 1800-1900), you will notice that the field has two dimensions. Many efforts have been made to make the MODEL language tolerate incompletenesses and inconsistencies in specifications. When incompletenesses and inconsistencies are found, the MODEL processor tries to correct the specification in a reasonable way. The user is saved from the work of writing a FILE statement for the interim field, WORK_FLD. In this case, the compiler assumes that there are two dimensions above the field with the same ranges as the source file. [illegible]
[illegible] Fortran can [illegible]

An interim structure is completely self-contained and data from it will never remain after the program has finished executing.

2.4.6. RANGES AND THEIR PROPAGATION

The program has two subscript variables, identified in line 1600. Subscript variables are used in equations to enable you to define an entire array structure in a single statement. The values which the subscripts will assume have to match the ranges of the dimensions of the variables which use them. For example, we know that the second dimension of NUMBER has a range of 5. Therefore, we need some way of ensuring that in the expression NUMBER(X,Y), the subscript Y assumes values which go from 1 to 5.

A subscript variable which appears in a subscript statement as in line 1600 is known as a *Global subscript.* This means that whatever range X and Y each have, they will vary over the same range in every equation in which they are present.

Whenever there is a source file structure with a record which has a (*) for a repetition count, the MODEL compiler assumes that the end of the source file is the end of the range. Therefore, the generated program will read until it reaches end-of-file.

We can now draw some conclusions about the range of the target file: The equation which gives a value to the target field, TOTALS, has TOTALS subscripted by X. Since X is a global variable, therefore we know that the first dimension of the source file (which is also subscripted by X) has the same range as the target file. Since the range of the source record is the same as the range of the target record, both files have the same number of records.

Suppose there are 500 records in the file. Then the array, FIGURES, will be 500x5. The vector SUMS will have 500 elements.

2.4.7. THE EQUATIONS

Now let us look at the equations in this program. One provides a link from source to interim, (equation 1, lines 1800-1900) and the other, from interim to target, (equation 2, line 2100).

Equation 1 contains the actual calculation of the sum of 5 numbers. It is in a form known as a conditional equation. Based upon the result of a logical test, (in this case, Y=1), the equation will assume different values, either NUMBER(X,1) or WORK_FLD(X,Y-1)+NUMBER(X,Y).

The simplest way to describe what equation 1 accomplishes is to evaluate WORK_FLD as we vary X and Y. With each record retrieved by the PL/I program, (i.e. as X varies in its range), X will be incremented and Y will vary within its range (up to 5). When Y is 1, WORK_FLD(X,Y) will be given a value of the first number field in the source record. When Y is 2, it will add the second number field in the source record to the value of WORK_FLD(X,2-1) [i.e. WORK_FLD(X,1)]. But, this value was obtained from the first number field in the source record. So the value of WORK_FIELD(X,2) will now reflect the sum of the first two numbers in the source record. The calculation will occur throughout the range of Y. When Y is 5, we will have the sum of all 5 numbers in the field WORK_FLD(X,5). This is the value we wish in the target file. Consequently, the second equation, is just a matter of making this assignment.

When this module is run through the MODEL compiler, a PL/I program is generated. (PL/I is a procedural programming language.) In this case, the generated PL/I program is about 84 lines, much longer than the specification which led to its creation. The next stage would involve sending the generated program through a PL/I compiler, converting the object file to an executable form and testing the program with test data.

3. PROGRESSIVE MODULAR DEVELOPMENT

3.1. INTRODUCTION

This chapter deals with the subject of progressive modular development. We will approach this subject in two different ways: First we will discuss the subject in general terms, describing the whole process of systems design in MODEL and the stages of prototyping. This discussion assumes that the entire system design, which may consist of several modules, is being undertaken, and outlines the approach by which the system gradually takes shape and definition and eventually the problem is solved.

Next, we shall look at the question of modular development on the lowest level, the stages of coding an individual specification. We shall go through the step-by-step stages which the designer went through before reaching the final specification. This application makes use of sublinear indices and therefore should be reviewed in detail by anyone wishing to understand how this feature adds to the power and scope of the MODEL language.

3.2. THE GENERAL PROCESS OF MODEL SYSTEMS DEVELOPMENT

What approaches are available to a systems analyst wishing to develop a new data processing application? Generally there have been two basic plans of attack.

One involves an attempt to solve the problem through brain-storming, developing a seemingly practical solution and presenting this to a programmer or programmers to begin the coding process. Generally, snags arise and the original plans are revised. This technique requires the analyst to have had experience and expertise in the discipline of systems design, to have an intimate, a priori knowledge of the particular aspect of the business to which the application will eventually lend assistance. Having programming skills is also frequently required.

The second method for systems development involves building a system up through prototyping with the assumption that a solution will eventually come through achieving a better focus on the specific characteristics of the problem. This method came about because it was felt that analysts spent too much time in the planning phase before all the basic components of the problem had been uncovered and subsequently had to scrap initial solutions which had been developed under incomplete scenarios.

Development of systems in MODEL involves a gradual convergence toward the solution through an iterative process which involves the non-programmer and the computer. Unlike traditional application design, in the MODEL development process, a knowledge-base of software engineering principles are stored in the computer and offer help and guidance. The non-programmer is led down the path toward a solution as he states and re-states his problem with the computer giving feedback each time. With each compilation, your specification or specifications help to clarify the elements involved. As the problem begins to take form you find that you are also simultaneously creating a solution.

The systems development process can be broken into five stages. If you are working on a specific problem, you may find it useful to try and answer some of the questions posed below as they apply to your objectives.

3.2.1. STAGE ONE: A SYSTEM OVERVIEW

Preparatory to any work, the designer will have an idea in general terms of what is needed to be accomplished. Meetings with users or other system stakeholders will have occurred. Though the actual form which the final specification will take need not be considered, a broad understanding of the basic elements is essential.

The following questions should be addressed: What data will be given to be used as sources of information? What is going to be the end use of this information? (Generally at this early stage, you have an idea of the data sources and the final product in the application, but need only a vague notion of how the two will be connected.)

The data analysis, identifying sources and targets, is the first stage in the evolution of the system configuration. Inputs can be connected to outputs through modules which are still undefined. In the next stage we shall further break down the function of each module.

3.2.2. STAGE TWO: FROM PROBLEM TO SUB-PROBLEMS

At this point you will have identified all inputs and outputs of one big box which represents the entire system. It is now necessary break it down further. The overall problem must be divided into sub-problems. The idea is to proceed based on first impressions of a general nature as to what you can identify. The result of this step should be a high level configuration of the system, with boxes representing modules. Source files are the inputs to the boxes and target files are the outputs.

Identify points of data overlap. This serves as glue between boxes. Which sub-problems share the same source data? Which target data is source data for another box? Whenever possible, communication may be established between sub-problems sharing data.

3.2.3. STAGE THREE: IDENTIFY GOALS/SUB-GOALS

Some areas are more clearly defined than others. As a general rule, you should begin with the areas which have the least definition. Focusing on these modules will uncover the rest of the required work.

The goals of each box should be listed. As you proceed, it may be apparent that a box can be broken down further into smaller boxes. These sub-divisions help to define the goals through the creation of sub-goals.

Sub-goals are frequently identical with the target files. Often, the sub-goals serve as an intermediate collection of data needed to achieve a desired goal.

3.2.4. STAGE FOUR: PROTOTYPING

Once the breakdown of the application is achieved, it will be most appropriate establish the relationships between files through equations. MODEL will help you along in this process. Define your Source, Interim, Control and Target variables and don't concern yourself with the order of execution calculations or details of field manipulations.

Great care should be given to mapping out the data structures. All subscripts should be defined and their ranges. Only the critical fields of Source, Interim and Target files (sort fields, control variables, ranges, in short all fields critical to the design) should be coded up. These initial versions of your files need not be complete.

As this stage progresses, the partial data structure along with groups of equations can be submitted to the MODEL COMPILER which will return reports illustrating the progress, as well as indications of inconsistencies pointing you back to reconsider approaches to the particular area of the application (and possibly to breaking goals into smaller sub-goals).

3.2.5. STAGE FIVE: PROGRAM GENERATION

The prototyping generation is the most time consuming. You may have to go through many iterations before the final product meets the required goals.

As soon as the relationships are defined to satisfaction, further details such as report layouts and texts can now be added to complete the specification.

When all the module specifications have been completed, not only are the modules defined, but the overall application configuration is established. Another system not defined here is the MODEL *configurator* which creates JCL (VAX only) and ensures that data will be processed in proper sequence.

The PL/I code is highly optimized. The system is ready for turnover to production.

3.3. DEVELOPMENT OF AN INDIVIDUAL MODULE: A SPECIFIC CASE

The following section charts a course through the stages of specification design as it was actually experienced by a non-programmer who was developing an application which would merge data from two files. The designer made several attempts to state the problem clearly. At each stage, the system directed him to the next area which needed to be addressed. In this manner, the specification eventually took form and became the basis for a section of a larger system.

3.3.1. REQUIREMENT

The problem analysis uncovered the following: There were two files with different data but with one key in common. There was a need to combine these files into a single file which would have all the data of both files, but it was desired that this be accomplished as follows: If there was a match on the key field, (i.e., if a certain key was present on both files), then there was to be only a single record created with the information from both present. If a given key was present in one file but not in the other, then the field which was missing data was to be given a value which would have a special meaning indicating that the key had not been present on that file. One additional characteristic is that both source files were sorted by the key field and when a key existed it was only to occur one time in a given file, (i.e. all the information associated with a key was unique).

Listed below is an example of sample data with 2 Source files and the Target file the system required.

```
FILE 1                    FILE 2

Key     Info              Key      Info

01      John              02       Susan
03      Jim               04       Trisha
05      Harold            06       Louise
06      Bill              07       Ruth
07      Tom               08       Mary


Target File

Key            Info1             Info2

01             John              Not found
02             Not found         Susan
```

```
03             Jim             Not found
04             Not found       Trisha
05             Harold          Not found
06             Bill            Louise
07             Tom             Ruth
08             Not found       Mary
```

The goal was to state the problem in MODEL and ultimately generate a PL/I program which would accomplish this task.

3.3.2. SPECIFICATION - FIRST FORM

The non-programmer went through the following analysis: "I have two files and I don't know anything of their respective ranges: Clearly they will both get repetition counts of '*'. Since they will almost definitely have different ranges, I will set up three global subscripts, one for each Source file and one for the Target.

"When the key of File1 is equal to the key of File2, my target file Fields will both have values. Otherwise, I shall use a less than, '<', condition to indicate that I don't have a match, and I will generate target records which have some data missing."

The non-programmer wrote the following specification and submitted it to the MODEL compiler.

```
Module: Match;
Source: File1,File2;
Target: Combine;

1  File1 is file,
  2  File1rec (*) is record,
     3  key1 is field (char 2),
     3  info1 is field (char 20);

1  File2 is file,
  2  File2rec (*) is record,
     3  key2 is field (char 2),
     3  info2 is field (char 20);

1  Combine is file,
  2  Comborec (*) is record,
     3  keyC is field (char 2),
     3  info1c is field (char 20),
     3  info2c is field (char 20);

(f1,f2,c) are subscripts;
keyc(c)= if key1(f1)=key2(f2) then key1(f1) else
         if key1(f1)<key2(f2) then key1(f1) else
                                   key2(f2);

info1c(c)= if key1(f1)=key2(f2) then info1(f1) else
           if key1(f1)<key2(f2) then info1(f1) else
                                     'NFOUND';

info2c(c)= if key1(f1)=key2(f2) then info2(f2) else
           if key1(f1)<key2(f2) then 'NFOUND' else
                                     info2(f2);
```

First, notice that the problem has been oversimplified for ease of progressive modular development. Although File1 and File2 have many fields in the real application, in this initial stage it is recommended to leave out the inessential calculations and express the problem in broadest terms.

Running the specification through the MODEL compiler produced the following error messages:

```
                    --- ERROR(S)/WARNING(S) DETECTED FOR COMBO1.INP: ---

   *NO ERROR/WARNING DETECTED DURING SYNTAX ANALYSIS*


*WARNING* IIX1: SOME SUBSCRIPTS APPEAR ON THE RIGHT-HAND-SIDE BUT NOT ON THE
        LEFT-HAND-SIDE OF AN ASSERTION. SELECTION IS IMPLIED FOR ",F2,F1" IN
        ASSERTION AASS10.
*WARNING* IIX1: SOME SUBSCRIPTS APPEAR ON THE RIGHT-HAND-SIDE BUT NOT ON THE
        LEFT-HAND-SIDE OF AN ASSERTION. SELECTION IS IMPLIED FOR ",F2,F1" IN
        ASSERTION AASS8.
*WARNING* IIX1: SOME SUBSCRIPTS APPEAR ON THE RIGHT-HAND-SIDE BUT NOT ON THE
        LEFT-HAND-SIDE OF AN ASSERTION. SELECTION IS IMPLIED FOR ",F2,F1" IN
        ASSERTION AASS9.
*WARNING* RGP1: DIMENSION 1 OF "AASS9" IN RANGE SET NUMBER 1 DOES NOT HAVE AN
        EXPLICIT RANGE.
*WARNING* RGP1: DIMENSION 1 OF "FILE1.INFO1" IN RANGE SET NUMBER 2 DOES NOT
        HAVE AN EXPLICIT RANGE.
*WARNING* RGP1: DIMENSION 1 OF "FILE2.INFO2" IN RANGE SET NUMBER 3 DOES NOT
        HAVE AN EXPLICIT RANGE.
*ERROR* EVL3: NO RANGE DEFINITION FOUND FOR RANGE 1 IN THE RANGE TABLE.
*ERROR* EVL3: NO RANGE DEFINITION FOUND FOR RANGE 1 IN THE RANGE TABLE.
*ERROR* EVL3: NO RANGE DEFINITION FOUND FOR RANGE 1 IN THE RANGE TABLE.
*ERROR* EVL3: NO RANGE DEFINITION FOUND FOR RANGE 1 IN THE RANGE TABLE.
*ERROR* EVL3: NO RANGE DEFINITION FOUND FOR RANGE 1 IN THE RANGE TABLE.
*ERROR* EVL3: NO RANGE DEFINITION FOUND FOR RANGE 1 IN THE RANGE TABLE.
*WARNING* EVL1: THE RANGE FOR DIMENSION 2 OF "FILE1.FILE1REC" NEEDS AN UPPER
        BOUND AND HAS BEEN ASSUMED TO BE 9999.
*WARNING* EVL1: THE RANGE FOR DIMENSION 2 OF "FILE1.FILE1REC" NEEDS AN UPPER
        BOUND AND HAS BEEN ASSUMED TO BE 9999.
*WARNING* EVL1: THE RANGE FOR DIMENSION 3 OF "FILE2.FILE2REC" NEEDS AN UPPER
        BOUND AND HAS BEEN ASSUMED TO BE 9999.
*WARNING* EVL1: THE RANGE FOR DIMENSION 3 OF "FILE2.FILE2REC" NEEDS AN UPPER
        BOUND AND HAS BEEN ASSUMED TO BE 9999.
*ERROR* RTB1: NO IMPLICIT RANGE HAS BEEN FOUND FOR RANGE NUMBER 1 IN THE RANGE
        TABLE. PLEASE CHECK RANGE DEFINITION(S).
   *STATISTICS: 10 WARNING(S) AND 7 ERROR(S) DETECTED IN SEMANTIC ANALYSIS*

   *-* JOB ABORTED DUE TO THE ERROR(S) NOTED ABOVE.
```

RANGE SET NUMBER 1, (also referred to as RANGE NUMBER 1 or RANGE 1), can be found through looking at the RANGE TABLE in the back of the listing. Looking it up there, it was found that this was the range associated with the Target file. The subscripts for the 2 Source files were automatically given ranges which end at end-of-file, but the system did not know how to determine the range of the Target file.

As a next step, the non-programmer tried to add an END statement for the output structure, 'END.COMBOREC(C)=ENDFILE.FILE1REC(F1)&ENDFILE.FILE2REC(f2);'. This cleared up the compiler error and a PL/I program was generated, however, upon trying to execute the generated program, the PL/I went into a loop and produced no output records.

3.3.3. SPECIFICATION - SECOND FORM - SUBLINEAR INDICES

Clearly, the problem required a higher degree of definition. The Source file dimensions needed to be constrained, and their connection to the Target nedded to be made more explicit. The reasoning was as follows: Whenever there is a specific pattern desired between different structure, whenever the incrementing of subscripts needs to be explicitly controlled, then through the facility of sublinear indices this will readily be accomplished.

Sublinear indices allow you to express the relationships between structures with different ranges in much greater detail. It was clear from the first attempt that the system was unable to determine the exact ordering of events to solve the problem. More information was required.

The non-programmer needed to define the relationship between Source and Target so the compiler would know how to perform proper sequencing.

There are two ways to approach this. The first involves expressing the range of the Target file as a sublinear of the two Source file indices. In this case, the subscripts would be F1, F2 and C(F1,F2). The problem with this approach is that if we take the case where the Records of FILE1 are lower in key than the current record of FILE2, then there is no way to tell the MODEL system that it should increment the F1 subscript and hold F2 constant since they are global subscripts and not sublinear.

The second approach involves expressing the ranges of the Source files as a sublinear of the Target file index. When the keys are equal, both will be incremented; when one is lower than the other, we shall only increment the former.

Consequently, the specification was modified as follows:

```
Module: Match;
Source: File1,File2;
Target: Combine;

1  File1 is file,
  2  File1rec (*) is record,
     3  key1 is field (char 2),
     3  info1 is field (char 20);

1  File2 is file,
  2  File2rec (*) is record,
     3  key2 is field (char 2),
     3  info2 is field (char 20);

1  Combine is file,
  2  Comborec (*) is record,
     3  keyC is field (char 2),
     3  info1c is field (char 20),
     3  info2c is field (char 20);

(c) are subscripts;
f1(c)=if c=1 then 1 else if key1(f1(c-1))<= key2(f2(c-1)) then f1(c-1)+1
                                                          else f1(c-1);

f2(c)=if c=1 then 1 else if key2(f2(c-1))<= key1(f1(c-1)) then f2(c-1)+1
                                                          else f2(c-1);

keyc(c)= if key1(f1(c))<=key2(f2(c)) then key1(f1(c)) else  key2(f2(c));
```

```
info1c(c)= if key1(f1(c))<=key2(f2(c)) then info1(f1(c)) else 'NFOUND';

info2c(c)= if key2(f2(c))<=key1(f1(c)) then info2(f2(c)) else 'NFOUND';
end.comborec(c)=endfile.file1rec(f1(c)) & endfile.file2rec(f2(c));
1 f1 is field (dec (7));
1 f2 is field (dec (7));
```

(The END statement for the Target file remains. Also, two data declarations for the sublinear indices have been added. These were optional).

First note that F1 and F2 have been removed from the subscript statement. They are now Interim variables with the same range as the Target record. (The end of the Source record range is still automatically generated as ENDFILE).

Look at the equation for F1(C).

```
f1(c)=if c=1 then 1 else if key1(f1(c-1))<= key2(f2(c-1)) then f1(c-1)+1
                                                           else f1(c-1);
```

Put into words, the statement says: If we are at C=1, then read the first record, otherwise, if the prior FILE1 key (indicated by a subscript of C-1) is less than or equal to the prior FILE2 key, then we wish to increment the sublinear, (i.e. increment the sublinear). Otherwise, we must leave the current record for later.

The second file is given a similar sublinear statement.

The target file Fields are set up with the following rationale: When the keys are equal, all info is present. On a greater than condition, (i.e. the ELSE condition for less than or equal), there will be a 'not found' condition. (Refer to statement below for INFO1C).

```
info1c(c)= if key1(f1(c))<=key2(f2(c)) then info1(f1(c)) else 'NFOUND';
```

This is ensured by the fact that the reading of records will be synchronized by the sublinears.

As it turns our, the desired result was achieved. The following target file was produced by the above specification.

```
Target File

Key          Info1          Info2

01           John           Not found
02           Not found      Susan
03           Jim            Not found
04           Not found      Trisha
05           Harold         Not found
06           Bill           Louise
07           Tom            Ruth
                            Not found
```

3.3.4. SPECIFICATION - THIRD FORM - ENDFILE CONDITIONS

Everything was working as planned, except there seems to be a problem with the last record. Instead of "08 Not found Mary" which was expected, the program produced a record of " Not found".

Immediately, the non-programmer pinpointed the problem as stemming from an inability on the part of the system to determine values for the target file variables at a point where one file had reached end-of-file before the other.

With the test data which had been set up, FILE1 reaches end-of-file after key '07' is processed, while FILE2 has one more record.

Donning his Sherlock Holmes cap, he realizes that the ENDFILE condition will have to be explicitly added to conditions. What seems to be happening is that after end-of-file, the key field associated with FILE1 is being given a value lower than the key in FILE2, and this is causing incorrect results.

It is therefore insufficient to simply use the keys comparison as a basis for determining the value of the field. How must the equations be modified? Let us consider the equation for KEYC:

```
keyc(c)= if key1(f1(c))<=key2(f2(c)) then
                        key1(f1(c)) else  key2(f2(c));
```

We know that we must test for ENDFILE.FILE1REC(F1(C)) to prevent what is currently happening. Is this sufficient?

```
keyc(c)= if key1(f1(c))<=key2(f2(c))  & ^endfile.file1rec(f1(c)  then
                        key1(f1(c)) else  key2(f2(c));
```

Well, it may solve the current problem. But the eventuality of FILE2 reaching end-of-file first must also be guarded against. In that case KEYC would incorrectly be given KEY2(f2(c)) instead of KEY1(f1(c)).

Therefore the correct logic would be:

```
keyc(c)= if endfile.file2rec(f2(c)) |
            (key1(f1(c))<=key2(f2(c)) & ^endfile.file1rec(f1(c)))
                                  then key1(f1(c)) else  key2(f2(c));
```

Modifying the equations to incorporate the ENDFILE conditions as above, the specification was once again sent through the system with the following results:

```
Target File

Key          Info1          Info2

01           John           Not found
02           Not found      Susan
03           Jim            Not found
04           Not found      Trisha
05           Harold         Not found
06           Bill           Louise
07           Not found      Ruth
                            Not found
```

3.3.5. SPECIFICATION - FINAL FORM

At first glance, the non-programmer felt that the prior modification had had no effect. Upon looking more carefully, it was noticed that this time, the '07' record is getting a 'Not found' condition from INFO1. this indicated that the ENDFILE condition was negating the key comparison logic.

When the ENDFILE condition is used, it is sometimes necessary to process the last record of a structure first and only go down the logical path of ENDFILE after this processing has been completed. In this case, through using a 'C-1' instead of 'C' as a subscript to the sublinear for the ENDFILE statements, the specification was corrected and the results were as desired.

```
Module: Match;
Source: File1,File2;
Target: Combine;

1  File1 is file,
  2  File1rec (*) is record,
     3  key1 is field (char 2),
     3  info1 is field (char 20);

1  File2 is file,
  2  File2rec (*) is record,
     3  key2 is field (char 2),
     3  info2 is field (char 20);

1  Combine is file,
  2  Comborec (*) is record,
     3  keyC is field (char 2),
     3  info1c is field (char 20),
     3  info2c is field (char 20);

(c) are subscripts;
f1(c)=if c=1 then 1 else if key1(f1(c-1))<= key2(f2(c-1)) then f1(c-1)+1
                                                        else f1(c-1);

f2(c)=if c=1 then 1 else if key2(f2(c-1))<= key1(f1(c-1)) then f2(c-1)+1
                                                        else f2(c-1);

keyc(c)= if endfile.file2rec(f2(c-1)) |
             (key1(f1(c))<=key2(f2(c)) & ^endfile.file1rec(f1(c-1)))
                                 then key1(f1(c)) else  key2(f2(c));

info1c(c)= if endfile.file2rec(f2(c-1)) |
             (key1(f1(c))<=key2(f2(c)) & ^endfile.file1rec(f1(c-1)))
                                 then info1(f1(c)) else 'NFOUND';

info2c(c)= if endfile.file1rec(f1(c-1)) |
             (key2(f2(c))<=key1(f1(c)) & ^endfile.file2rec(f2(c-1)))
                                 then info2(f2(c)) else 'NFOUND';
end.comborec(c)=endfile.file1rec(f1(c)) & endfile.file2rec(f2(c));
1 f1 is field (dec (7));
1 f2 is field (dec (7));
```

At this point, the non-programmer added the actual detail calculations and real field names desired and the specification and its generated program became a part of their system.

PART II: MODEL STATEMENTS AND METHODOLOGY

4. COMMONLY USED LANGUAGE ELEMENTS

4.1. INTRODUCTION

This Chapter will tell you about the basic elements you can put together to write a MODEL specification. Here we will discuss the use of characters, operators, variable names, constants, expressions, and functions. We assume that you already know how to get access to the MODEL system. You can use a text editor to write your specification as an input file. (When using a text editor, it is a good idea to build your specification in a file which has a name which matches your module name, for example, SUMMARIES.INP on a VAX system. In this way, you will have consistency between the module name and the external name.) You also have the option of using other methods of input, such as tape or cards. The individual statements you write are made up of the basic elements to be described in this chapter.

4.2. EBNF NOTATION

In this Section we will explain EBNF (Extended Backus-Nauer form), which is the notation we will be using to describe the syntax of statements. EBNF is used to construct syntax diagrams which give the allowable relationships between syntax elements. In the syntax diagrams, elements consist of words which are combined to form clauses and expressions and are then combined to form statements. We will use syntax diagrams in this Chapter to define the commonly used language elements and in subsequent Chapters to give the syntax of MODEL data declaration and equations.

Syntax diagrams in EBNF employ a set of symbols which are used to express how one syntax element is defined in terms of another. They also indicate how the parts of a statement may be ordered, when elements are optional, and when they may be repeated. The following set of EBNF symbols will be used in syntax diagrams defining MODEL language elements. Please review this carefully, as it will make it possible for you to understand the significance of the diagrams which follow later:

```
1)  ::=    means that the left hand side "is defined by" the right
           hand side.
2)  [...]  means that the enclosed is optional.
3)  |      means that the immediate elements on both sides are
           alternatives, either of which can be used
4)  [...]* means that the enclosed repeats zero or more times.
5)  <...>  means that the enclosed will be defined in another
           statement in EBNF notation on the left hand side of ::= .
```

The following is an an example of a syntax diagram:

```
1  <MODEL specification> ::= [<statement>;]*
   2 <statement> ::= <header statement> |
                     <data declaration statement> |
                     <equation>
```

The first line of the syntax diagram indicates that each MODEL specification is composed of any number of statements, each ending with a semi-colon. The second line indicates that a statement may be one of three types: a header statement (MODULE, SOURCE, and TARGET), a data declaration statement (FILE, GROUP, RECORD, and FIELD), or an equation (simple and conditional).

4.3. MODEL CHARACTER SET

The character set used by the MODEL language consists of 82 characters, made up of characters, digits, and delimiters, which are listed in Figure 3.1. Words are either "reserved words," that is, words that have a meaning defined by the system (e.g., FILE) or they are names of variables (e.g., TOTAL). Letters used in words in MODEL statements may be in either upper or lower case, however, for the purpose of this text, reserved words are represented in upper case letters. Characters, consisting of the letters, $, and _ can be used in MODEL variable names. The numeric digits 0 to 9, can be used in variable names, in any but the first character. They are also used in declaring FIELD data types. Delimiters, consisting of special characters, are used to separate words and expressions in MODEL statements. Other characters, which are not part of the MODEL language such as ", ?, %, or # cannot be used in MODEL variable names or statements, but they can be used literally in character string constants or variables, as described in Sections 4.7 and 4.6.

```
<character> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|$|__|
                a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<digit> ::= 0|1|2|3|4|5|6|7|8|9
<delimiter> ::= .|<|(|+|&|||*|)|;|^|-|/|,|>|'|:|=|(BLANK)
<non-MODEL character> ::= "|?|%|#
```

Figure 4.1
MODEL Character Set

4.4. DELIMITERS

Delimiters, (line 3 in above), are used to separate one word or one expression from another, that is, to show where one ends and the next begins.

All MODEL statements end with a semi-colon. Usually, you write one statement per line, but a complex statement can run over several lines. Alternatively, several short statements can be put on a single line. A common syntax error is to omit the semi-colon at the end of one of your statements. This error, while common, is difficult for the MODEL compiler to resolve automatically, so it's important to remember to put a semi-colon, ';', at the end of each of your statements.

Blank spaces are used to separate words. Extra spaces and blank lines are ignored.

In MODEL commas serve to separate elements in a series, variable names, subscripts, or function arguments. You can write comments, which are notes to yourself or others that have no effect on the system, by beginning your comment with /*, slash-asterisk, and ending it with */, asterisk-slash as follows:

```
1  <comment> ::= /*<text>*/
   2  <text> ::= [<any character>]*
      3  <any character> ::= <character> | <digit> | <delimiter> |
                             <non-MODEL character>
```

Documenting your specification by writing comments to explain variable names or the purposes of equation statements is a good technique, especially if other people will use a specification that you have written.

Parentheses are used to indicate that a list of elements or an expression is to be treated as a single element relative to the elements outside the parentheses (see Section 4.9.2 for more details and examples). Specific uses of parentheses in different types of MODEL statements are explained as each statement is discussed in later chapters.

The use of most of the other delimiters will be explained in Section 4.8 on operators.

4.5. VARIABLE NAMES

Variable names can be up to 31 characters long. They must begin with a letter from the alphabet or the character $ (dollar sign), but after that, they can include the numbers, and the character '_' (underscore) as well. Therefore, the syntax diagram for variable names is as follows:

```
<name> ::= <character> [<character> | <digit>]*
```

Only the leftmost 31 characters of a variable name will be recognized by MODEL; the rest will be disregarded. You can make up the names of the variables as you please, but it's a good idea to make up names that are good mnemonics. For example, ITEM would be a good name for an element in a stock inventory.

The '_' is used to join parts of a name, instead of a hyphen (which would be confused with a minus sign if it were used). If you want a variable name to consist of more than one word, then you should connect the words with '_', as in OUR_ITEM.

4.6. QUALIFIED NAME VARIABLES AND RESERVED WORDS

The period is used to join names to form a qualified name variable. A qualified name variable consists of a variable name, preceded by one or more special prefixes. The syntax of a qualified name variable is as follows:

```
1 <Qualified name variable> ::= <prefix>.[<prefix>.]*<name>
  2 <prefix> ::= <keyword prefix> | <parent name>
       3 <keyword prefix> ::= SIZE | END | LEN | NEXT | POINTER | FOUND |
                              SUBSET | FOR__EACH | MALDATA | ENDFILE | EMPTY
    3 <parent name> ::= <name> | OLD | NEW
```

A qualified name variable can be formed by preceding a variable with a keyword prefix (which is a "reserved word" and may not be used as a variable name).

Qualified name variables with keyword prefixes are also called *control variables*. They allow you to define and refer to specific variable attributes, such as a range of a dimension. For example, if ITEM is the name of a FIELD variable, then NEXT.ITEM is a qualified name variable that refers to the corresponding FIELD in the next record. LEN.NEXT.ITEM referring to the length of NEXT.ITEM is also a legal qualified name, showing that a single variable can have more than one keyword prefixes. The qualified variables generally have the game shape as the variable named in the suffix, i.e. the same number of dimensions with the same respective ranges. the one exception to this rule is the SIZE prefixed variable which has at least one dimension less than its suffix. For a quick summary of the use of keyword prefixes see Figure 4.2.

EMPTY.X	Denotes whether file X is empty (no records)
ENDFILE.X	Denotes whether the record X is the last in the respective file.
END.X	Denotes the condition of the last element at at the end of the rightmost dimension of the variable X. It is a two value Boolean variable.
FOR__EACH.X	Denotes global subscript with same range as the lowest, rightmost dimension of variable X.
FOUND.X	Denotes whether a RECORD X exists in the ISAM FILE with same KEY as given by POINTER.X. Two value Boolean variable.
INITIAL.X	Denotes the initial value of the suffix variable X to be used in an iterative solution of simultaneous equations where X is defined.
LEN.X	Denotes the length of a FIELD X
NEXT.X	Denotes FIELD X in the next sequentially ordered RECORD in a SOURCE FILE.
POINTER.X	Denotes a KEY FIELD that references RECORD X in an ISAM FILE
SIZE.X	Denotes the size of the range of the lowest, rightmost dimension of variable X.
SUBSET.X	Denotes whether record X is included in forming a TARGET file. Two value Boolean variable.
MALDATA.X	Denotes whether a conversion error occurred when reading a field of source record X. Two value Boolean variable.

FIGURE 4.2
Use of MODEL Keywork Prefixes

A qualified name variable can also be formed by preceding a variable with OLD, NEW or a filename. This form has the purpose of eliminating ambiguity. If you give a variable used in an equation a prefix of OLD or NEW, it will distinguish between the same FIELD in a SOURCE FILE and its updated TARGET version (when both FILES and both FIELDS have the same name). An example is the equation

```
NEW.BALANCE = OLD.BALANCE + DEPOSIT;
```

The second way qualified names are used to eliminate ambiguity is to give a variable a prefix of the name of the FILE in which it is a member if the same variable name is used in two different FILES. (However, you cannot use the same name for two variables in the same file.)

Consider the case where the same FIELD variable, ITEM, is contained in both a SOURCE FILE, IN, and a TARGET FILE, OUT. (IN and OUT are FILE names, not keywords.) For unambiguous reference in your equations, the name of this FIELD could be preceded by the name of the FILE it came from as in

```
OUT.ITEM = IN.ITEM + SOMETHING;
```

4.7. CONSTANTS

A constant is a string of characters or numbers appearing in equations. You write constants directly as operands in your equations. This means that they are fixed, in the sense that their values don't depend on the external given data. MODEL recognizes three types of constants: character string, bit string, and arithmetic.

4.7.1. CHARACTER STRING CONSTANTS

Character string constants are defined as follows:

```
1  <character string constant> ::= '<any character> [<any character>]*'
   2  <any character> ::= <character> | <digit> | <delimiter> |
                          <non-MODEL character>
```

A character string constant is formed by enclosing a string of any characters and of any length in apostrophes, for example, 'JON'. Character strings can be used with any operators or functions that work on character strings, such as the concatenation operator, || (next Section), or the function SUBSTRING (Section 3.10)

4.7.2. BIT STRING CONSTANTS

A bit string constant is similar to a character string constant, but it contains characters listed below. It is defined as follows:

```
1 <bit string constant> ::= '<bit> [<bit>]*'B[<n>]
  2 <bit> ::= 0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F
  2 <n> ::= 1|2|3|4
```

A bit string constant can be of any length and should be enclosed with apostrophes. To distinguish it from a character string constant, bit string constant is followed by a capital B, as '101011'B.

MODEL accepts four forms of bit string constants. These are strings expressing values in base 2, base 4, base 8, or base 16. Bit string constants are used in logical and string expressions (Sections 4.9.4 and 4.9.5). To show that a bit string constant is in base 2, you would follow the string with a B or B1. A base 2 bit string constant may only contain 1's and 0's. This is the form of bit strings which is most common. (A base 4 bit string constant is indicated by following the string with a B2, base 8 with a B3 and base 16 with a B4.

4.7.3. ARITHMETIC CONSTANTS

Arithmetic constants are used in arithmetic operations and are written using a string of digits. They are defined as follows:

```
1    <arithmetic constant> ::= <unsigned number> [<exponent>]
     2  <unsigned number> ::= <unsigned integer> [<fraction>] |
                              <fraction>
        3  <unsigned integer> ::= <digit> [<digit>]*
        3  <fraction> ::= .<unsigned integer>
     2  <exponent> ::= E[<sign>] <digit> [<digit>]
        3  <sign> ::= + | -
```

Arithmetic constants are always decimal (not binary). They can have a fractional component (also decimal) and can be expressed in exponential notation which is a shorthand way of expressing numbers which are very large or very close to 0. In exponential notation the unsigned number part of an arithmetic constant (called the mantissa) is multiplied by 10 raised to the power of the exponent. The exponent must be an integer. In the Vax version of PL/I, the exponent cannot have an absolute value higher than 34 (see Section 4.6 for more information). The "E" preceding the number means "10 to the". Examples of arithmetic constants are 10011, 503.64, and 9.45E-23. (To write 9.45E-23 without exponential notation, you would have to put 22 zeros between the 9 and the decimal point at the beginning of the fraction.)

As you will see in the description of arithmetic expressions, arithmetic constants can also be given a positive or negative sign in front of them. Arithmetic expressions and the operators used in them will be discussed in detail in the next two sections of this chapter.

EXPRESSION	OPERATORS	OPERAND
ARITHMETIC	ARITHMETIC	ARITHMETIC (Decimal,Binary, Numeric String,)
	LOGICAL	LOGICAL
STRING	STRING (Concatenation)	STRING (Character,Bit)
BOOLEAN	LOGICAL	BIT, COMPARISON EXPRESSION
COMPARISON	COMPARISON	ANY EXPRESSION

Types of Expressions
Figure 4.3

4.8. OPERATORS

Certain of the MODEL delimiters, called operators, indicate arithmetic, logical, or string operations. These symbols are used with variables, called operands, to create expressions. There are restrictions on which operator can go with which kind of operand (see Figure 4.3). These restrictions have to do with the idea of data types, (Section 5.6) It does not make sense to divide one character string by another character string. And although it may make sense to "add" two strings, by stringing them end to end, MODEL uses || for this rather than +. This operation is actually called *concatenation*. When you declare variables in your specification, you also indicate the data type of each variable. If you try to use an operator with the wrong type, for example, a + between two variable names that stand for character strings, you will get a conversion error message (Section 4.11). The conversion is however performed automatically if it is feasible, namely if the character string consists only of numbers.

The following symbols are the *arithmetic operators* which can be used with any arithmetic variables, constants, or functions that are decimal or binary. Arithmetic operators are used in arithmetic expressions (next Section).

```
+   addition, or a prefix indicating a positive number;
-   subtraction, or a negative number;
*   multiplication - NOTE: This MUST be used for multiplication.
                           Unlike algebra, you do not indicate
                           multiplication by no symbol at all.
/   division
**  exponentiation or raising to power - For example, 2**3 means
                           2 to the third power, or 8.
```

In addition to arithmetic operators, there are the following *comparison operators*, which are used in Boolean expressions (see next Section):

```
>    greater than
<    less than
=    equals
^>   not greater than
^<   not less than
>=   greater than or equal to
<=   less than or equal to
^=   not equal to
```

The equals sign '=' is also used in MODEL equations to mean "is defined as".

Another set of operators, called *logical operators*, can be used in logical or Boolean expressions.

```
^    not
&    and
|    or
```

These operators work on Boolean, or bit string variables (including binary). A Boolean variable has a value of either true or false represented by a 1 or a 0. In Boolean expressions, these operators work on single Boolean variables, while in logical expressions, these operators work on bit strings and binary variables (see Sections 5.7.2 and 5.9.4), which are equivalent to several Boolean variables strung together.

Parentheses, '(' and ')', are called group operators and can be used in any type of expression and with any data type. They usually surround one expression which is contained within a second expression. They mean that the expression they surround is to be evaluated separately and treated as a unit in evaluating the second expression.

The symbol || is called the concatenation operator or string operator. It is used with bit strings and character strings in string expressions. It means "string together" or place end to end. Other operations performed on strings use reserved functions instead of operators, and we shall discuss these later.

4.9. EXPRESSIONS

Expressions are combinations of operators, functions, constants, and variables which act like clauses or phrases in a sentence to express a partial value. Expressions can also be combined to form larger expressions. The major types of expressions used in MODEL equations are arithmetic, logical, string, and Boolean. Comparison expressions are a subclass of Boolean. The types of expressions, along with their preferred operators and operands were listed in Figure 4.3. It is also possible to write expressions containing mixed operands. Depending on the particular case, one of the operands may be converted by the PL/I compiler, so that the expression can be evaluated, or you may get an error message. (Section 4.11 describes the treatment of expressions with mixed operands by the MODEL system).

4.9.1. USE OF OPERATORS IN EXPRESSIONS

Expressions containing operators are evaluated in the following order of priority, based on the operators they contain:

```
 1.  ( ) (expressions enclosed in parentheses)
 2.  functions
 3.  expressions used as subscripts (subscript expressions)
 4.  + (as a prefix, meaning a positive number)
     - (as a prefix, meaning a negative number)
     ** (exponentiation)
     ^ (logical not)
 5.  * (multiplication)
     / (division)
 6.  + (addition)
     - (subtraction)
 7.  || (concatenation)
 8.  = (equal to)
     > (greater than)
     < (less than)
     ^> (not greater than)
     ^< (not less than)
     ^= (not equal to)
     >= (greater than or equal to)
     <= (less than or equal to)
 9.   & (logical and)
10.   | (logical or)
```

In this table, operators with higher priority (lower numbers in the table) are applied first in evaluating expressions. Thus, in A * B + C, A and B are multiplied before C is added. You can tell this because * is priority 5 and + is priority 7. Similarly in ^D || E, logical not, ^, is applied to bit string D before it is concatenated to bit string E, because ^ is priority 4 while || is priority 7. All the operators under the same number have the same priority. If several operators of the same priority are present in the same expression, then the operators will be applied sequentially from left to right, except for exponentiation, which is applied from right to left, and parentheses. (See next section)

4.9.2. PARENTHESES IN EXPRESSIONS

Expressions can be combined to form larger expressions. At the end of Chapter 2, we discussed structuring data in terms of trees where nodes can keep branching into new nodes deeper in the tree. The same thing is true for expressions; the various types of expressions can include expressions as basic elements. For example, the syntax diagram for arithmetic expressions (Section 4.9.3) shows that an arithmetic expression, surrounded by parentheses, can take a sign, be raised to a power by an exponent, etc. These nested expressions could contain other expressions, which contained still other expressions, and so on.

You show that expressions are nested in other expressions by enclosing the inner expression in parentheses. By creating embedded expressions, you can change the order that operators will be applied in evaluating your expressions. This is because parentheses have the highest priority as operators. For example, in the expression A + B * C, according to the above order of operations, the multiplication is done before the addition. However if you rewrote the expression using parentheses to form (A + B) * C, the parentheses would cause the addition in the inner expression to be done first.

Parentheses can also be used to change the order that expressions containing logical and string operators are evaluated. For example, to apply a logical not, ^, to two strings after they have been concatenated, you should enclose the two strings and the concatenation operator in parentheses as in ^(A || B). If you use several sets of parentheses inside each other, the expression embedded within the innermost parentheses will be evaluated first, then the second innermost, and so on. You can also use parentheses to improve the readability of expressions by setting off related elements together, without changing the value of the expression, as in ((A**3)/B) - ((X**4)/Y).

4.9.3. ARITHMETIC EXPRESSIONS

The basic components of arithmetic expressions are arithmetic constants, variables, functions, other arithmetic expressions, as well as the various arithmetic operators. Any variables or functions used in arithmetic expressions, must have interpretable values which are numeric. They may be of decimal, binary, numeric string, or picture data types. Subscript variables are a special class of arithmetic variables. (Arithmetic expressions used as subscripts, which may contain subscript variables, are called subscript expressions. These are described in Chapter 7.)

The syntax diagram defining arithmetic expressions is as follows:

```
1  <arithmetic expression> ::= <arithmetic term> [<addition operator>
                               <arithmetic term>]*
   2  <addition operator> ::= + | -
   2  <arithmetic term> ::= <arithmetic factor> [<multiplication operator>
                            <arithmetic factor>]*
      3  <multiplication operator> ::= * | /
      3  <arithmetic factor> ::= <arithmetic primary>
                                 [** <arithmetic primary>]*
         4  <arithmetic primary> ::=   [<sign>] <arithmetic element>
         4  <sign> ::= + | -
          5  <arithmetic element> ::= <arithmetic constant> | <arithmetic
                                      variable> | <subscript variable> |
                      arithmetic function> |
                                      (<arithmetic expression>)
```

Operations in arithmetic expressions apply to both decimal and binary variables, functions, and nested expressions. However, as described in Section 3.7.3, arithmetic constants must be decimal.

The following are examples of arithmetic expressions:

```
4
ELEPHANT + 3
JACK * QUEEN
PAUL + DAVID * ANN/(LUKE - 4)
-12E-15
```

```
SIN(A**-2.75) + COSD(B**((4.79 +C) * 5.79 - D)
```

4.9.4. LOGICAL EXPRESSIONS

The syntax diagram of logical expressions is as follows:

```
1  <logical expression> ::=  <logical term> [| <logical term>]*
   2  <logical term> ::= <logical factor> [& <logical factor>]*
      3  <logical factor> ::= [^]<logical primary>
         4  <logical primary> ::= <bit string constant> | <bit string
                                  variable> | <binary variable> |
                  <logical function> | (<logical expression>)
```

The operands of logical expressions are bit strings (including functions, constants, and variables), binary variables, and nested expressions. (Bit string constants with a base other than 2 are represented internally as base 2 strings in logical expressions.) Each 1 or 0 in these operands corresponds to a separate Boolean variable. (In essence 1 means true and 0 means false.) The logical operators &, |, and ^ can be used with operands of any length; & and | take two operands, while ^ takes one. The result of any of the above operations is always a single bit string.

The logical operators, and, &, and or, |, work on individual bits in the same respective positions from the right ends of two logical operands. When these operators are applied to operands of different lengths, 0's are added to the right of the shorter operand until the operands are of the same length. Thus, the length of the result will be the length of the longer operand.

If the logical operation is & and if the corresponding bits in the two operands are both 1's (true), then the resulting bit in the new string is a 1; otherwise it is a 0 (false). For the | operation, if one or both of the bits in the two operands is a 1, then the resulting bit is also a 1; otherwise it is a 0. For example, if X is '10110'B and Y is '11000'B, then X & Y is '10000'B and X | Y is '11110'B.

The logical not operator, '^', reverses the truth value of what it precedes; it exchanges '1'B's for '0'B's and '0'B's for '1'B's in a logical primary. For example, if X is '10110', then ^X is '01001'. (On some keyboards it looks like a sideways 'L' and is usually above the "6").

4.9.5. STRING EXPRESSIONS

String expressions, which can incorporate only the concatenation operator, ||, are defined in the following syntax diagram:

```
1  <string expression> ::= <string term> [|| <string term>]*
   2  <string term> ::= <character string constant> | <character string
                        variable> | <string function> |
                        <logical factor> | (<string expression>)
```

In string expressions, character string constants, variables, functions, and nested expressions can be joined together to form longer character strings. For example, 'Learning the MODEL language' || 'is fun.' becomes 'Learning the MODEL language is fun.' Logical factors can also be strung end to end using the concatenation operator, so that '111101'B || ^'0101'B becomes '1111011010'B. Finally, you can concatenate character strings to logical factors. The end result will be a character string, with the logical factor being converted according to PL/I conventions. For example, 'WILLIAM' || '1001'B becomes 'WILLIAM1001'.

4.9.6. BOOLEAN EXPRESSIONS

Boolean expressions, like a Boolean variable, have a single value of true or false. Their syntax diagram is as follows:

```
1  <Boolean expression> ::= <Boolean term> [| <Boolean term>]*
   2  <Boolean term> ::= <Boolean factor> [& <Boolean factor>]*
      3  <Boolean factor> ::= [^] <comparison expression>
         4  <comparison expression> ::= <Boolean primary> [<comparison
                        operator> <Boolean primary>]*
            5  <Boolean primary> ::= <arithmetic expression> | <logical
                                     expression> | <string expression> |
                                     (<Boolean expression>)
            5  <comparison operator> ::=  = | < | > | <= | >= | ^= |
                                          ^< | ^>
```

Boolean expressions act like logical expressions and use the same operators. The only difference is that Boolean expressions can only have a single bit as an operand rather than bit strings of arbitrary length. Boolean expressions can use nested comparison expressions as operands, since comparison expressions have a single bit value. An example is the expression A = B | C = D & E = F, which is true when either A equals B or both C equals D and E equals F.

4.9.7. COMPARISON EXPRESSIONS

Comparison expressions can use any type of expression as operands. If the two operand expressions hold the relationship, given by the comparison operator, then the comparison expression has a bit value of true. If not, then it has a bit value of false. For example, the comparison expression A > B has a value of true only when A is greater than B; otherwise it is false. When you write a comparison expression containing more than one non-nested comparison operator, such as A < B < 10, the comparisons will be made in order from left to right. In this case the comparison expression is always true, because the comparison of A and B will result in a true or false bit (no matter what A and B are), which will be converted to a 0 or a 1 for the next comparison which will be smaller than 10.

When two expressions are compared, they should be of the same type, that is, both arithmetic, both logical, both character string, or both nested Boolean. Also remember that comparison operators have a higher priority than logical 'and' (&) and logical 'or' (|), so if you wish to compare two logical expressions containing & or |, you should enclose the logical expressions in parentheses.

When containing arithmetic constants, functions, and variables, comparison expressions follow the same principles as in arithmetic, for example, 3 >= 2 is true, while 7 < 5 is false. Comparison operations can involve bit or character strings as well as numbers. When comparison operators are applied to bit strings, if the strings are of different length, then the shorter string is first extended with 0's on the right so that both strings will be of the same length. So, when evaluating the truth of '10010'B > '10001001'B, '10010'B will be extended to '10010000'B. (This is the same as the convention for using logical operators on strings of different length discussed earlier.) The bit strings are then compared as if they were binary numbers. Therefore, using the example from above, the Boolean expression '10010'B > '10001001'B is true, because '10010000'B is greater than '10001001'B. (If the numbers 10010 and 10001001 were directly compared in binary, then 10001001 would be greater.)

Character strings are compared character by character from left to right. If the character strings are of different lengths, blanks will be added to the right of the shorter string before the comparison is made. The comparison is alphanumeric, with letters near the end of the alphabet defined as greater than letters near the beginning of the alphabet, and lowercase letters greater than uppercase. Also letters are defined as greater than numbers, and numbers are defined as greater than most punctuation. (The basis for each comparison is the ASCII code numbers of the relevant characters, and on IBM, EBCDIC conventions apply). If the first character in two strings is the same, then the second is compared, and so on. All of the following are true comparisons of character strings:

```
'ABZ' > 'ABY'
'NOAH' < 'Noah'   (This is just the opposite in EBCDIC)
'cat1' > 'cat.'
```

4.10. FUNCTIONS

There are certain tedious calculations that most users want their specifications to perform at one time or other. To save you from the necessity of having to write equations to specify these calculations, you can use functions. These functions are either built into the MODEL system or added by a programmer. To use a function, you enter the name of the function along with the variable(s) or expression(s) that you want to be operated on. An example would be the function SQRT(X), which evaluates the square root of a positively valued arithmetic expression surrounded by parentheses. X is called the argument of the function, what the function operates on. In this case, X represents an arithmetic argument, but for other types of functions, it could represent other types. More complex functions can have several arguments. The single result that a function with any number of arguments returns is called the *function's value.*

Function names in a MODEL specification are reserved words, which means they should not be used as variable names. MODEL can use any of the functions available in PL/I. Additional functions can also be written in MODEL and added to your system. Once a function is created in this way, its name also becomes a reserved word. A sample of PL/I functions, plus a list of additional functions written especially for MODEL, is contained in chapter 11.

The syntax diagram for functions is as follows:

```
1  <function> ::= <function name> [(<argument> [, <argument>]*)]
   2  <argument> ::=  <arithmetic expression> | <logical expression> |
                      <character string expression>
```

To use a function you type in the function's name followed by the variable(s) or expression(s) to be used as arguments, enclosed in one set of parentheses. If the function takes more than one argument, they are separated by commas. Some functions do not take any arguments. In that case the parentheses are omitted.

There are functions appropriate for all expressions and data types. SQRT(X), which we already mentioned, is an ARITHMETIC function which can be applied to decimal, binary, numeric string, or picture data types. Another example of an ARITHMETIC function is SUM. SUM(X(I),I) adds up all the elements in vector X(I) and returns the result as a scalar. For a variable with two or more dimensions, a SUM function can be used to add the elements along any one of its dimensions. SUM is referred to as a reduction function, because it reduces the number of dimensions of the result to one less than originally contained in the argument.

Other functions can be used on character or bit string variables. An example would be the string function SUBSTR(X1,X2[,X3]), used to extract a smaller string or substring from string variable X1. X2 gives an integral value corresponding to the position, counting characters from the left, where you want the substring to be extracted from X1 to start. X3 is optional, as indicated by the square brackets. You can use X3 to give the number of characters in the substring. If you omit it, then the substring will extend from position X2 to the end of X1.

4.10.1. CREATING USER-DEFINED FUNCTIONS

If there is a group of logical processes common between several programs, in the interests of ease and efficiency it is sometimes helpful to separate out the procedure and make it available to a large number of specifications. The user-defined function facility makes this possible.

User-defined functions have the following requirements:

```
  1- A source file with up to 10 parameters (input fields).
  2- A target file with one field (containing the result of the function).
```

Functions are defined as any other specification except that the keyword 'function' replaces the word 'module' in the header section. They must include only one source file and one target file and the target file may have only one field.

In order to get your function registered in the system, the function specification is run through the MODEL compiler to create a PL/I source program. There is a special file in the system call the user function library (UFCNLIB.DAT). This file contains PL/I source for all the functions available to your programmers in your installation. In order to add a new function, simply insert the PL/I code from the MODEL compilation of your function into the UFCNLIB file. Make sure to insert it after the end of the last function in the file. At this point the function is available to be run from any specification.

Functions are invoked from within specifications through function name (must be different than the names of any fields or other data structure elements) followed by the function parameters in parentheses separated by commas. The parameters can be in the form of literals or variables.

At execution time, the program will call the desired function and the correct value is returned.

The following is an example of a function and a specification which references it. The function returns one of the roots of a quadratic equation given the coefficients and the program writes a report.

```
Function: Quad;
Source: quadin;
Target: quadout;

1  quadin is File,
   2  quadinr  is record,
      3  AIN is field (pic 's99999'),
      3  BIN is field (pic 's99999'),
      3  CIN is field (pic 's99999');

1  quadout is File,
  2  quadoutr is record,
     3   X is fld (Decimal (7,2));

/* Interim fields for floating point calculations */
1  A is field (decimal Float(10));
1  B is field (decimal Float(10));
1  C is field (decimal Float(10));

X = (-B+(B**2-4*A*C)**.5)/2*A;
A=AIN;  B=BIN;  C=CIN;
```

<An Example of a Function, "QUAD">

```
Module: Call_fnc;
Source: Coeffic;
Target: Quadrpt;

1  Coeffic is File,
   2  Corec (*) is record,
      3  Coeff_1 is field (pic 's99999'),
      3  Coeff_2 is field (pic 's99999'),
      3  Coeff_3 is field (pic 's99999');

1  quadrpt is File,
  2  quadrec (*) is record,
     3   Literal1 is fld (char 22),
     3   Aout is field (pic 's99999'),
     3   Literal_1a is fld (char 5),
     3   Bout is field (pic 's99999'),
     3   Literal_1b is fld (char 2),
     3   Cout is field (pic 's99999'),
     3   Literal2 is fld (char (17)),
     3   Answer is fld (pic 'szzz99v.99');

Answer = Quad(Coeff_1,Coeff_2,Coeff_3);

Aout=Coeff_1;
Bout=Coeff_2;
Cout=Coeff_3;      equation       is
Literal1 = 'The coefficients are';
Literal2 = '. The answer is';
Literal_1a= 'X**2'; +
Literal_1b= 'X';
              +
                      <A specification which invokes the Function, "QUAD">
```

4.11. CONVERSION OF DATA TYPES IN EXPRESSIONS AND FUNCTIONS

As explained in Section 4.9 and 4.10, MODEL expressions and functions were designed to use certain specific data types as operands and arguments. Figure 4.4 explains how the MODEL system evaluates expressions or functions written using non-preferred data types.

LHS TYPE	OPERATOR	RHS TYPE
1,2,3,4,5,6,7	+,-,*,/	1,2,3,4,5,6,7
1,2,3,4,5,6,7	**	--------------
0,1,3,5,7	||	0,1,3,5,7
5	&,|	5
5	^	--------------
0,1,2,3,4,5,6,7	COMP.	0,1,2,3,4,5,6,7

```
COMPARISON OPERATOR =    = | ^= | > | >= | < | <=
```

0	CHARACTER STRING
1	FIXED BINARY
2	NUMERIC
3	FIXED DECIMAL
4	FLOAT BINARY
5	BIT STRING
6	FLOAT DECIMAL
7	PIC STRING

Figure 4.4

Conversion of Data Types in Expressions with Mixed Operands

For an operator in a given row of the above table, you can write a legal expression using as operands any data type mentioned in the LHS and RHS columns of the same row. When an expression uses mixed operands, the RHS operand is automatically converted into the LHS type. Therefore, the evaluated expression will also be of LHS type.

In conditional equations of the form:

```
A = IF C THEN EXP1
         ELSE EXP2;
```

C must have a bit string data type. The choice of data types for EXP1 and EXP2 obeys the above rule for comparison expressions. If different from A, the data types of EXP1 and EXP2 will be converted (See Chapter 6 for more information on conditional equations.)

Functions, as described in Section 4.10, have data types associated with their arguments and returned values. When a function is written using a non-preferred data type, the rules of automatic conversion in the "COMP" row of the table are applied. This means that argument(s) will be converted to the type(s) conventionally used with the function, so that the function can be evaluated normally.

Warning messages are issued whenever automatic conversion is applied. Error messages are issued whenever the above rules are violated.

5. DATA DECLARATION IN MODEL

5.1. OVERVIEW

This Chapter will discuss the syntax and semantics of MODEL data declaration statements. Syntax, as you recall from the previous chapter, is concerned with how the elements of a statement are defined in terms of other elements and how they are ordered. Semantics, on the other hand, is concerned with the meanings produced by choosing to use certain elements rather than others The discussion of the syntax of MODEL data declaration relies heavily on EBNF notation (Section 4.2.) Therefore, make sure you understand how these diagrams are read before you go on. In the following sections syntax rules will be applied in examples to help give you some idea about how best to use the options in data declaration to solve particular problems.

5.2. MODEL HEADER

MODEL specifications start with a program header. The header has three naming functions: It names the specification, source files, and target files. These naming functions are carried out by using different statements as expressed in the syntax diagram of Figure 5.1.

```
1 <header statements> ::= <MODULE statement> [<SOURCE FILE statement>]
                          [<TARGET FILE statement>]
  2 <MODULE statement> ::= <MODULE> : <name>;

    3 <MODULE> ::= MODULE|MOD

  2 <SOURCE FILE statement> ::= SOURCE [FILE|FILES]: <name> [, <name>]*;

  2 <TARGET FILE statement> ::=  TARGET [FILE|FILES]: <name> [, <name>]*;
```

FIGURE 5.1: Syntax of Header Statement

An example of a MODULE statement would be:

MODULE: FRED;

In this example the word MODULE is a MODEL keyword, The keyword MODULE is used to name a specification. The example tells that FRED is the name of the specification. (Each specification can have only one name.) It is important to note that all MODEL statements, end with a semi-colon.

The source file statement gives the names of source files. These serve as input files containing the data to be used in the specification. They are typically read in from external devices.

Examples of legal source file statements would be:

```
SOURCE: RALPH;
SOURCE: RALPH, NORTON;
SOURCE FILE: NORTON;
SOURCE FILES: NORTON, RALPH;
```

It is possible to create a MODEL specification that does not have source files if you define all the dependent variables in your equations in terms of constants, or variables that already have been defined in terms of constants. For example, the specification in Figure 5.2, will produce a series of 12 numbers in which each number is equal to the sum of the two previous numbers in the series. (This is known as the Fibonacci series.) It was necessary to define only the first two members of the series using constants; this provided enough information to define the others.

```
MODULE: SUM;
TARGET: OUTPUT;

        1  OUTPUT IS FILE,
           2  RECOU (12) IS REC,
              3  E IS FLD (PIC 'ZZ9');

        I IS SUBSCRIPT;

        E(I) = IF I = 1 THEN 1 ELSE
               IF I = 2 THEN 1 ELSE E(I-1) + E(I-2);
```

Figure 5.2

Example of a Specification Without a SOURCE FILE

Target files are the output from running the program produced from your specification, generally to be stored on some external device. The syntax of target file statements is similar to source file statements.

Examples of legal TARGET FILE statements would be:

```
TARGET: ALICE;
TARGET FILE: TRIXI;
```

It is possible to use the same FILE as both SOURCE and TARGET. To do this, you include the same file name in both the SOURCE FILE and TARGET FILE statements. You then only have to declare the structure of the FILE once. Later, you specify which FILE a particular variable in your equations belongs by adding the qualifying prefixes OLD for source, or NEW for target, as described in Section 4.6.

5.3. DATA DECLARATION SYNTAX

Data declaration in MODEL allows you to express the hierarchical organization of your variables. Data declaration statements express several pieces of information. They tell the name of each variable and whether it refers to an elemental unit of data, such as a field, an aggregation of fields and other units, such as a record or group, or the highest level, a file. *Two variables in the same file should not be given the same name.* Data declarations also denote the relative positions of variables in the data tree, that is, which group is above which record, etc. For variables below the level of FILE, data declaration statements may give repetition counts, with the option of leaving them unspecified. A FILE declaration may have optional information about external devices; a FIELD declaration gives the data type length and scale attributes.

The structure of a whole data tree is described in a single statement ending with a semi-colon. Variables are declared in depth-first order in the data tree, with the declaration of each variable delimited by a comma. The syntax diagram is shown in Figure 5.3.

```
1 <data declaration statement >::=

  [<level number>]<variable>[(<repetition>)][<is>]<tree level>
                             [<level argument>][,]

  [<level number><variables>[(<repetition>)][is>]<tree level>
                             [<level arguments>][,]]*;

  2 <level_number>::=<unsigned integer>

  2 <variable>::= <name>
                | (<name> [,<name]*)

    3 <name>::= <character> [<characters)<digit>]

  2 <is> ::= <character> [characters)<digit>]

  2 <tree level>::= FILE|GROUP|GRP|RECORD|REC|FIELD|FLD

  2 <level argument> ::=<FILE argument>
                      |<[(]<data-type definition> [)]

    3 < repetition> ::= *
                      | < unsigned integer >
                      | < min repetition >:< max repetitions>
      4 < min repetition> ::= < unsigned integer >
      4 < max repetition> ::= < unsigned integer >
```

Figure 5.3: Syntax of Data Declaration Statement

A data declaration statement consists of one or more phrases, one phrase for each variable in the data tree. The phrases must be ordered depth first, left to right, according to the position of the respective variable in the data tree. Each phrase consists essentially of four parts. The level number gives the distance from the apex of the tree. For interim data, the level number may be omitted if only the terminal leaf of the tree is declared. The rest of the tree is then filled automatically. (For non-interim data, the level number is mandatory. The level number is followed by the name of the variable. A variable name is followed by a repetition count. The repetition part must be omitted if the variable does not repeat (has one occurrence only). If a non-one repetition count is given then this means that the variable is a vector. It may be an element in a vector parent thus giving it two dimensions, etc. The number of repetition gives the range of the respective dimension. If it is to be determined based on the data itself, this is denoted by using an * or giving a minimum and maximum. If this option is used it will be necessary to define the range by an equation.

This is followed by a choice of keywords IS, ARE, or =, followed by indicating the tree level, which is denoted by choice of a keyword: FILE, GROUP, RECORD or FIELD. FILE is the apex of the tree. RECORD is a structure that is communicated as one unit to or from external devices, such as a record in a database or a line to the printer. GROUP is any other non leaf node in the tree and FIELD is the leaf node. Finally this is followed by an argument for a file-node describing its organization, and for a field-node describing its data type length and scale. This completes the phrase for each variable. All phrases, except the last one, are terminated by a comma (,), with the last one terminated by a semicolon(;).

Figure 5.4 contains an example of data declaration syntax. Indenting of lines is not necessary, but can probably help you keep better track of the relationships between your variables. Subsequent sections describe in further detail the semantics of each of the above described phrases.

```
1 SCORE IS FILE,
  2 TEST__SCORE(3) IS RECORD,
     3 STUDENT(12) IS FIELD (PIC '99');

1 STAT IS FILE,
   2 TEST__STAT(3) IS RECORD,
     3 (MEAN_TEST,STD_TEST) IS FIELD (PIC 'ZZZ9V.9999');

1 INT IS FILE,
  2 INT__TEST(3) IS GROUP,
    3 (M__TEST,S__TEST)(12) IS FIELD (PIC '9999V.9999');
```

Figure 5.4

DATA DECLARATION SAMPLE

5.4. FILE DECLARATION STATEMENTS

All external data must be declared as part of Source or Target files. Source, Target, and Interim files are all declared the same way. Figure 5.5 shows the syntax for the File declaration phrase shown in Figure 5.3).

A FILE declaration phrase gives the name of the File and optionally describes the File's organization. A File's organization refers whether the file uses a sequential access method (SAM) or an indexed sequential access method (ISAM).

A SAM file cannot be referenced by key, i.e. the computer has to search from the beginning of the file, record by record, until the appropriate record is found.

An ISAM (or VSAM) file may be referenced by key. Keys in ISAM files act like catalogue numbers in a library which allow particular books to be found without having to search through all the shelves (see below).

A MAIL file is communicated to another specification without intermediate storage on an external device.

A POST file is similar to MAIL, only it includes the address of its destination as a KEY.

Figure 5.5 shows the syntax of a FILE phrase:

```
1  <FILE declaration phrase> ::= 1 <variable> [<IS>] <FILE>
                                            <FILE argument>;
   2  <FILE> ::= FILE | FILES
   2  <FILE argument> ::=

      3  <FILE description> ::= [KEY  [<IS>]  name>]  [ORG[ANIZATION]
                                             [<IS>] <TYPES>]
         4 <TYPE> ::= SAM|SEQUENTIAL|ISAM|MAIL|POST
```

Figure 5.5

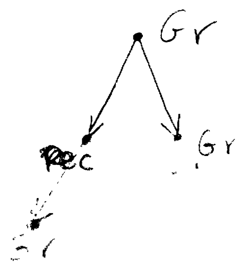
Syntax of the FILE Declaration Phrase

5.5. GROUP AND RECORD DECLARATION PHRASES

Groups and Records are the intermediate level of the data tree, with substructures below them. Records are the unit of physical transfer of information between internal and external data storage, while Groups are intermediate data structures. Groups can be above or below Records in the data tree. A Group of Records represents one or more units of physical transfer of data. Groups below Records represent logical subdivisions of the Record.

Groups can be above or below other Groups, for example

```
5 TONY (7) IS GROUP,
  6 MARIA IS GROUP
     7 ... FIELD
  6 JETS(3) IS GROUP,
     7 ... FIELD;
```



Each FIELD or piece of data, except for interim Fields, must have only one Record above it in the File. (Interim Fields need not be in Records).

5.6. FIELD DECLARATION STATEMENTS AND DATA TYPES

FIELDS are the parts of the data tree which hold the values of individual pieces of data. Each FIELD contains a particular data type. In this section, we will explain the characteristics of each data type and examine how to represent them.

Fields are at the lowest level of the data tree. The FIELD declaration statement contains a <data type definition> syntax element which is unique to FIELDS (see Figure 5.6). This element gives the Field's data type. There are six main data types which can be used in MODEL Fields: character string, bit string, numeric string, decimal, binary,and picture. The decimal and binary data types further divide into fixed and floating subtypes. Each of these data types will be explained in turn.

```
1  <FIELD declaration Phrase> ::= [<level number>] <variable> [<IS>] <FIELD>
                                        [(] <data type definition> [)]
                                      [<on conversion error>];
   2  <FIELD> ::= FIELD|FIELDS|FLD|FLDS
   2  <data type definition> ::= <character string> | <bit string> |
                                       <decimal> | <binary> | <picture>
      3  <character string> ::= CHAR[ACTER] <string format>
         4  <string format> ::=  [(] <no. of elements> | <minimum no. of
                                elements> : <maximum no. of elements> [)]
            5  <no. of elements> ::= <unsigned integer>
            5  <minimum no. of elements> ::= <unsigned integer>
            5  <maximum no. of elements> ::= <unsigned integer>
      3  <<bit string> ::= BIT [(] <no. of elements> [)]
      3  <numeric string> ::= NUM [(] <no. of elements> [)]
      3  <decimal> ::= DEC[IMAL]  <fixed format> | <floating format>
         4  <fixed format> ::= [FIX[ED]] [(] <no. of elements> [)]
         4  <floating format> ::= FLOAT [(] <precision> [)]
            5  <precision> ::= <unsigned integer>
      3  <binary> ::= BIN[ARY]  <fixed format> | <floating format>
      3  <picture> ::= PIC[TURE] '[[(<no. of repetitions>)]
                           9|Z|*|Y|.|,|/|B|S|+|-|$|T|I|R|E|X|A ]*'
         4  <no. of repetitions> ::= <unsigned integer>
  2 <on conversion error>::= ON_CNVERR [:] <stop or substitute>
      3  <stop or substitute>::= STOP
                                     <constant>
```

Figure 5.6
Syntax of the FIELD Declaration Statement

Data types used in MODEL fall into two classes, printable and non-printable. Printable data types, that is, character string, numeric string, and picture, are stored in the form of conventional characters and digits.

Non-printable data types, that is, decimal, binary, and bit, are represented by the computer in a compact form taking up less room in memory. If you try to view information in this form, you will see what appears to be a mixture of random characters which you will be unable to read. However, you will want to use non-printable data types for certain applications, such as in applications with many calculations, because they will be processed more quickly than the printable data types.

Manually entered data and printed reports should use printable data types. You can use non-printable data types in interim FIELDS for calculations, and then write equations declaring the variables as non-printable and then define the latter in terms of printable target file variables, so that you will get readable output.

5.6.1. CHARACTER STRING VARIABLES

Character string variables are strings of characters. They are made up of combinations of any characters which your keyboard will print including numbers and letters. Character string variables, like character string constants, can be used with the concatenation operator (Section 4.8) and string functions (See Chapter 11) as parts of string expressions (Section 4.9).

When you declare a character string variable, you must specify its length. You have two options for doing this. One option is to declare a specific number of characters that you expect your character string variable to be, as in,

```
    10  WANDA IS FIELD (CHARACTER (10));
```

The other alternative is to enter a minimum and maximum expected length for the character string variable, as in,

```
        10  GLENDA IS FIELD (CHAR (3:7));
```

In this case you must include an equation to define the length of the character string variable using a control variable with a LEN prefix. For example you could write

```
                LEN.GLENDA = GLEN_INFO;
```

where GLEN_INFO is a numeric variable which contains the length of the field GLENDA.

```
Example 1
  NAME is field (char (10));
  NAME='Bill Blass';

In target, NAME would appear as ---Bill Blass

Example 2
  NAME is field (char (10));
  NAME='Mike';

In target, NAME would appear as ---MikeBBBBBB (where B is a blank space)

Example 3
  NAME is field (char (10));
  NAME='Bobby Jones';

In target, NAME would appear as ---Bobby Jone (result is truncated)
```

5.6.2. BIT STRING VARIABLES

Like bit string constants, bit string variables can be used with logical operators as operands in logical expressions. They can also be used in string expressions like character string variables. Unlike bit string constants, bit string variables must be base 2. Unlike character string variables, the length cannot be specified as a range. An example of a bit string declaration is

```
        10    HENRY IS FIELD (BIT (3));
```

5.6.3. NUMERIC STRING VARIABLES

Numeric string variables may be used in arithmetic expressions with arithmetic operators. Numeric string variables are unsigned integers. When they are declared, you should specify the number of expected digits, as in

```
          5  HERO IS FIELD (NUM (6));
```

```
Example 1
  INT is field (num 7);
  INT=3100;

In target, INT would appear as 0003100
```

5.6.4. DECIMAL AND BINARY VARIABLES

Decimal and binary variables are arithmetic variables, also used in arithmetic expressions with arithmetic operators. These types also allow you to use a "floating point" feature which allows your specification to handle very large or very small numbers with many digits to the right or left of the decimal point. The binary data type is preferred to the decimal when you perform complex arithmetic computations. Binary data is stored more compactly in the memory of the computer than decimal data. Arithmetic operations are more efficient using binary operands.

Decimal and binary data types can both be expressed in a fixed or floating format. In fixed format, you tell the computer how many digits you expect there to be in your data variable. For example,

```
            6  LISA (4) IS FIELD (DECIMAL FIX (5));
```

tells the computer that LISA consists of four five-digit numbers.

Since arithmetic variables store numeric values, there is a finite range of values that a variable can assume. This range is determined by two attributes of an arithmetic variable, size and scale. Together, size and scale are known as the precision of a number.

Decimal and binary data can also be expressed in floating format. In this case the computer keeps track of where the decimal point should be placed for each piece of data. In floating format, data is expressed as a number containing a certain number of digits multiplied by 10 raised to a particular power. This data type can save a lot of space in the computer's memory for very large or very small numbers.

In floating format notation, the number is called the mantissa. The number of digits in the mantissa is its precision. The precision represents the number of digits (called significant digits) that you want the computer to keep track of, and it is specified as part of the declaration of each floating point variable. You can specify up to a maximum precision of 34 for floating point decimal and 113 for floating point binary. An example of a floating point declaration would be

```
           10 STAN IS FIELD (DECIMAL FLOAT (25));
```

When doing calculations if the result has more significant digits than the precision you specified, the computer will round the result.

5.6.5. PICTURE VARIABLES

The picture data type is used for either character or arithmetic data. The presence of an 'A' or 'X' picture symbol defines a Field to be for character data; otherwise, it is numeric and may be used in arithmetic expressions with arithmetic operators.

You declare a picture variable in terms of a series of symbols through which you specify what characters will be allowed in each position in the variable. A Z means that a leading zero should be omitted and printed as a blank, an S holds a space for a positive or negative sign, an A represents a space where a letter can be inserted, and so on (see Figure 5.7 below).

By putting these symbols in the proper order, you can exercise control over how your data will be read and printed. For example

```
   10  MEAN_TEST IS FIELD (PIC 'ZZZ9V.9999');
```

The 9's here stand for decimal digits. The Z's also stand for digits, but indicate that leading Zero's should be printed as blanks. The period shows where the decimal point is positioned, and the V shows that it should be printed as part of the number. In total, ZZZ9V.9999 means that there are four possible digits to the left and to the right of the decimal point, and leading Zero's are not to be printed.

A Picture repetition factor specifies the number of repetitions of the picture symbol which immediately follows. A Picture repetition factor must be an integer enclosed in parentheses. The following Picture Fields would result in the same description.

```
     BOB is Field (Pic '999V.99');
     BOB is Field (Pic '(3)9V.2(9)');
```

When a decimal point follows the 'V' (as in above example), it indicates, if the field is for a source file, the decimal point is present in the stored data. If for a target file, the decimal point is to be printed.

The symbols used for picture in the author's implementation of MODEL are explained in Figure 5.7. Your system may allow different symbols. Therefore, you may want to check the User's Guide for the version of PL/I implemented on your system.

X stands for any character. A picture variable of all X's is just like a character string variable.

A stands for any alphabetic character or a blank character.

9 stands for a decimal digit in a given position.

Z also stands for a decimal digit, except that zeros on the left will not be printed. For example, 0064 would be printed as 64 if the field were declared as ZZZZ.

* also stands for a decimal digit, but the leading zero is replaced with an asterisk instead of being omitted. Hence **64.

Y stands for a decimal digit, except that a spac will be printed for any zero in any position.

(n) can be used to indicate a number of repetitions of the following character. For example, (3)9 indicates 3 decimal digits. In some versions of PL/I it may cause problems if you try to show that a character is repeating 10 or more times.

T stands for a digit or a plus sign or minus sign, if there is one.

I is the same, except that only a plus sign will be printed. Negative numbers will be printed without a minus sign.

R is the same, except that it prints a minus when the number is negative, but no plus when it is positive or zero.

. indicates the position in which a variable's decimal point is expected to appear.

V indicates the position where you would like a decimal point to be placed when a string of numbers is read, without your having to actually type the decimal point in. If no V character is used, this tells the computer that the decimal point is on the right; that is, the picture variable is seen as an integer. If you use V followed by a period, ".", this will cause a decimal point to be inserted, and later printed out as part of the variable.

, is the position for a comma to be inserted.

$ is the position for the dollar sign. When leading zeros are omitted, the dollar sign will be moved up next to the leftmost printed digit. This symbol cannot occur in the middle of a field.

+ is the position for a plus sign. This is like I except that the sign is placed next to the leftmost digit. This symbol cannot be used in the middle of a field.

- is the same, for a minus sign, in case the number is negative. This symbol cannot occur in the middle of a field.

S will print either sign, like T, but next to the leftmost digit. This symbol cannot occur in the middle of a field.

E indicates the position of the exponent in a floating point number. For example, 4E3 is 4 time ten to the third, or 4,000. Likewise, 5E-4 is 5 time 10 to the minus fourth, or .0005.

Figure 5.7: Symbols Used in the Picture Data Type

5.6.6. ON CONVERSION ERROR

Errors in data type of incoming data are frequently discovered on the input of the data and its conversion to the specified data type. This occurs frequently especially in dynamic testing of programs. The detection of bad data defines automatically a qualified variable MALDATA.R, where R is the name of the SOURCE RECORD which includes the bad data (see section 8.5). The user can declare, with a source FIELD, the disposition of a data type error. There are three options: a) the default is to define the MALDATA.R variable as true and continue with the program b) declare that the program should stop or c) declare to substitute for the FIELD The value of a constant and continue the program. The syntax is shown in Figure 5.6. An example is as follows:

```
        A IS FIELD (PIC '999') ON_CNVERR: 0;
```

If the field A is detected on input to have other than a three digit integer than it will be defined as having the value of zero, and the program will continue.

5.7. SHORTCUTS IN DECLARATION OF DATA STRUCTURES

If you want to include the same data structure in two or more separate FILES, then MODEL allows you to declare the redundant data structure only once. For example, you can declare two or more FILES in one statement, as in:

```
1 (MICKEY, PLUTO) ARE FILES etc.
```

When two or more FILES are declared together by enclosing the names of the FILES in parentheses, separated by commas, The GROUPS and other child variables below them need to be declared only once and will be made part of both FILES, with the same names, repetition counts, and other arguments.

The advantage of using this shortcut is that you only have to describe a particular complex data FILE structure once. Two FILES having the same data structure do not necessarily contain the same data.) Note that in doing this, you create several variables with the same name, which causes ambiguity. Variables with the same names must be distinguished in equation statements by giving them a prefix of the name of their parent file, (see section 4.6)

You can also create sibling FIELD variables of the same data type and length in the same declaration, as in:

```
   15   (CASPER, WENDY) ARE FIELDS (PIC 'ZZ9V.99');
```

Their names will be sufficient to distinguish them from each other, within a single FILE. However, you can't declare multiple RECORDS or GROUPS at one declaration, because the names in the data structures under these RECORDS or GROUPS will be made the same. You would end up with variables with the same name in the same FILE, and you wouldn't be able to use the name of the parent FILE as a prefix to separate them.

5.8. DECLARING REPETITIONS AND OPTIONAL DATA STRUCTURES

In MODEL data declaration there are three options for expressing the repetition counts of GROUPS, RECORDS, and FIELDS.

These options are as follows:

1) If you know how many repetitions a particular variable will have, then you can enter that number as the repetition count. So, to indicate that there are 12 STUDENT Fields for each TEST_SCORE Record we write

```
   3  STUDENT(12) IS FIELD (PIC '99');
```

2) You can enter a minimum-maximum range on the number of expected repetitions of a variable. This method may be your best compromise in saving space. An example of using a repetition range, would be

```
2 PRODUCT (1:8) IS RECORD
```

3) If you do not know the maximum and minimum repetitions of a variable, then you can use an asterisk in place of a number for the repetition count, as in

```
1 DEPT IS FILE,
   2 PRODUCT (*) IS RECORD
```

Whenever you use an asterisk in place of a repetition count, then the MODEL system will try to optimize the use of memory space. If optimization is not possible and all the repetitions of a variable have to be in memory, then space is directly available for up to 9999 repetitions. You will get a warning message in your report. Giving more specific information, if you know it, will save space, and allow higher repetition counts.

If you decide to use either an asterisk or a minimum-maximum range when declaring the repetition count of a variable, you must be sure that your specification includes information for specifying that variable's range. Such information might be ENDFILE markers, range propagation, or control variables with a SIZE or END prefix. See Sections 7.3, 8.2 and 8.3 for detailed discussion of these options.

When you use a repetition range whose minimum is 0, you are defining an optional data structure. This means that the relevant variable and all its child variables (if it is not a Field) may or may not exist at all.

5.9. DECLARING INTERIM DATA STRUCTURES

Interim data structures are associated with interim fields. The values they contain are only present during the execution of the program. Afterwards, they are not available, as target file fields, for further processing.

Interim structures may be handled in three ways:

```
1-  The entire structure may be coded, with Groups with repetition
    counts and Fields and their attributes.

2-  The Fields and their attributes may be coded without the structure,
    i.e. without the Groups with repetition counts.

3-  The Fields may be omitted entirely.
```

In the case of options 2 and 3, the MODEL system will be making assumptions to fill in missing information. It will use equations defining the variables to try to determine the structure and will choose a field attribute if the Field was omitted entirely.

5.10. VARIABLE RANGE DEFINITION

The range of a rightmost dimension of a structure corresponds to the number of repetitions of the structure. It may be a constant or it may vary. If it is constant, it can be specified in the declaration of the strcuture.

For example:

```
A (10) IS GROUP
```

If the range is a variable then this is expressed in the data declaration in either of two ways. Either by specifying the repetitions by (*) or by giving the minimum and maximum, i.e. (minimum: maximum). For example:

```
A (*) IS GROUP or A(1:1000) IS GROUP.
```

The MODEL compiler will attempt to assign as little memory for the respective dimension as possible. However, if the maximum is known, it is a good practice to provide it, otherwise if the MODEL compiler determines that it cannot allocate less memory it will use by default a maximum repetitions of 9999.

If the range is a variable then it must be defined elsewhere. Following are the ways to define the range.

1. Use of Control Variables (see chapter 8).

a. By defining the range variable as a qualified variable prefixed by the keyword SIZE and the variable whose range of the rightmost dimension is to be defined. For example:

```
SIZE.A = IF B THEN 10 ELSE 20;
```

The SIZE prefixed qualified variable must have a lesser dimensionality than the suffix variable, always omitting at least the rightmost dimension of the suffix variable.

b. By defining the condition of the last element in a dimension to be represented by a qualified variable, using the prefix END and as suffix the name of the variable whose range of the rightmost dimension is defined.

For example:

```
        END.A(I)  = A(I)= 'LAST'.
```

This variable has the same shape as the variable in the suffix.

2. By default

If the last element of the dimension could be deduced from an end-of-file marker or from the end-of-record information, then there is no need to specify the range.

3. By propagation

The MODEL compiler attempts to deduce the range from anyone of the other variables whose range is given. This is based on the position of a variable in a tree, on referencing the variable using global subscripts or on the variables participating in an equation. Thus, generally, if there are a number of variables with a dimension of the same range, it should be sufficient to define the range for only one of the variables. If this is not adequate the MODEL compiler will solicit additional range definitions.

6. EQUATIONS

6.1. OVERVIEW

Equations define all the FIELD variables and control variables making up your interim and TARGET data structures. Each equation statement defines one dependent variable placed on the left-hand-side of an equals sign. This variable is defined as equal to an expression on the right-hand-side composed of constants, operators, variables, and functions.

As stated previously, in MODEL each variable can have only one value. Therefore each FIELD to be defined with a different value has to have a unique name or distinguishing subscript. Since MODEL is non-procedural, you don't have to worry about the order in which your variables are defined. You just have to make sure to write an equation to define every field in your TARGET and interim data structures.

In some cases you can optionally omit subscripts when writing equations. (The conditions for subscript omission, as well as other aspects of using subscripts, will be explained in the next chapter.) This is meant to be a shortcut in writing rather than in thinking. Keep in mind that subscripts are implied, when you write equations using variables with omitted subscripts.

The objective of this Chapter is to present the syntax of equations statements along with examples. Equation statements in MODEL may be either simple or conditional, as shown by the following syntax diagram:

```
<equation statement> ::= <simple equation statement> | <conditional
                             equation statement>
```

A simple equation defines the value of the dependent variable on the left side of the equation in terms of a single expression on the right side. An example is

```
        A = B + 3
```

Conditional equation defines a dependent variable as equal to one of several expressions, depending on the value of a condition. An example of a conditional equation is

```
    A = IF B > 2 THEN 7 ELSE 12;
```

In the previous Chapter we explained how you have the option of leaving certain attributes of your SOURCE data, such as variable ranges, unspecified in data declaration. You must then define these attributes in your equation statements using control variables (introduced in Section 3.6). For example, suppose you declared a FIELD variable ALBATROSS to have an uncertain number of repetitions, as follows

```
ALBATROSS(*), IS FIELD ...
```

You could then define the range of ALBATROSS in an equation, such as

```
          SIZE.ALBATROSS = 2;
```

Control variables, such as SIZE.ALBATROSS, may then be used as independent variables in defining other variables in other equations. The uses of the various types of control variables will be explained in the Chapter 8.

6.2. SIMPLE EQUATIONS

As stated previously, equations in MODEL are of two kinds, simple and conditional. Simple equations contain only one expression to define the dependent variable. This expression may be a logical, arithmetic, Boolean, string, or comparison expression, as described in Chapter 3. The syntax diagram for simple equations is shown in Figure 6.1.

```
1 <simple equation statement> ::= <subscripted variable> = <any expression>;
   2 <subscripted variable> ::=   <name> [(<subscript expression>
                                  [, <subscript expression>]*)] |
                                  (<name> [(<subscript expression>
                                  [, <subscript expression>]*)]
                                  [,<name> [(<subscript expression>
                                  [, <subscript expression>]*)]])
      3 <subscript expression> ::= <arithmetic expression>
   2 <any expression> ::= <logical expression> | <arithmetic expression> |
                          <string expression> | <Boolean expression> |
                          <comparison expression>
```

Figure 6.1

Syntax of Simple Equation Statements

The dependent variable in a simple equation statement may take subscripts, and may be a qualified name variable. (The use of subcript expressions is described in detail in the next chapter.) Two or more dependent variables can also be defined in one equation statement, as long as each receives the same definition. In that case, the list of variable names must be enclosed in parentheses, with individual names separated by commas. You can define the dependent variable in terms of an expression containing qualified name variables, constants, non-qualified variables (with or without subscripts), and functions.

Some examples of legal simple assertion statements are as follows:

```
A = B + 5;
SIZE.JLA = 12;
X(I,J) = 4 * I**J;
(JUNG,FREUD) = ADLER;
```

6.3. CONDITIONAL EQUATIONS

6.3.1. OVERVIEW

Conditional equations are more complicated than simple equations, because the choice of defining expression depends on the value of a condition. The dependent variable in a conditional equation is defined in terms of one expression if the condition is true, and in terms of another if the condition is false. This condition is a Boolean expression. A Boolean expression, as defined in Section 3.9, is an expression which has a binary truth value of true or false. For example, the comparison (Boolean) expression "A > 7" must be either true or false, when A is an arithmetic variable.

The process works as follows. In the conditional equation

```
MONTANA = IF A > 7 THEN 5 ELSE 12;
```

the dependent variable MONTANA will be defined as equalling 5 if the condition A > 7 is true, or as 12 if the condition is false. The part of the above statement containing the keyword IF, followed by the conditional Boolean expression, is called the IF-clause. The first defining expression for the dependent variable in a conditional equation, 5 in the above case, is preceded by the keyword THEN. The second defining expression, 12 above, is preceded by the keyword ELSE. A more complex conditional equation statement may contain more than one IF-clause and more than two defining expressions (see below).

The syntax diagram for conditional equations is shown in Figure 6.2.

```
1  <conditional equation statement> ::= <subscripted variable> =
                                          <conditional expression>;
   2  <conditional expression> ::= <IF-clause> THEN <conditional>
                                   [ ELSE <conditional> ]
      3  <IF-clause> ::= IF <Boolean expression>
      3  <conditional> ::= <conditional expression> | <any expression>
```

Figure 6.2

Syntax of Conditional Equation

6.3.2. NESTED CONDITIONAL EQUATIONS

As stated previously, conditional equations allow you to write equations containing several conditions and defining expressions. Suppose, for example, we wanted to write a specification to keep track of how much we should pay various employees in our company. We can use the variable PAY(I) to keep track of paychecks, with the subscript I referring to each individual (e.g. by their number on the payroll). Amount of pay to each person depends on two factors, job-type and number of hours worked, which we can call JOB(I) and HOURS(I), respectively. We will also assume that there are three possible types of jobs in our company: executive vice-president in charge of advertising, programmer-analyst, and janitor. We can write an equation to define a pay amount for each individual (in terms of what they're worth) as follows:

```
PAY(I) = IF JOB(I) = 'PROGRAMMER-ANALYST'
         THEN 50.00*HOURS(I)
         ELSE IF JOB(I) = 'JANITOR'
              THEN 12.50*HOURS(I)
              ELSE 2.17*HOURS(I);
```

That the executive vice-president in charge of advertising should be the one making $2.17/hour is understood. (It's the only job-type left.)

The above example, shows the nesting of an additional condition and two additional defining expressions within the ELSE part of a conditional equation. The example is relatively simple, because the choice of the expression used to define a value for PAY(I) always depends on a single condition. A more complex situation occurs when the choice of defining expression depends on the truth value of two or more conditions. Extending the above example, the amount of pay within each job type could also vary depending on how long the person was with the company. To keep track of this we need a new variable called YEARS(I). When the number of years someone is with our company reaches 10, that person will get a raise.

The expanded equation to calculate pay could be as follows:

```
PAY(I) = IF JOB(I) = 'PROGRAMMER-ANALYST'
         THEN IF YEARS(I) < 10
              THEN 50.00*HOURS(I)
              ELSE 100.00*HOURS(I)
         ELSE IF JOB(I) = 'JANITOR'
              THEN IF YEARS(I) < 10
```

```
                    THEN 12.50*HOURS(I)
                    ELSE 15.00*HOURS(I)
               ELSE IF YEARS(I) < 10
                    THEN 2.17*HOURS(I)
                    ELSE 2.18*HOURS(I);
```

This new example, shows the nesting of an additional condition and two defining expressions within each defining expression of the first example, which nested an additional condition and two defining expressions within the first ELSE expression. The choice of which expression is used to define the dependent variable depends on two conditions. (When writing your own nested conditional expressions, you'll find it helpful, as shown above, to indent, so that conditions and defining expressions at the same level of depth are vertically aligned.)

6.3.3. SIMULTANEOUS EQUATIONS

A MODEL specification may include equations that form a set of simultaneous equations that define variables of any shape and use any operations or functions. These equations may be linear or non-linear. The MODEL compiler identifies these equations and implements their solution by incorporating in the produced program a Gauss_Seidel iterative solution method. By arranging the equations in a block, the user can optionally provide guidance to the compiler in how to make the application of the iterative solution method more efficient.

The syntax is shown in Figure 6.3.

```
<simultaneous equations block>::=

     BLOCK <name>,
     [[SOLUTION] METHOD <IS> GAUSS_SEIDEL,]
     [[MAX[IMUM] ITER[ATIONS]<IS> <number>,]
     [[RELATIVE] ERR[OR] <IS> <fraction>];

     [<equations>]*
     [<simultaneous equation block>]*
     [<equations>]*

     END <name>;
```

Figure 6.3: Syntax of a Simultaneous Equation Block

```
     <initial value equation> ::=
             INITIAL.<variable name>=<expression>
```

Figure 6.4: Syntax of initial value specification

The user provides first a BLOCK declaration giving a name to the block of simultaneous equations. This may be followed optionally by declaring the choice of the solution method to be employed by the compiler. The maximum number of iterations in the iterative method- after which the solution will stop - can also be provided optionally. The default value is 100. The iterations stop also if convergence is achieved. A convergence condition can be defined optionally by giving a fraction by which new solution values would differ in ratio from the previous iteration. The default value of the fraction is 0.001 - namely of a final solution differs by less than 1/1000th of its value from the preceeding iteration value. This then is followed by the simultaneous equations. The substitution of values in the iterative solution method will then proceed in the order of the equation. This order may be important in achieving the convergence faster. Simultaneous equation solutions may be nested one within the other to accelerate and make more efficient the solution process. This is shown by the nesting of simultaneous equation blocks. Finally the block ends with an END statement.

Figure 6.4 shows how the iterative solution method can optionally start with specified initial values of the variables. The initial value is represented by a qualified name variable, with the prefix INITIAL and the variable name as a suffix. The INITIAL. <variable name> defining equations can be placed anywhere in a specification, not necessarily together with the simultaneous equation block. The default is having the value 1 as initial value.

Consider the following example of a nested simultaneous equation block.

```
.
.
.

BLOCK SIMEQ1,
        SOLUTION GAUSS_SEIDEL,
        ITERATIONS 50,
        ERROR 0.005;

A = C1 * B + C2* C + C3 * D;
        BLOCK SIMEQ2
                ERROR 0.01;

        B = C4 * A + C5 * C;
        C = C6 * D + C7 * B;

        END SIMEQ2,

D = C8 * A+C9 *C;

END SIMEQ1;

INITIAL.A = 10;
```

There are four variables A,B,C,D and respective defining equations. The C1 to C9 coefficients are defined elsewhere. The nested block with equations for B and C is to be solved iteratively within the nesting (outside) block. Only A is given an initial value. By default the initial values of the other variables would be 1. If the BLOCK, END and INITIAL.A statements are omitted then the compiler would generate a procedure to solve iteratively the five equations, using 1 as initial value for all five variables. The convergence error would be less than .001, and the iterations would stop after 100 iterations if convergence has not been attained previously.

7. USING SUBSCRIPTS IN EQUATIONS

7.1. OVERVIEW

Whenever we mentioned subscripts earlier in this text, we were really talking about subscript expressions. Subscript expressions specify which elements of a subscripted variable are to be used in an equation as independent or dependent variables. We express a subscripted variable by following the name of a repeating data structure (usually a FIELD) with one or more subscript expressions, separated by commas. The following is a sample of the variety of legal subscript expressions used in subscripted variables:

```
HAWKEYE(I,J)
BJ(I,J+3)
RADAR(KLINGER(K))
HOT__LIPS(FOR__EACH.BURNS)
```

Subscript expressions are arithmetic expressions, as defined in Section 3.9.3. These expressions give integer values corresponding to the postions of elements in a data array or tree, with one subscript expression for each dimension or subscript (see Section 2.2). (The word index is also used to stand for the value of a subscript expression.) If the values of the subscript expressions of a subscripted dependent variable are constants, then the definition will apply only to the element with those indices. For example, the statement

```
ALBERT(2) = 3;
```

defines the second element of the subscripted variable ALBERT as equal to 3, but does not affect the value of any of the other elements.

An equation like the one above, which defines only one element of an array is very inefficient. When a subscripted variable contains a subscript expression that can take on a range of values, an equation can simultaneously define values for all the elements of that array variable whose index values are in that range. In an equation such as

```
BEAN(I) = 3;
```

all elements of the dependent variable BEAN(I) are given the same value. Alternatively, if both independent and dependent variables are subscripted, then each element of the dependent variable can be defined differently, depending on the value of the corresponding element of the independent variable. For example, in

```
WILMA(I) = FRED(I);
```

each element of WILMA(I) is defined as equal to the element of FRED(I) with the same index value of I.

By including conditions, you can limit which elements of subscripted independent variables will be used to define the dependent variable. For example, in

```
BETTY(I) = IF I < 4 THEN 0 ELSE BARNEY(I)*17;
```

only elements of BARNEY(I) with subscripts of 4 or greater will be used in defining values of BETTY(I). If a variable in the defining expression is subscripted, ;but the dependent variable is not, then you must include conditions to define the dependent variable in terms of only one of the elements of the independent variable. (Otherwise, you violate MODEL assumptions.) An example would be,

```
ROCKY = IF I = 4 THEN BULLWINKLE(I);
```

Although BULLWINKLE(I) can contain many elements, only the fourth one will be used to define the scalar ROCKY. (Alternatively, you could have written

```
        ROCKY =   BULLWINKLE(4);)
```

The definition of all the elements of an array in a single equation, is done through the use of subscript variables. A subscript variable is an arithmetic variable which can take on any integer value from one up to the total number of elements of a subscripted variable (along a particular dimension). MODEL allows the use of global and local subscript variables. A global subscript variable takes the same range in all the assertions in which it is used throughout the specification, while a local subscript variable may take on a different range in each assertion. Global and local subscript variables are discussed in more detail in Section 7.3. A subscript expression may contain a subscript variable by itself, or as part of a complex arithmetic expression containing constants, functions, and variables, which may themselves be subscripted. The next section will examine how the different types of subscript expressions are classified.

7.2. TYPES OF SUBSCRIPT EXPRESSIONS AND THEIR USES

Subscript expressions in MODEL can be categorized according to their form. The MODEL processor compiles some forms more efficiently than others, so that these are preferred. The types of subscript expressions are as follows:

```
1)  I,
2)  I-K, where K > 0,
3)  none of the others (e.g., constant
    variables or other expressions),
4)  X(I), where X(I) is sublinear
    indirect indexing vector,
5)  X(I-C)-K, where X(I) is a sublinear index
    and C + K =>1,
```

The preceding description of types of subscripts uses the following nomenclature:

I is a subscript variable which can take on any integer value in the range of the variable for which it is an index.

C and K are integer constants.

X is a sublinear indirect indexing vector (see below).

The MODEL compiler generates a more efficient program when you use Type 1 or 2 subscript expressions, then when you use Type 3. An indirect indexing vector is a subscripted variable which is used in the subscript expression of another subscripted variable. The use of indirect indexing vectors as subscript expressions is optimized when those indirect indexing vectors are sublinear, as illustrated by Type 4 and 5 subscript expressions above.

Sublinear indirect indexing variables are used when we want to define an array variable as consisting of selected elements of another array variable, where the selected elements in these arrays are in the same order. In the case where both arrays are one dimensional, we use only the sublinear indirect indexing variable. The MODEL system implements a much more efficient computation using these two types (4 and 5 above) of indirect indexing variable than those with subscripts of type 3 for such transformations.

7.2.1. SUBLINEAR INDIRECT INDEXING

X(I) is a sublinear indirect indexing vector if it is defined by an assertion of the form:

```
                X(I) = IF I = 1
                       THEN IF<Boolean Expression 1> THEN 1 ELSE 0
```

```
                    ELSE IF <Boolean Expression 2>
                         THEN X(I-1)
                         ELSE X(I-1) + 1;
Alternately X(1) may be defined by the SUBLINEAR function:
              X(I)= SUBLINEAR (<Boolean Expression 1>,
                               <Boolean Expression 2>;
```

In words, this says that X(I) is equal to either 1 or 0, when I is equal to 1. When I is greater than 1, the value of X(I) for each I, is defined as equal to either the value of X(I-1) or X(I-1) plus 1, depending on a condition of whether the respective element is or is not selected. The effect is that the sublinear indirect indexing vector X(I) always takes integer values and is monotonically increasing with I. It is also less than or equal to I, for any I, because I always increases by 1 (linearly), while X(I) increases by 0 or 1 (less than linearly or sublinearly). If X(I) is used in the subscript position for a dimension of any other variable then the range of the dimension must not be specified, as it is dependant on the range of I.

The usefulness of sublinear indirect indexing vectors in business applications is illustrated by the following specification, displayed in Figure 7.1. We start with a FILE of life insurance information. The FILE is called DATA. Each RECORD in the FILE contains three FIELDS: the first to give the year of birth of the policy-holder, the second to give his or her name, and the third to give the amount he or she pays each year. The three FIELDS are called BIRTH, NAME, and PREMIUM, respectively. We decide to separate the RECORDS into three distinct TARGET FILES based on year of birth, so that we can be sure everyone is paying the appropriate premiums. (These TARGET FILES will have the same data structure, except for number of RECORDS, as the original SOURCE FILE.) We want the RECORDS of everyone born before 1920 to go into the first TARGET FILE, OLDER, the RECORDS of people born from 1920 to 1950 to go into the second FILE, MIDDLE, and the RECORDS of people born after 1950 to go into the third, YOUNGER.

```
MODULE: INSURANCE;
SOURCE: DATA;
TARGET: OLDER,MIDDLE,YOUNGER;

1 (DATA, OLDER, MIDDLE, YOUNGER) ARE FILES (R(*));
  2 (*) R IS RECORD;
     3 BIRTH IS FIELD (PIC '9999');
    3 NAME IS FIELD (CHAR (20));
    3 PREMIUM IS 20 (PIC 'ZZ99');

1 INT IS FILE
  2 DATE1 (*) IS FIELD (NUM (5));
  2 DATE2 (*) IS FIELD (NUM (5));
  2 DATE3 (*) IS FIELD (NUM (5));

 I IS SUBSCRIPT;

DATE1(I) = IF DATA.BIRTH(I) < 1920
           THEN IF I = 1
                THEN 1
                ELSE DATE1(I-1) + 1
           ELSE IF I = 1
                THEN 0
                ELSE DATE1(I-1);

DATE2(I) = IF DATA.BIRTH(I) >= 1920 & DATA.BIRTH(I) <= 1950
           THEN IF I = 1
                THEN 1
                ELSE DATE2(I-1) + 1
           ELSE IF I = 1
                THEN 0
                ELSE DATE2(I-1);

DATE3(I) = IF DATA.BIRTH(I) > 1950
           THEN IF I = 1
                THEN 1
                ELSE DATE1(I-1) + 1
           ELSE IF I = 1
                THEN 0
                ELSE DATE3(I-1);
```

FIGURE 7.1 CONTINUED NEXT PAGE

```
OLDER.BIRTH(DATE1(I)) = IF I = 1 & DATE1(I) = 1
                        THEN DATA.BIRTH(I)
                        ELSE IF DATE1(I) > DATE1(I-1)
                             THEN DATA.BIRTH(I);

OLDER.NAME(DATE1(I)) = IF I = 1 & DATE1(I) = 1 THEN DATA.NAME(I)
                       ELSE IF DATE1(I) > DATE1(I-1) THEN DATA.NAME(I)

OLDER.PREMIUM(DATE1(I)) = IF I = 1 & DATE1(I) = 1 THEN DATA.PREMIUM(I)
                        ELSE IF DATE1(I) > DATE1(I-1) THEN DATA.PREMIUM(I);

MIDDLE.BIRTH(DATE2(I)) = IF I = 1 & DATE2(I) = 1 THEN DATA.BIRTH(I)
                         ELSE IF DATE2(I) > DATE2(I-1) THEN DATA.BIRTH(I);

MIDDLE.NAME(DATE2(I)) = IF I = 1 & DATE2(I) = 1 THEN DATA.NAME(I)
                        ELSE IF DATE2(I) > DATE2(I-1) THEN DATA.NAME(I);

MIDDLE.PREMIUM(DATE2(I)) = IF I = 1 & DATE2(I) = 1 THEN DATA.PREMIUM(I)
                         ELSE IF DATE2(I) > DATE2(I-1) THEN DATA.PREMIUM(I);

YOUNGER.BIRTH(DATE3(I)) = IF I = 1 & DATE3(I) = 1 THEN DATA.BIRTH(I)
                          ELSE IF DATE3(I) > DATE3(I-1) THEN DATA.BIRTH(I);

YOUNGER.NAME(DATE3(I)) = IF I = 1 & DATE3(I) = 1 THEN DATA.NAME(I)
                         ELSE IF DATE3(I) > DATE3(I-1) THEN DATA.NAME(I);

YOUNGER.PREMIUM(DATE3(I)) = IF I = 1 & DATE3(I) = 1 THEN DATA.PREMIUM(I)
                          ELSE IF DATE3(I) > DATE3(I-1) THEN DATA.PREMIUM(I);
```

Figure 7.1

Sample Specification Using Sublinear Indirect Indexing Vectors

We can accomplish this task using three sublinear indirect indexing vectors: DATE1(I), DATE2(I), and DATE3(I), to keep track of the FIELDS to be placed into the three TARGET FILES. (Figure 7.2 shows sample values of the three sublinear indexing vectors for different values of I and DATA.BIRTH(I).) We defined DATE1(I) as follows:

```
DATE1(I) = IF DATA.BIRTH(I) < 1920
     THEN IF I = 1
          THEN 1
          ELSE DATE1(I-1) + 1
     ELSE IF I = 1
          THEN 0
          ELSE DATE1(I-1);
```

so that it only increases when I corresponds to a RECORD containing a BIRTH FIELD with a value less than 1920. We then wrote the equation

```
OLDER.BIRTH(DATE1(I)) = IF I = 1 & DATE1(I) = 1
                        THEN DATA.BIRTH(I)
                        ELSE IF DATE1(I) > DATE1(I-1)
                             THEN DATA.BIRTH(I);
```

using DATE1(I) as a subscript expression, to keep track of the BIRTH FIELDS entered into the TARGET FILE OLDER. When DATA.BIRTH(I) has a value greater than 1920, and DATE1(I) doesn't change, then nothing is added to the new FILE. However, every time DATE1(I) is increased by 1, the FIELD BIRTH (which has the index I in the FILE DATA and the index DATA1(I) in FILE OLDER) is added to TARGET FILE OLDER.

I	DATA.BIRTH(I)	DATE1(I)	DATE2(I)	DATE3(I)
1	1915	1	0	0
2	1930	1	1	0
3	1947	1	2	0
4	1953	1	2	1
5	1918	2	2	1
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

Figure 7.2
Sample Values for Several Sublinear Indirect Indexing Vectors

DATE1(I) is used in the same way to keep track of the of the FIELDS DATA.NAME(I) and DATA.PREMIUM(I). We write equations to add them to the FILE OLDER when the value of DATE1(I) increases. Similarly, we can define the sublinear indirect indexing vectors DATE2(I) and DATE3(I) as increasing for values of DATA.BIRTH(I) in the ranges 1920 to 1950 and greater than 1950, respectively. This allows us to write equations defining the values of MIDDLE.BIRTH(DATE2(I)), MIDDLE.NAME(DATE2(I)), MIDDLE.PREMIUM(DATE2(I)), YOUNGER.BIRTH(DATE3(I)), YOUNGER.NAME(DATE3(I)), and YOUNGER.PREMIUM(DATE3(I)).

Note that the range of the records of source file DATA.R, is not defined, and by implication it is given by the end-of-file of the DATA file. The ranges of OLDER, MIDDLE and YOUNGER are implied from the equations for DATE1, DATE2, and DATE3 which select the elements of DATA.R. Therefore, these ranges must not be specified. Namely, data indexed by sublinear indirect indices must not have a range specification. The equations with variables on the left hand side which use the sublinear indirect index (e.g. BIRTH, NAME, PREMIUM) can actually be much simpler by omitting the conditions on the right hand side as they are always inserted automatically by the MODEL compiler. Namely these equations may be

```
OLDER.BIRTH(DATE1(I)) = DATA.BIRTH(I);
OLDER.NAME(DATE1)I)) = DATA.NAME(I);
OLDER.PREMIUM(DATE(I)) = DATA.PREMIUM(I); etc.
```

The implications in the above equations is that if there exists an ambiguity, where the left hand variable appears to be defined multiple times, for several values of I, then only the definition with the lowest value of I applies.

The sublinear index may appear on the right hand of equations as well in expressions of the form X(I-C)-K where C+K=>1. (see type 5 above)

The equation defining a sublinear index must be stated in the form

```
X(I) = IF  I = 1 THEN[IF CONDITION 1 THEN 0 ELSE]1
       ELSE IF ANY CONDITION 2 THEN X(I-1)+1
ELSE X(I-1);
```

or alternately by using the SUBLINEAR function, as stated above.

7.3. SUBSCRIPT VARIABLES

Subscript variables in MODEL fall into two classes, global and local. They differ in their scope of applicability throughout the specification. The two types of subscript variables are defined syntactically in Figure 7.3.

```
1  <subscript variable> ::= <global subscript variable> |
                            <local subscript variable>
  2  <global subscript variable> ::= <name> | FOR__EACH.<name>
  2  <local subscript variable> ::= SUB1 | SUB2 | SUB3 | SUB4 | SUB5 |
                                    SUB6 | SUB7 | SUB8 | SUB9 | SUB10
```

Figure 7.3

Types of Subscript Variables

WHY NOT
1) ARE GLOBAL SUB[illegible]
LOCI.LOCJ ARE LOCAL SUBSC[illegible]

A global subscript has the same range in all the assertion statements in which it is used. For example, the global subscript variable I, once it has been declared in a statement like

```
I IS SUBSCRIPT (10);
```

can be used in different assertions as part of the subscript expressions of different variables, and it will always take the same range (see example below).

```
CHESTER(I) = JESSICA(I) - DONAHUE(I);
CORRINE(I) = EUNICE(11-I);
```

There are two types of global subscript variables. The first is declared in a subscript declaration statement. The second is a qualified name variable of the form FOR__EACH.X, where X is the name of a data structure. The syntax for declaring global subscript variables in a subscript declaration statement is shown in Figure 7.4. The following are examples of legal subscript declaration statements:

```
ROBIN IS SUBSCRIPT;
(BATMAN,SUPERMAN) ARE SUBSCRIPTS;
FLASH SUB (15);
```

The examples show that more than one global subscript variable can be declared in the same statement Also, declaring ranges for global subscript variables is optional in certain situations (see below).

```
1  <subscript declaration statement> ::=
   <variable> [<IS>] <SUBSCRIPT> [(<range of subscripts>)];
   2  <SUBSCRIPT> ::= SUBSCRIPT|SUBSCRIPTS|SUB|SUBS
   2  <range of subscripts> ::= <unsigned integer>
```

Figure 7.4

Syntax for Global Subscript Declaration

Global subscript variables do not always need to be declared, and if declared, their ranges do not always need to be specified in subscript declaration. The MODEL compiler can sometimes calculate a range for a global or local subscript variable which was not declared, as long as the variable is used as part of a subscript expression in an equation. Range propagation is the process of automatically assigning ranges (or repetition counts) to global and local subscript variables. This can be done because each specification contains several sources of information about ranges. For example, if the range of a subscript variable is not declared, but the subscript variable is used with a SOURCE FILE or TARGET FILE variable with a declared range, then the subscript variable will automatically take this range in all other equations in which it appears. Once the range of a subscript variable is set, then the MODEL compiler will know this to be the range of every other variable using it as a subscript, even ones whose ranges weren't specified. Assertions that define control variables, prefixed with SIZE or END, and the positions of ENDFILE markers are additional sources of information used in range propagation as well (see Sections 8.2, 8.3 and 8.10).

Another way to create a global subscript variable is to add the prefix FOR_EACH to the name of a data structure. A qualified name global subscript variable, such as FOR_EACH.KIRK, will have the same range as the rightmost subscript dimension of the variable, KIRK, whose name it incorporates. Once a qualified subscript is defined, it can be used as a subscript for other variables with the same range. If SPOCK had the same range as KIRK in its rightmost subscript dimension, then SPOCK(FOR_EACH.SPOCK) and SPOCK(FOR_EACH.KIRK) would be equivalent.

Predefined local susbscript variables named SUB1, SUB2, ..., SUB10 are available in the MODEL system. Each of these local subscript variables may have a different range in each equation in which it is used. Note that if the same local subscript variable is used in the subscript expressions of several variables in the same equation, then it will have the same range in all of those variables. The ranges of local subscript variables are not declared, but instead are derived by the MODEL compiler through range propagation.

7.4. CONVENTIONS FOR SUBSCRIPT OMISSION

As explained previously, variables in MODEL equations are always FIELD variables. For a FIELD variable of n dimensions, it takes n subscript expressions to distinguish each individual element. However in an equation containing several subscripted variables, copying long lists of subscripts after each variable can get quite tedious. Therefore MODEL has a convention for omitting subscripts. Subscript expressions to be omitted must have the following characteristics:

1) **They are common to all variables in an equation.**
2) **They are Type 1 subscript expressions (as defined in Section 7.2).**
3) **They are not used as independent variables in the equation, as in:**

```
SHEEBA(I,J) = 4 * I + J;
```

4) **They are in the same order in each variable from which they are removed.**
5) **They are on the left of other subscript expressions (or have no subscript expressions to the right).**

If all the subscript expressions in an assertion have the above characteristics, then they can be omitted, without changing the meaning of the equation, as in,

```
A(I,J,K) = 2 * B(I,J,K) + C(I,J);   subscripts I,J can be omitted
A(J,K) = 2 * B(J,K) + C(J);         subscript J can be omitted
A(K) = 2 * B(K) + C;                subscript K can be omitted
```

8. CONTROL VARIABLES

8.1. OVERVIEW

This Chapter will explain how to use control variables. Control variables allow you to use equations to define data attributes that you chose not to specify in data declarations, such as the range of a subscripted variable, the length of a piece of data, or the POINTERS for the RECORDS of an ISAM FILE. When you define a control variable, you can use it as an independent variable in defining expressions or conditions. In these ways, control variables increase your flexibility in setting up your specifications.

Control variables are qualified name variables, which means they are constructed by attaching MODEL keyword prefixes to variable names. (The prefix is attached to the variable name by using a period, as in SIZE.CHARLES; see Section 3.6 for more information.) Although the keyword prefixes of control variables may be attached to the names of GROUPS or RECORDS (as in POINTER.X), the control variables themselves act as FIELDS, each holding an individual piece of data. The MODEL control variable keyword prefixes to be discussed in this chapter are SIZE, END, LENGTH, NEXT, SUBSET, POINTER, FOUND, ENDFILE and EMPTY.

8.2. SIZE.X

SIZE is one of two keyword prefixes in the MODEL language used to define the range of a dimension of an array variable, if that range was not specified in data declaration. The other prefix used for this purpose is END, discussed in the next section. If X is a subscripted FIELD, GROUP, or RECORD variable, then SIZE.X may be used to define and represent the number of elements in the rightmost subscript dimension of X in terms of an arithmetic expression whose value is an integer. SIZE.X may be defined equal to 0; see Section 4.4 on optional data structures. For example, suppose X was declared as follows:

```
        3  X (*) IS FIELD (PIC '99');
```

A equation of the form

```
             SIZE.X = 10;
```

will set the range of X to 10. (If the range of I was declared in a subscript declaration statement, then an equation defining SIZE.X is unnecessary, because X(I) is given the same range as I through range propagation, as explained in Section 7.3.)

If X has more than one subscript, then SIZE.X may have subscripts as well, although the total number will always be at least one less than in X. This is because we can define the number of elements in a one-dimensional vector with a zero-dimensional scalar, as the above example shows. If SIZE.X is subscripted, then each of its elements will give the range of a vector in X (see below).

If variable X has n subscripts, we can define SIZE.X variables having from 0 to n-1 subscripts, as illustrated in the following example. Consider FILE Z, declared in Figure 8.1(a), which contains RECORDS Y and FIELDS X. Y and X repeat an unknown number of times. Subscript I is used to distinguish the elements of Y, and subscript J distinguishes the elements of X. Figures 8.1(b) and 8.1(c) display two possible data arrays which are consistent with this declaration. In the first, each RECORD contains the same number of FIELDS, while in the second, the number of FIELDS vary between RECORDS. Our goal is to write equations to define the ranges of X and Y for each of the two arrays.

```
1  Z IS FILE,
   2  Y(*) IS RECORD,
3  X(*) IS FIELD (PIC '9');
I IS SUBSCRIPT;
                    J IS SUBSCRIPT;
```

(a)   **Declaration of FILE Z**

I	J = 1	J = 2	J = 3
1	4	9	7
2	3	6	1

(b)   **Possible Structure for X(I,J)**

I	J = 1	J = 2	J = 3
1	2	8	3
2	5	7	
3	6	11	9

(c)   **Another Structure for X(I,J)**

Figure 8.1
Data Structures for FILE Z

RECORD variable Y(I) is one-dimensional, so that its range can be defined with a scalar in a simple equation such as:

```
    SIZE.Y = 2;     for the first array (Figure 8.1(b)), and

   SIZE.Y = 3;     for the second (Figure 8.1(c)).
```

FIELD variable X(I,J) is two-dimensional. In the first array, the range of J is the same no matter what the value of I. In this situation, we can write a simple equation on to define one range which holds for the whole FILE Z, as in

```
SIZE.X = 3;
```

In the second array, the range of J changes for each RECORD, as the value of subscript I changes. The range can no longer be defined in a simple equation. Instead a conditional equation is required, for example

```
        SIZE.X(I) = IF I = 2 THEN 2 ELSE 3;
```

In this case, a vector is needed to define the range of the rightmost dimension of X(I,J), because the range of J depends on the value of I.

We saw in the above example that we could completely define the range of the doubly subscripted X(I,J) with either the singly subscripted SIZE.X(I) or the non-subscripted SIZE.X, depending on the structure of X(I,J). These principles generalize to the writing of equations using SIZE qualified control variables to define the range of data arrays containing many more dimensions and susbscripts. If SIZE.X has m subscripts and X has n subscripts, then $0 <= m < n$. Always the nth dimension in X must be omitted in SIZE.X.

8.3. END.X

The MODEL keyword prefix END provides an alternative to SIZE in forming a control variable to define ranges. Unlike SIZE.X, END.X is a Boolean variable (defined in Section 3.9.6), and takes the same number of subscripts as repeating variable X. Each element of END.X consists of a single bit value of true or false. An element of END.X (distinguished by n supscripts) is defined as true when the rightmost subscript of X takes its maximum value. Otherwise, each element of END.X is defined as false.

The array END.X is usually defined in terms of a comparison expression which will be true only when the rightmost subscript takes its maximum range. To define the range of a vector S(I) as equal to some constant K, we could write

```
    END.S(I) = I = K;
```

END.S(I) would be true when I equals K and false otherwise, thereby defining the range of S(I) as K.

IF X contains two or more dimensions, then the array END.X can be used to define the range of the rightmost dimension. For example, to define the range of variable X(I,J) from Figure 8.1(b), we could write the simple equation

```
    END.X(I,J) = J = 3;
```

thereby defining the range of X(I,J) as 3, irrespective of the value of I. The values of the Boolean elements of END.X(I,J) based on the above assertion are shown in Figure 8.2(a).

		J	
I	1	2	3
1	F	F	T
2	F	F	T

(a) **Values of Elements of END.X(I,J) for Figure 8.1(b).**

		J	
I	1	2	3
1	F	F	T
2	F	T	
3	F	F	T

(b) **Values for Elements of END.X(I,J) for Figure 8.1(c)**

Figure 8.2

Values of Elements of END.X(I,J)

However, sometimes the range of a variable in a particular dimension may change depending on the value of a subscript for a higher dimension in the array. This is the case for the range of X(I,J) as illustrated in Figure 8.1(c). To define the range of the three vectors X(I,J) for I equalling 1, 2, and 3 we need to write a conditional equation in which the value of END.X(I,J) depends on both the value of I and the value of J. Such an equation would be

```
END.X(I,J) = IF I = 2 THEN J = 2 ELSE J = 3;
```

The values of the elements of END.X(I,J) defined by this equation are shown in Figure 8.2(b). END.X(I,J) is defined as true when J equals 3 for the first vector, when J equals 2 for the second, and when J equals 3 for the third.

As an alternative, you can make range definition via an END qualified control variable depend on the value of the elements of the original variable, rather than their subscripts. You do this by setting up the data so that the last element of the subscripted variable with an unspecified range is given a unique value, such as 999, making it a termination marker. You can then write an equation like

```
END.Z(I) = Z(I) = 999;
```

This will define END.Z(I) as true when Z(I) equals the termination value of 999.

You can also use an expression containing an END qualified control variable in the condition in a conditional equation. This way you can make the value of a dependent variable depend on whether the END control variable is true. For example, we could write

```
A(I) = IF END.Y(I) THEN 6 ELSE 8;
```

This defines the value of A(I) as 6 when END.Y(I) is true (at the maximum range of Y(I)), and 8 otherwise.

Sometimes, defining unspecified variable ranges through SIZE or END qualified control variables is not necessary, even if they cannot be inferred from subscripts through range propagation. This occurs when variable ranges may be obtained by the MODEL compiler from an ENDFILE marker. Whenever a set of data is read from a data FILE, the last element of that FILE is automatically marked. This last element can be used by the compiler to define the range of a subscripted varible, just as you would define it using an END control variable. This repeating variable can be a RECORD or GROUP. The restrictions are that there can be only one subscripted variable (with uncertain range) in the FILE, and that the last element of this repeating variable has to occur at the end of the FILE. When these restrictions are satisfied, the unspecified range of the subscripted variable will be defined automatically. Otherwise, range definition through SIZE or END control variables are required.

8.4. LEN.X

The LEN.X control variable is used to define the length (number of characters) in FIELD variable X, when this information is left unspecified in data declaration. (As described in Section 4.6.1, the length of a FIELD variable of character data type may be declared as a minimum-maximum range.) If X is subscripted, then LEN.X will have the same number and range of subscripts, for example

```
        LEN.MARIAN = 7;
        LEN.EVE(I) = IF I < 5 THEN 9 ELSE 12;
```

Each element in LEN.X(I) defines the length of the element of X(I) with the corresponding subscript in terms of any arithmetic expression whose value is an integer. The only restriction is that the value of LEN.X cannot depend on the value of any FIELDS physically positioned after FIELD(S) X in the same SOURCE FILE RECORD.

8.5. MALDATA.X

MALDATA.X is a Boolean variable which takes the same number of subscripts as repeating SOURCE RECORD X. If a conversion error occurred when reading in a FIELD of X, then the element of MALDATA.X corresponding to that RECORD is defined as true. Otherwise, each element of MALDATA.X is defined as false.

8.6. NEXT.X

If X is a FIELD in a sourcefile RECORD, then NEXT.X has a value equal to the contents of the corresponding FIELD in the same position in the following RECORD in the FILE. (This will be the RECORD with the next higher RECORD-level subscript, unless it is the last RECORD in a GROUP, in which case the next RECORD will be the first one in the next GROUP.) NEXT.X can be of any data type, depending on the data type of X.

We can use NEXT.X to define the range of a GROUP of RECORDS by making the definition of the END of the GROUP contingent on the Boolean value of an expression comparing X and NEXT.X. For example, Figure 8.3 shows the data declaration of a FILE (named SALES) where RECORDS (named INVOICE) are placed in GROUPS (named PRODUCT), but the number of RECORDS in each GROUP is unspecified. Each RECORD contains an identifying FIELD called PIN, short for product identification number, which has a common value for all the RECORDS in any one GROUP, but is different among GROUPS. By writing an equation like

```
                END.INVOICE(I,J) = PIN(I,J) ^= NEXT.PIN(I,J);
```

we can define the Boolean variable END.INVOICE(I,J) as true, specifying that a particular RECORD is the last one in a PRODUCT GROUP, when the value of PIN is different in the following RECORD. This allows the number of INVOICE RECORDS in each PRODUCT GROUP to be determined.

```
1 SALES IS FILE,
  2 PRODUCT(41) IS GROUP,
    3 INVOICE(*) IS RECORD,
       4 PIN IS FIELD (PIC '999999'),
      4 QUANT IS FIELD (PIC '9999'),
      4 PRICE IS FIELD (PIC '999V.99');
I IS SUBSCRIPT;
J IS SUBSCRIPT;
```

Figure 8.3

Data Declaration of FILE SALES

NEXT.X cannot be used for RECORDS in ISAM FILES. A second restriction is that the number of FIELDS to the left of FIELD X should be fixed in the RECORD. If there are a varying number of FIELDS to the left of FIELD X in each RECORD, then the NEXT.X FIELD will not be located correctly. There is no problem, however, if there is a varying number of FIELDS to the right of X in each RECORD.

8.7. SUBSET.X

SUBSET.X is a Boolean control variable in which each element corresponds to an element of subscripted RECORD variable X. You can define each element of SUBSET.X as true or false, depending on whether you want the RECORD from X with the corresponding subscripts to be included in a TARGET FILE. For example, to omit the second of the three RECORDS in TARGET FILE E, declared below in Figure 8.4, you could write

```
                SUBSET.F(I) = I ^= 2;
```

The above equation defines a value of true for all the elements of SUBSET.F(I), except the second. Those RECORDS of F(I) for which SUBSET.F(I) is true will be included in TARGET FILE E; the second RECORD, for which SUBSET.F(I) is false, won't be.

```
         1 E IS FILE,
                      2 F(3) IS RECORD,
                         3  OUT IS FIELD (PIC '9');
                     I IS SUBSCRIPT;
```

Figure 8.4

TARGET FILE to Demonstrate the Use of SUBSET

E must be a target file and can be either sequential or index sequential organization. The use of SUBSET does not affect computations. End as TARGET data. For example, you could define a FIELD in another TARGET FILE in terms of omitted FIELD OUT(2). Therefore you must be sure to declare the full range of RECORDS for your TARGET FILE, including those to be omitted.

When SUBSET prefixes a record name in a target index sequential (ISAM) file, it denotes whether the respective record is to be retained (or deleted) from the file (see further discussion in Sections 8.8 and 8.9).

8.8. POINTER.X

The next two keywords, POINTER and FOUND, are used with keyed FILES. Section 5.4 describes how each RECORD, X, in a keyed FILE has a FIELD, called a KEY, which contains a unique identifying alphanumeric string. The control variable POINTER.X contains an array of these strings, called POINTERS, as elements. When you define the strings making up POINTER.X, you simultaneously define the desired organization of RECORDS of the keyed FILE. The keyed FILE is rearranged, in terms of number and ranges of dimensions, so that the positions of the RECORDS, as identified by their KEYS, match up with the positions of the corresponding POINTERS in POINTER.X. For example, the assertion statement

```
POINTER.X(I) = Y(I);
```

would shape the RECORDS of the keyed FILE into a one-dimensional vector, while the statement

```
POINTER.X(I,J) = Z(I,J);
```

would reorder them into a two-dimensional matrix. RECORDS from the keyed FILE having KEYS for which there are no corresponding POINTERS will be excluded from the reorganized FILE.

POINTER.X is usually defined in terms of an array of FIELDS taken from a separate reference SOURCE FILE. Once the RECORDS of the keyed FILE are given the structure of this array, you can use subscripts to refer to specific FIELDS from keyed RECORDS as independent or dependent variables in your assertion statements. In this way, you can use a keyed FILE as a SOURCE FILE, a TARGET FILE, or as both a SOURCE and TARGET FILE. In the last case, as explained below, you can use POINTER and FOUND to easily update certain RECORDS in a keyed FILE while leaving others unchanged.

For example, Figure 8.5 illustrates the specification you would write to define the FIELDS of a TARGET FILE in terms of the FIELDS from a keyed SOURCE FILE. SOURCE FILE E is declared as ISAM, with single RECORD. SOURCE FILE, B, contains a two-dimensional FIELD variable, D(I,J), which gives the values of the POINTERS we will use to restructure FILE E. The elements of D(I,J) are shown in Figure 8.6(b).

We then use the statement

```
POINTER.FT(I,J) = D(I,J);
```

to define the shape of keyed RECORDS as the two-dimensional matrix. The FIELD OUT2 of the keyed RECORD variable FT can then be described in terms of this structure, allowing the FIELD variable K(I,J) in TARGET FILE G to be defined. The values of the elements of TARGET FIELD K(I,J), defined from SOURCE FIELD OUT2(I,J), are shown in Figure 8.6(c).

```
MODULE: A;
SOURCE: B,E;
TARGET: G;

1 B IS FILE,
  2 C(*) IS RECORD,
     3 D(*) IS FIELD (PIC '9');

1 E IS FILE KEY IS OUT1 ORG IS ISAM,
  2 FT IS RECORD,
    3 OUT1 IS FIELD (PIC '9'),
    3 OUT2 IS FIELD (PIC 'Z99');


1 G IS FILE,
  2 H(*) IS RECORD,
    3 K(*) IS FIELD (PIC 'Z99');

I IS SUBSCRIPT (2);
J IS SUBSCRIPT (3);

POINTER.FT(I,J) = D(I,J);

K(I,J) = OUT2(I,J);
```

Figure 8.5

Example Using POINTER with Keyed FILE as SOURCE

If the POINTER variable contains an identifying string which is not a KEY for any RECORD in the keyed FILE, then the message

```
RECORD NOT FOUND IN FILE filename WITH KEY key-value
```

is printed. (see discussion of FOUND in next section) Also, if the KEY of a particular RECORD from the keyed FILE is not included in the POINTERS from the reference FILE, then the FIELDS of that RECORD will not be used to define the values of FIELDS in the TARGET FILE. For example, elements of SOURCE FIELD OUT2 from RECORDS of ISAM FILE E with KEY values of 7, 8, and 9, as shown in Figure 8.6(a), will not be used to define TARGET FIELD K(I,J).

```
                  KEY FIELD OUT1      FIELD OUT2

                        6                 19
      R           ------------------------------
      E                 3                 27
      C           ------------------------------
```

```
O                        4                   33
R                   ------------------------------
D                        5                   64
                    ------------------------------
F                        7                   95
T                   ------------------------------
                         8                  128
                    ------------------------------
                         9                   43
                    ------------------------------
                         1                   29
                    ------------------------------
                         2                  203
```

(a) Contents of ISAM SOURCE FILE E

Figure 8.6 Continued Next Page

I	J = 1	2	3
1	2	4	6
2	5	3	1

(b) Contents of D(I,J)

I	J = 1	2	3
1	203	33	19
2	64	27	29

(c) Contents of K(I,J)

Figure 8.6

Contents of Data Structures in Specifcation of Figure 8.5

Using a keyed FILE as TARGET is an easy way to fill an empt'y SAM or ISAM FILE with data. For example, Figure 8.7 shows the specification you would write to define the FIELDS of keyed ISAM TARGET FILE M. (Here the RECORDS of the keyed FILE will be shaped into a vector rather than a matrix.) Whenever the POINTER gives a KEY that is not already contained in the keyed TARGET FILE, then a new RECORD for that FILE is defined, with its KEY equal to the value of the POINTER FIELD. (This method could also be used to add RECORDS to an already existing keyed FILE.) When the POINTER gives a KEY value which corresponds to a RECORD already contained in the TARGET FILE, nothing is changed and the message

```
FILE filename IS TARGET ONLY,
RECORD UPDATE WITH KEY key-value IS IGNORED
```

is printed.

```
MODULE: A;
SOURCE: B;
TARGET: M;

1 B IS FILE,
  2 C(*) IS RECORD,
    3 D IS FIELD (PIC '9'),
    3 G IS FIELD (CHAR (5));

 1 M IS FILE KEY IS OUT1 ORG IS ISAM,
  2 F IS RECORD,
     3 OUT1 IS FIELD (PIC '9'),
    3 FIELD .(CHAR (5));
```

```
I IS SUBSCRIPT;

POINTER.FT(I) = D(I);

OUT2(I) = G(I);
OUT1(I) = D(I);
```

Figure 8.7

Example Using POINTER with Keyed FILE as TARGET

The most common usage of the POINTER control variable is with a keyed FILE which is used as both SOURCE and TARGET. This allows you to alter certain RECORDS in the keyed FILE while not changing others. For example, you could update the RECORDS of those items in a stock inventory which just arrived in a shipment, while not affecting the RECORDS of your other items. Figure 8.8 shows the ISAM FILE E used as both a SOURCE FILE and a TARGET FILE. This example is basically similar to the two previous ones What is different is that every time one of the variables from the ISAM FILE is mentioned in an equation, it must be accompanied by a keyword prefix NEW or OLD to tell if it refers to the variable before or after the ISAM FILE was updated. (See Section 3.6 for a discussion of the use of the keywords NEW and OLD.) RECORDS with KEYS not included among the list of POINTERS will not be updated before being entered into the TARGET FILE. Unless redefined in an additional assertion, the KEY FIELDS will also be the same in both the SOURCE and TARGET versions of the keyed FILE.

```
MODULE: A;
SOURCE: B,E;
TARGET: E;

1 B IS FILE,
  2 C(*) IS RECORD,
     3 D IS FIELD (PIC '9'),
    3 G IS FIELD (PIC 'Z99');

1 E IS FILE KEY IS OUT1 ORG IS ISAM,
  2 FT IS RECORD,
    3 OUT1 IS FIELD (PIC '9'),
    3 OUT2 IS FIELD (PIC 'Z99');

I IS SUBSCRIPT;

POINTER.OLD.FT(I) = D(I);

 NEW.OUT2(I) = OLD.OUT2(I) + G(I);
```

Figure 8.8

Example with POINTER Using Keyed FILE as both SOURCE and TARGET

8.9. FOUND.X

FOUND.X is a Boolean variable with the same size and shape as POINTER.X, that is, with the same range and number of dimensions. It is also used with keyed FILES, which usually have INDEX SEQUENTIAL organization. A element of FOUND.X contains a value of true, if the corresponding RECORD, whose KEY is given in POINTER.X, actually exists in the keyed FILE. If the RECORD does not exist, then the value of the element of FOUND.X with corresponding subscripts is false.

An example of the use of FOUND.X is the following assertion which can be added to the specification in Figure 8.7. The assertion is

SUBSET.NEW.FT = FOUND.OLD.FT;

and it uses both FOUND and SUBSET control variables. The effect of the assertion is to delete from the updated ISAM FILE all RECORDS which weren't changed (whose KEYS weren't pointed to and found). Only old RECORDS which were included in the update will remain in the keyed TARGET FILE.

The FOUND prefixed variables are defined automatically and there is no need to write an equation to define them. Thus they are referenced in equations only on the right-hand side of the equal sign, i.e. only as independent variables.

8.10. ENDFILE.R

Consider for example a SOURCE file that has RECORD arrays R1, R2 and R3. Then the MODEL system will define automatically arrays of respective shapes (dimensionality and ranges) ENDFILE.R1, ENDFILE.R2 and ENDFILE.R3. Each element of the latter arrays denotes if the corresponding record is the last in the file. Thus the ENDFILE prefixed variables are Boolean. They are defined automatically and there is no need to define them by equations. They can be referenced as independent variables in equations.

8.11. EMPTY.F

The keyword EMPTY can be used to prefix a SOURCE file name, e.g. EMPTY.F. It denotes whether the suffixed SOURCE file is empty, namely has no RECORDs in it. Such variables are Boolean and are defined automatically. They can be referenced as independent variables in equations. They are used mainly to define other attributes of data, such as ranges.

9. DEBUGGING AND DOCUMENTING MODEL SPECIFICATIONS

The topic of debugging in MODEL is very closely related in many respects to progressive modular development. Specifications which have been processed by the MODEL compiler receive information in the output reports and error messages which aid the user in proceeding to the next level of analysis. Whether this information is an indication that the system design has inconsistencies or it is telling the user of an error or oversight (for example a specification statement is missing a semi-colon), he/she must know how to understand the messages and reports so that the prototyping proceeds.

In this chapter we'll discuss three different aspects of debugging. First, a high-level view of the structure of the MODEL compiler with the goal of assisting the user in his/her debugging of MODEL specifications. The compiling process goes through many stages, and the informational, error and warning messages which the compiler generates originate from a particular part in the compilation process. Knowing something of this process will help the user identify the areas of concern in the specification.

Next, we'll discuss the reports generated by the compiler and what information each contains.

And finally, we will introduce the parameters of the MODEL Control File. These may be manipulated by the MODEL specification designer to request or suppress the various reports and/or options available in the MODEL system.

The chapter ends with a discussion of the Test-data Generation facility of MODEL used for development and testing.

9.1. DEBUGGING A MODEL SPECIFICATION

9.1.1. ERROR MESSAGES

Every time a specification is run through the MODEL compiler an "Error" file is created. This file contains messages from the MODEL system which supply the user with valuable information.

Each message issued by MODEL is proceeded by a message type followed by a four character error code. The different message types are:

Message Type	Explanation
Error	A problem in the specification was so severe that MODEL could not continue processing reliably. No code was generated. Correct the error and try again.
Failure	The specification uncovered a problem internal to the compiler. Contact your representative for assistance. Save the offending specification for your representative.
Limit	The specification requires an allocation beyond what is presently allowed. Contact your representative to determine if the limit can be increased.

`Warning`	`The compiler made an assumption about the specification. Verify the cause of the warning and determine whether the assumption is the intended meaning. Modify and recompile accordingly.`

The message type is followed by a four character error code. It is an abbreviation for procedures within MODEL and indicates which issued the message.

The four stages of the MODEL compiler include:

```
1)  Syntax Analysis
2)  Precedence Analysis and Dimension Propagation
3)  Range Propagation
4)  Scheduling and Code generation
```

If an error is detected in a specification, the compiler stops execution at the end of the stage in which the error occurred. The four or five character error codes contain a three character mnemonic which can be used to determine in which stage of the compilation process the message was issued. (A discussion of each of the four stages follows.) In figure 9.1 we have listed the three character prefixes alphabetically along with the stage of the compiler which issues these messages.

MESSAGE MNEMONIC	STAGE
APD	1
ASS	2
BKT	4
BTC	4
CDG	4
CHK	3
CLS	4
CRD	2
CRE	1
DMP	2
EED	2
EHR	2
EMD	2
EMT	2
ENP	2
EVL	4
FED	4
FIR	2
FLM	1
FPK	4
FSB	2
GAS	4
GDL	4
GIO	4
HSR	2
IIX	2
INA	1
INI	1
ISF	1
ITF	1
LEX	1
MIN	1
MNT	1
NRC	1
RGP	3
RTB	3
RTV	1
SAP	1
SCD	4
SFD	1
SFL	1
SPS	1
SVF	1
SVT	1
WID	4
WPL	4
XRF	1

(Note: above stages refer to the following:
1) Syntax Analysis
2) Precedence Analysis and Dimension Propagation
3) Range Propagation
4) Scheduling and Code generation)

Fig. 9.1: Procedures issueing messages and respective stages.

9.1.2. THE STAGES OF THE MODEL COMPILER

We have chosen an example of a specification to illustrate the types of error messages generated by MODEL and what the corresponding equations which trigger these messages look like. The specification illustrated below may be used to sort a Source file with up to 9,999 records. It accomplishes this through creating a two dimensional Interim file with sorted columns. When end-of-file is reached in the Source file, the last column of the Interim file is used to give values to the Target file.

```
Module: Sort;
Source: Datain;
target: Sorted;

1 Datain is file,
  2 Datarec (*) is record,
    3 key_fld is field (char (5));

1 Sorted is file,
  2 Sortrec (*) is record,
    3 sort_fld is field (char (5));

1 inter is file,
  2 rows (*) is group,
    3 column (1:9999) is record,
      4 interfld is fld (char (5));
(i,j) are subscripts;
interfld(i,j)= if i=1 then key_fld(1) else
               if j<i then
                  if interfld(i-1,j)< key_fld(i) then interfld(i-1,j) else
                  if j=1 then key_fld(i) else
                  if interfld(i,j-1)< key_fld(i) then key_fld(i) else
                        interfld(i-1,j-1)
               else
               if interfld(i,j-1)<key_fld(i) then key_fld(i) else
                        interfld(i-1,j-1);

file_size=if endfile.datarec(i) then i;

size.column(i)=i;
size.sortrec=file_size;

sort_fld(j)=if i=file_size then interfld(i,J);
```

9.1.3. STAGE ONE: SYNTAX ANALYSIS

Before any equations, header statements, data declarations or equations are broken down and examined for underlying relationships, every statement in the specification is tested for syntax correctness.

Errors in syntax usually cause the MODEL compiler to terminate without further processing. They may be caused by a missing semi-colon, a misspelled key word or any other violation of the rules for syntax.

The error message will contain a reference to the statement number in the specification where the error occurred. This number points to a line in the listing file which was created at compilation time.

In the following example, modifications causing errors have been made to the specification "Sort".

```
Module: Sort;
Source: Datain_file;
target: Sorted;

1 Datain_file is file,
  2  Datarec (*) is record,
     3  key_fld is field (char (5))

1 Sorted is file,
  2  Sortrec (*) is record,
     3  sort_fld is field (char (5));

1 inter is file,
  2  rows (*) is group,
     etc.
```

We have broken two syntax rules in this specification: First, we have violated the syntax for source file names by making it greater than seven characters. Second, we omitted the semicolon at the end of the data declaration for the same source file.

Upon running the MODEL compiler, the following messages were generated:

```
*ERROR* ISF1: SOURCE FILE NAME "DATAIN_FILE" IS LONGER THAN 7 CHARACTERS.
*ERROR* SAP74: MISSING ';'  AT END OF STATEMENT AT LINE: 6 OF STMT: 4, AT
        SYMBOL "1".
     *STATISTICS: 0 WARNING(S) AND 2 ERROR(S) DETECTED IN SYNTAX ANALYSIS*

     *-*JOB ABORTED DUE TO THE ERROR(S) NOTED ABOVE.
```

The second message points to line 6 of statement 4. The next step is to consult the generated listing shown in figure 9.2.

```
     ***********************************************************************
     *                                                                     *
     *                     SORT MODULE SPECIFICATION                       *
     *                                                                     *
     ***********************************************************************

1      MODULE: SORT;
2      SOURCE: DATAIN_FILE;
3      TARGET: SORTED;

     ***********************************************************************
     *                                                                     *
     *                        FILE DESCRIPTIONS:                           *
     *                                                                     *
     ***********************************************************************


     ***********************************************************************
     *                                                                     *
     *                  DESCRIPTION OF DATAIN_FILE FILE                    *
     *                                                                     *
     ***********************************************************************

4      1 DATAIN_FILE IS FILE,
4        2  DATAREC (*) IS RECORD,
4           3  KEY_FLD IS FIELD (CHAR (5))

4      1 SORTED IS FILE,
4        2  SORTREC (*) IS RECORD,
5           3  SORT_FLD IS FIELD (CHAR (5));

     ***********************************************************************
     *                                                                     *
     *                     DESCRIPTION OF INTER FILE                       *
     *                                                                     *
     ***********************************************************************

5      1 INTER IS FILE,
5        2  ROWS (*) IS GROUP,
5           3  COLUMN (1:9999) IS RECORD,
5              4  INTERFLD IS FLD (CHAR (5));
6      (I,J) ARE SUBSCRIPTS;
7 8    INTERFLD(I,J)= IF I=1 THEN KEY_FLD(1) ELSE
7                    IF J<I THEN
7                      IF INTERFLD(I-1,J)< KEY_FLD(I) THEN INTERFLD(I-1,J) ELSE
7                        IF J=1 THEN KEY_FLD(I) ELSE
7                        IF INTERFLD(I,J-1)< KEY_FLD(I) THEN KEY_FLD(I) ELSE
7                              INTERFLD(I-1,J-1)
7                     ELSE
7                     IF INTERFLD(I,J-1)<KEY_FLD(I) THEN KEY_FLD(I) ELSE
7                              INTERFLD(I-1,J-1);

8 9    FILE_SIZE=IF ENDFILE.DATAREC(I) THEN I;

9 10   SIZE.COLUMN(I)=I;
```

```
10      SIZE.SORTREC=FILE_SIZE;

11      SORT_FLD(J)=IF I=FILE_SIZE THEN INTERFLD(I,J);
```

Fig. 9.2: Specification listing for "Sort".

Notice that the compiler is unable to determine the end of statement 4. The Target file SORTED is included in this statement. In this particular case the compiler is confused with its sequencing. It gives a lot of information and it is up to the user to find the exact location of the error. -MISSING ';' AT END OF STATEMENT AT LINE: 6 OF STMT: 4, AT SYMBOL "1"- The best indicator is the SYMBOL "1" at line 4 of statement 4. This is the symbol which tells the compiler that a semi-colon was missing.

9.1.4. STAGE TWO: PRECEDENCE ANALYSIS AND DIMENSION PROPAGATION

In MODEL, the sequence of program events is not given by the user. Instead, it is determined by the MODEL processor. How this is accomplished is a complex series of events, but there are certain occurrences of which the user should be aware to help in debugging of specifications.

This stage creates a dictionary of all data and equation elements in the specification. Source and Target files are considered external files and any data structure which is not in Source or Target is assumed by the compiler to be an Interim structure.

If an interim field is not defined by any equation, an error message is sent to inform the user. It is probable that the user forgot to write the equation. However, if a field in a Target file is not defined explicitly, the MODEL processor will try to find an implicit source, through name matching, to define that field. The MODEL processor tolerates this kind of incompleteness and saves the user work of writing assertions for merely copying fields from a Source file to a Target file.

The user is allowed to omit subscripts in equations in certain cases which do not lead to ambiguity. These are tolerated by the MODEL processor and are resolved by the process known as Dimension Propagation. The number of dimensions of a data element is defined by counting the number of ranges between it and the FILE statement. Any divergence from this number of subscripts in equations causes this stage to determine whether subscripts were omitted in a logical fashion and if so, the subscripts are implicitly added.

In the following example, the equation for INTERFLD in the specification SORT was modified to introduce an error in Dimension Propagation.

```
interfld(i,j)= if i=1 then key_fld(1) else
               if j<i then
                  if interfld(i-1,j)< key_fld(i) then interfld(i-1) else
                  if j=1 then       etc.
```

Although INTERFLD has two dimensions, the final reference to it is INTERFLD(i-1), implying that there is only one dimension.

After running this specification through the MODEL compiler, the following error message was generated:

```
*ERROR* DMP1: CONTRADICTION IN DIMENSIONALITY
        DETECTED IN THE DATA ELEMENT INTER.INTERFLD WITH 2 DIMENSIONS.
        AASS8 APPEARS TO HAVE 0 DIMENSIONS OMITTED.
        SO, THE DATA ELEMENT INTER.INTERFLD MUST HAVE 1 DIMENSION.
        AGAIN, AASS8 APPEARS TO HAVE 0 DIMENSIONS OMITTED.
        THEREFORE, THE DATA ELEMENT INTER.INTERFLD HAS 2 DIMENSIONS.
```

The message points to field INTERFLD within statement 8, (AASS8), and indicates confusion in dimensionality. At this point the user should consult the listing to determine which statement that is (in this case it is the one we modified) and carefully examine all subscript expressions to determine whether or not a dimension was accidently omitted.

9.1.5. STAGE THREE: RANGE PROPAGATION

If no range is specified for the rightmost dimension of a Source file, the system will assume this to be end-of-file. If two data elements are in the same equation with the same global subscripts, then the system will assume they have the same ranges.

The ranges of an equation's subscripts restrict the number of instances the equation will be executed. A criterion for placing a number of equations in the scope of one loop is that they all have subscripts of the same range. In order to develop a high degree of efficiency in the generated PL/I program, the MODEL compiler identifies all subscripts of the same range.

9.1.6. STAGE FOUR: SCHEDULING AND CODE GENERATION

Most of the error messages which users will encounter come from the first three stages. The final stage executes a series of complex analyses to set up a highly efficient PL/I program. Having begun with a non-procedural specification, this stage takes all the information which the MODEL compiler has determined and initiates the task of structuring and generating the PL/I program.

We are now in a position to understand all the Warning messages generated by the specification "Sort". These messages are:

```
                    --- ERROR(S)/WARNING(S) DETECTED FOR SORT.INP: ---

    *NO ERROR/WARNING DETECTED DURING SYNTAX ANALYSIS*


*WARNING* CRD9: A DECLARATION HAS BEEN SUPPLIED FOR THE UNDECLARED VARIABLE
        "FILE_SIZE".
*WARNING* IIX1: SOME SUBSCRIPTS APPEAR ON THE RIGHT-HAND-SIDE BUT NOT ON THE
        LEFT-HAND-SIDE OF AN ASSERTION. SELECTION IS IMPLIED FOR ",I" IN
        ASSERTION AASS12.
*WARNING* IIX1: SOME SUBSCRIPTS APPEAR ON THE RIGHT-HAND-SIDE BUT NOT ON THE
        LEFT-HAND-SIDE OF AN ASSERTION. SELECTION IS IMPLIED FOR ",I" IN
        ASSERTION AASS9.
*WARNING* RGP7: "INTER.COLUMN,2" AND "SORTED.SORTREC,1" USED AS LEFT-HAND-SIDE
        VARIABLES,  HAVE INCOMPATIBLE RANGES; THEY CANNOT SHARE THE SAME
        SUBSCRIPT.
*WARNING* RGP1: DIMENSION 1 OF "DATAIN.KEY_FLD" IN RANGE SET NUMBER 1 DOES NOT
        HAVE AN EXPLICIT RANGE.
*WARNING* EVL1: THE RANGE FOR DIMENSION 1 OF "DATAIN.DATAREC" NEEDS AN UPPER
        BOUND AND HAS BEEN ASSUMED TO BE 9999.
```

```
*STATISTICS: 6 WARNING(S) AND 0 ERROR(S) DETECTED IN SEMANTIC ANALYSIS*
```

First note that there are no syntax errors. Had there been any, the compiler would have stopped processing.

The CRD and IIX messages are from STAGE TWO. The undeclared variable FILE_SIZE was given a declaration automatically during Precedence Analysis. In statements 9 and 12, (the equations for FILE_SIZE and SORT_FLD respectively), all fields had complete and correct subscript expressions, however the compiler is indicating to the user that there is a dimension mismatch between the two sides of the equation.

The RGP messages are from STAGE THREE. In order to fully understand these messages the user must look at the Range Table for this specification. There it will be found that the compiler had to set up separate ranges for the Target file and the Interim file second dimension even though they share a global subscript and have the same range.

Finally, the EVL messages come from STAGE FOUR. In this stage, the Scheduler determined that the generated PL/I code would need an upper limit to compile successfully and therefore inserted 9999 into the data declaration for DATAREC.

9.2. REPORTS PRODUCED FOR DOCUMENTING A SPECIFICATION

The MODEL compiler produces a series of reports with the dual objective of serving as reference in debugging and providing automated documentation for future maintenance of the specification. In addition to the error and warning messages illustrated above, there are seven reports described below.

The first report - Specification Listing was illustrated in figure 9.2. This report assigns numbers to statements, which are used in error and warning messages to point to respective problem areas.

The other reports are illustrated below through the "Sort" example and briefly described in the following.

The Cross Reference report is illustrated in figure 9.3. It lists all the variables in the specification, showing for each where it has been declared and where it is referenced. It is useful, for instance, if you have a problem with a variable, to find where the variable is declared and referenced. In this way you can evaluate all the involvements of the variable in the specification.

The Flowchart Report is illustrated in figure 9.4. It is not used in debugging a specification for correctness, but only for attempting to make a specification more efficient. This is described in section 10.3. The report shows in a linear flowchart-like form all the events in the program. It is produced at the end of the SCHEDULING STAGE and reports the decisions made by the compiler in the detailed design of the program.

The Range Table Report is illustrated in figure 9.5. It may be produced twice- once before the SCHEDULING STAGE as an aid to debugging if an error is detected prior to scheduling, and once after the SCHEDULING STAGE to indicate final program design decisions. The report first shows the distinct ranges found by the compiler in the specification. This is followed by showing for each variable its dimensions, numbered from left to right, the range of the dimensions, whether the memory allocation is virtual or physical, and its window size.

Figure 9.6 illustrates the Formatted Report. It is a restatement of the Specification Listing with all omitted information filled in. Further, an indentation scheme is used for easy readability. Also the statements are reordered, starting with source declarations and following with intermediate and target declarations - each with the defining equations for its variables. This report is intended solely for program documentation.

The generated PL/1 program is illustrated in figure 9.7. The user need not understand the generated PL/1 code. However, messages issued when the program is executed may include a PL/1 statement number. This may occur especially in dynamic debugging of the program with test data. In such a case you will find at the beginning of the PL/1 statement, which is identified by its corresponding number in the PL/1 Compiler listing and consequently the message during execution, a reference to the number of the respective equation statement in the MODEL specification (indicated as a number within a comment; i.e. /* 8 */). The latter is the statement that needs to be analyzed to find the cause of the problems.

Finally, the File Information report is illustrated in figure 9.8. This is a separate report from the others and is documentary information on file and record sizes by aggregates from their fields. It is useful in determining offset boundaries and record lengths.

```
                 *****  CROSS REFERENCE AND ATTRIBUTES REPORT  *****

              WHERE                                               STATEMENT NO.
  NAME       DECLARED           ATTRIBUTES                          REFERENCE
  ----       --------           ----------                       ----------------

COLUMN              6  RECORD,( 1 SUB-MEMBERS), IN FILE INTER     6,    10,    14

DATAIN              4  FILE,SOURCE,UNSORTED                       2,    15
DATAREC             4  RECORD,( 1 SUB-MEMBERS), IN FILE DATAIN    4,     9,    15

FILE_SIZE          13  FIELD, GENERIC [INTERIM]                   9,    11,    12
I                   7  SUBSCRIPT                                  8,     9,    10,
                                                                 12
INTER               6  FILE,UNSORTED                             14
INTERFLD            6  FIELD, CHARACTER(5) IN FILE INTER          6,     8,    12

J                   7  SUBSCRIPT                                  8,    12
KEY_FLD             4  FIELD, CHARACTER(5) IN FILE DATAIN         4,     8

ROWS                6  GROUP,( 1 SUB-MEMBERS), IN FILE INTER      6

SORT                1  MODULE NAME
SORTED              5  FILE,TARGET,UNSORTED                       3
SORTREC             5  RECORD,( 1 SUB-MEMBERS), IN FILE SORTED    5,    11

SORT_FLD            5  FIELD, CHARACTER(5) IN FILE SORTED         5,    12

(ENDFILE)
 .DATAREC        ----  DATA STRUCTURE PARAMETER                   9
(SIZE)
 .COLUMN         ----  DATA STRUCTURE PARAMETER                  10
 .SORTREC        ----  DATA STRUCTURE PARAMETER                  11
```

Fig. 9.3: Cross Reference and Attributes Report for "Sort".

```
                        *****  FLOWCHART REPORT  *****

              NEST
NAME          LVL: DESCRIPTION                 EVENT
----          ---- -----------                 -----

SORT               MODULE NAME                 PROCEDURE HEADING
DATAIN             FILE                        OPEN FILE
                 1 ITERATION: RANGE NO. 1      FOR $I1 UNTIL END OF FILE
DATAREC            RECORD IN FILE DATAIN       READ RECORD
[ENDFILE]-v
DATAREC            SPECIAL NAME
AASS10             EQUATION
KEY_FLD            FIELD IN RECORD DATAREC
AASS9              EQUATION
[SIZE]-v
COLUMN             SPECIAL NAME                TARGET OF EQUATION: AASS9
                 1 END ITERATION               FOR $I1
                   FIELD                       TARGET OF EQUATION: AASS10
AASS11             EQUATION
[SIZE]-v
SORTREC            SPECIAL NAME                TARGET OF EQUATION: AASS11
                 1 ITERATION: RANGE NO. 1      FOR $I1 UNTIL END OF FILE
                 2 ITERATION: RANGE NO. 2      FOR $I2 UNTIL SIZE.X SPECIFIED
AASS8              EQUATION
INTERFLD           FIELD IN RECORD COLUMN      TARGET OF EQUATION: AASS8
COLUMN             RECORD IN FILE INTER        WRITE RECORD
                 2 END ITERATION               FOR $I2
                 2 ITERATION: RANGE NO. 3      FOR $I2 UNTIL SIZE.X SPECIFIED
AASS12             EQUATION
                 2 END ITERATION               FOR $I2
ROWS               GROUP IN FILE INTER
                 1 END ITERATION               FOR $I1
INTER              FILE                        CLOSE FILE
                 1 ITERATION: RANGE NO. 3      FOR $I1 UNTIL SIZE.X SPECIFIED
SORT_FLD           FIELD IN RECORD SORTREC     TARGET OF EQUATION: AASS12
SORTREC            RECORD IN FILE SORTED       WRITE RECORD
                 1 END ITERATION               FOR $I1
SORTED             FILE                        CLOSE FILE
$YSGEN1            GROUP
$YSGEN2            GROUP
                   END
```

Fig. 9.4: Flowchart Report for "Sort".

```
                         *****  RANGE TABLE  *****

----------------------------------------------------------------------------
|RANGE NO.  |  RANGE DEFINITION          WHERE DEFINED                      |
----------------------------------------------------------------------------
|         1 |   END OF FILE                DATAIN.DATAREC                   |
|         2 |   SIZE                       INTER.COLUMN                     |
|         3 |   SIZE                       SORTED.SORTREC                   |
----------------------------------------------------------------------------
                            *****  RANGE TABLE  *****

----------------------------------------------------
|                                  | RANGE NO.     |
|                                  |    1    2    3 |
----------------------------------------------------
----------------------------------------------------
| NODE NAME                        | DIMENSION NO.  |
----------------------------------------------------
|*ASSERTION(S):                                    |
| 10                               |  1V            |
| 12                               |  2V        1V  |
| 8                                |  1V        2V  |
| 9                                |  1V            |
|*QUALIFIED_NAME:(DATAIN.)                         |
| DATAREC                          |  1V            |
| KEY_FLD                          |  1P            |
|*QUALIFIED_NAME:(ENDFILE.DATAIN.)                 |
| DATAREC                          |  1V            |
|*QUALIFIED_NAME:(INTER.)                          |
| COLUMN                           |  1V   2V       |
| INTERFLD                         |  1V2  2P       |
| ROWS                             |  1V            |
|*QUALIFIED_NAME:(SIZE.INTER.)                     |
| COLUMN                           |  1P            |
|*QUALIFIED_NAME:(SORTED.)                         |
| SORTREC                          |            1V  |
| SORT_FLD                         |            1P  |
|*GLOBAL SUBSCRIPT:                                |
| I                                |  1V            |
| J                                |            1V  |
----------------------------------------------------
NOTE: ENTRY COL.1-DIMENSION NUMBER
                2-PHYSICAL(P)/VIRTUAL(V) DIMENSION
                3-WINDOW SIZE, IF MORE THEN ONE
```

Fig. 9.5: Range Table for "Sort".

```
                    *****  FORMATTED REPORT  *****

 *********************************************************************
 *                                                                   *
 *                       SORT MODULE SPECIFICATION                   *
 *                                                                   *
 *********************************************************************

MODULE: SORT;
SOURCE: DATAIN;
TARGET: SORTED;

 *********************************************************************
 *                                                                   *
 *                              DATA DESCRIPTION:                    *
 *                                                                   *
 *********************************************************************


 *********************************************************************
 *                                                                   *
 *                         DESCRIPTION OF DATAIN                     *
 *                                                                   *
 *********************************************************************

1 DATAIN IS FILE,
  STORAGE NAME IS NSTGNM1
  ORG IS SAM,
  2 DATAREC IS RECORD ,
    3 KEY_FLD IS FIELD (CHAR(5));

 *********************************************************************
 *                                                                   *
 *                           INTERIM SPECIFICATION                   *
 *                                                                   *
 *********************************************************************

1 INTER IS FILE,
  STORAGE NAME IS NSTGNM3
  ORG IS SAM,
    2 ROWS IS GROUP ,
      3 COLUMN IS RECORD ,
        4 INTERFLD IS FIELD (CHAR(5));
  1 FILE_SIZE IS FIELD (FLOAT BINARY(53));

       /* ASSERTION(S) FOR INTERIM(S) */

 /*9*/
FILE_SIZE =
IF ENDFILE.DATAIN.DATAREC (I )
THEN I ;

 /*8*/
INTERFLD (I ,J )=
IF I =1
THEN DATAIN.KEY_FLD (1 )
```

```
ELSE
  IF J <I
  THEN
    IF INTER.INTERFLD ((I -1 ),J )<DATAIN.KEY_FLD (I )
    THEN INTER.INTERFLD ((I -1 ),J )
    ELSE
      IF J =1
      THEN DATAIN.KEY_FLD (I )
      ELSE
        IF INTER.INTERFLD (I ,(J -1 ))<DATAIN.KEY_FLD (I )
        THEN DATAIN.KEY_FLD (I )
        ELSE INTER.INTERFLD ((I -1 ),(J -1 ))
  ELSE
    IF INTER.INTERFLD (I ,(J -1 ))<DATAIN.KEY_FLD (I )
    THEN DATAIN.KEY_FLD (I )
    ELSE INTER.INTERFLD ((I -1 ),(J -1 ));

 /*10*/
SIZE.INTER.COLUMN (I )=I ;

 /*11*/
SIZE.SORTED.SORTREC =INTERIM.FILE_SIZE ;

 **********************************************************************
 *                                                                    *
 *                      SUBSCRIPT(S) SPECIFICATION                    *
 *                                                                    *
 **********************************************************************

I SUBSCRIPT;
J SUBSCRIPT;

 **********************************************************************
 *                                                                    *
 *                        DESCRIPTION OF SORTED                       *
 *                                                                    *
 **********************************************************************

1 SORTED IS FILE,
  STORAGE NAME IS NSTGNM2
  ORG IS SAM,
  2 SORTREC IS RECORD ,
    3 SORT_FLD IS FIELD (CHAR(5));

       /* ASSERTION(S) FOR FILE(SORTED) */

 /*12*/
SORT_FLD (J )=
IF I =INTERIM.FILE_SIZE
THEN INTER.INTERFLD (I ,J );

 **********************************************************************
 *                                                                    *
 *                        END OF FORMATTED REPORT                     *
 *                                                                    *
 **********************************************************************
```

Fig. 9.6: Formatted Report for "Sort".

```
                    /************************************/
                    /*           PL/I PROGRAM           */
                    /************************************/


SORT: PROCEDURE OPTIONS(MAIN);
DCL $MALSTR CHAR(1);
DCL DATAINS RECORD SEQL INPUT;
DCL $FSTDATAINS BIT(1) INIT('1'B);
DCL ENDFILE$DATAINS BIT(1) INIT('0'B);
DCL $FB36 CHAR(5) VARYING INIT('');
DCL $FX36 FIXED BIN(31,0);
DCL $RV37 CHAR(5) VARYING;
DCL $RX37 FIXED BIN(31,0);
DCL $EOF37 FIXED BIN(31,0);
DCL $RV47 CHAR(5) BASED(ADDR(SORTED)) ;
DCL $W_$D43(2) FIXED BIN(31,0);
CALL $INITWIN($W_$D43,2);
DCL SORTEDT RECORD SEQL OUTPUT;
DCL $FSTSORTEDT BIT(1) INIT('1'B);
OPEN FILE(SORTEDT) OUTPUT;
DCL $ERROR_BUF CHAR(270) VAR;
DCL ERRORF FILE RECORD OUTPUT;
DCL ERRORF_BIT BIT(1) STATIC INIT('1'B);
DCL $ERROR FIXED BIN(15,0) INIT(0);
DCL $NOT_DONE(20) BIT(1);
DCL $TMP_VAL FLOAT BIN;
DCL ($RD_LP$,$R_L) LABEL;
DECLARE
   1 DATAIN,
    2 DATAREC,
     3 KEY_FLD(9999) CHAR(5);
DECLARE
   1 INTER,
    2 ROWS,
     3 COLUMN,
      4 INTERFLD(9999,2) CHAR(5);
DECLARE
   1 SORTED,
    2 SORTREC,
     3 SORT_FLD CHAR(5);
DECLARE
   1 INTERIM,
    2 SIZE$D49 FIXED BIN(31,0) ,
    2 FILE_SIZE BIN FLOAT(53),
    2 $YSGEN1,
     3 SIZE$D50(9999) FIXED BIN(31,0) ,
    2 $YSGEN2,
     3 ENDFILE$D51 BIT(1) ;
DCL $I1 FIXED BIN(31,0);
DCL $I2 FIXED BIN(31,0);
DCL (TRUE,SELECTED) BIT(1) INIT('1'B);
DCL (FALSE,NOT_SELE,NOT_SELECTED) BIT(1) INIT('0'B);
$ROTATE: PROCEDURE(WIN_VEC,LEN); /* $START$ */
/*$PARMS: 01,9,9 */
  DCL WIN_VEC(*) FIXED BIN;
  DCL (I,LEN,TEMP) FIXED BIN;
  TEMP=WIN_VEC(1);
  DO I = 2 TO LEN;
    WIN_VEC(I-1)=WIN_VEC(I);
  END;
  WIN_VEC(LEN)=TEMP;
END $ROTATE;   /* $END$ */
$INITWIN: PROCEDURE(WIN_VEC,LEN); /* $START$ */
/*$PARMS: 01,9,9 */
  DCL WIN_VEC(*) FIXED BIN;
  DCL (I,LEN) FIXED BIN;
  DO I=1 TO LEN;
    WIN_VEC(I)=I;
  END;
END $INITWIN;   /* $END$ */
ON ENDFILE(DATAINS) BEGIN;
ENDFILE$DATAINS='1'B;
 $FB36=COPY(' ',5);
 END;
DCL LRD$37 BIT(1) INIT('0'B);
ON UNDEFINEDFILE(ERRORF)  ERRORF_BIT='0'B;
DECLARE PLI$_CNVERR GLOBALREF VALUE FIXED BIN(31);
DECLARE RMS$_RLK GLOBALREF VALUE FIXED BIN(31);
ON ERROR BEGIN;
IF ONCODE()=RMS$_RLK THEN GOTO $RD_LP$;
IF $ERROR=0 THEN CALL RESIGNAL();
IF ONCODE()=PLI$_CNVERR THEN DO;
  $ERROR=1;
  IF ERRORF_BIT & $ERROR>0 THEN WRITE FILE(ERRORF) FROM ($ERROR_BUF);
 END;
  ELSE CALL RESIGNAL();
END;
OPEN FILE(DATAINS) INPUT SEQL RECORD;
$I1 =0;
DO  WHILE(^ENDFILE$DATAINS);
  $I1 = $I1 +1;
  IF ^$FSTDATAINS THEN READ FILE(DATAINS) INTO ($RV37);
  IF ENDFILE$DATAINS THEN GO TO $LB37;
  $RX37=1;
  $ERROR_BUF=$RV37;
  $RV37=$RV37||COPY(' ',5);
  IF ENDFILE$D51 THEN;
  ELSE ENDFILE$D51= LRD$37;
  LRD$37=ENDFILE$DATAINS;
  IF $FSTDATAINS THEN $FSTDATAINS = '0'B;
  /* 9 */IF ENDFILE$D51 THEN INTERIM.FILE_SIZE=$I1;
   ELSE ;
  DATAIN.KEY_FLD($I1)=SUBSTR($RV37,$RX37,5) ;
  $RX37=$RX37+5 ;
  /* 10 */SIZE$D50($I1)=$I1;
END;
$LB37:$EOF37=$I1;
/* 11 */SIZE$D49=INTERIM.FILE_SIZE;
DO $I1 =1 TO SIZE$D49;
  $I2 =0;
  DO $I2 = 1 TO $EOF37;
    /* 8 */IF $I2=1 THEN INTER.INTERFLD($I2,$W_$D43(2))=
    DATAIN.KEY_FLD(1);
```

```
      ELSE IF $I1<$I2 THEN IF INTER.INTERFLD($I2-1,$W_$D43(2))<
     DATAIN.KEY_FLD($I2) THEN INTER.INTERFLD($I2,$W_$D43(2))=
     INTER.INTERFLD($I2-1,$W_$D43(2)); ELSE IF $I1=1 THEN
     INTER.INTERFLD($I2,$W_$D43(2))=DATAIN.KEY_FLD($I2); ELSE IF
     INTER.INTERFLD($I2,$W_$D43(1))<DATAIN.KEY_FLD($I2) THEN
     INTER.INTERFLD($I2,$W_$D43(2))=DATAIN.KEY_FLD($I2); ELSE
     INTER.INTERFLD($I2,$W_$D43(2))=INTER.INTERFLD($I2-1,$W_$D43(1));
     ELSE IF INTER.INTERFLD($I2,$W_$D43(1))<DATAIN.KEY_FLD($I2) THEN
     INTER.INTERFLD($I2,$W_$D43(2))=DATAIN.KEY_FLD($I2); ELSE
     INTER.INTERFLD($I2,$W_$D43(2))=INTER.INTERFLD($I2-1,$W_$D43(1));
     /* 12 */IF $I2=INTERIM.FILE_SIZE THEN SORTED.SORT_FLD=
     INTER.INTERFLD($I2,$W_$D43(2)); ELSE ;
  END;
  WRITE FILE(SORTEDT) FROM ($RV47);
  CALL $ROTATE($W_$D43,2);
END;
$I1 =0;
DO $I1 = 1 TO $EOF37;
  DO $I2 =1 TO SIZE$D50($I1);
  END;
END;
CLOSE FILE(SORTEDT);
RETURN;
END SORT;
```

Fig. 9.7: PL/1 Listing for "Sort".

```
                *********   FILE INFORMATION REPORT   **********

MODULE: SORT
SOURCE: DATAINS SEQL     LENGTH: 5    TYPE: F

 DCL                                                  MAX
LEVEL        NODE  NAME                       TYPE  REPEAT   LENGTH   INDEX
-------------------------------------------------------------------------
  1   DATAIN                                  FILE       *        *     *
  2   DATAREC                                 RECD       1        5       1
  3   KEY_FLD                                 FLD        1        5       1

TARGET: SORTEDT SEQL     LENGTH: 5    TYPE: F

 DCL                                                  MAX
LEVEL        NODE  NAME                       TYPE  REPEAT   LENGTH   INDEX
-------------------------------------------------------------------------
  1   SORTED                                  FILE       *        *     *
  2   SORTREC                                 RECD       1        5       1
  3   SORT_FLD                                FLD        1        5       1
```

Fig. 9.8: File Information Report for "Sort".

9.3. PARAMETERS OF THE MODEL COMPILER

The MODEL compiler and its outputs are controlled via a System Input (SYSIN) Control File. It contains a series of control words, or parameters, which activate or inhibit certain stages and/or outputs of MODEL. There is also a means for the System Managers to "watch" the compiler as it executes; although you'll probably never use this feature, it is helpful to know it exists and it can aid in debugging strange or complex problems.

The parameters, separated by commas, appear on the first few lines of the SYSIN control file with explanatory notes (never recognized by MODEL) on the latter lines. MODEL only recognizes those parameters in upper case. The upper/lower case toggle provides a convenient, easy means for activating(upper case) or deactivating(lower case) control options and allows storage of all parameters for quick reference and usage. These parameters and their effects are described in Figure 9.9.

PARAMETER	ACTION/EFFECT
AUTODOCU	Activates production of the File Information report.
BPTYPERR	Allows generation of PL/1 code even though the data-type checking phase reported errors.
DATA	Activates the Test-data generator (discussed in section 9.4).
DEBUG=()	For System Manager's use; allows trace of compiler execution by phases (the phases being identified by a numeric code in the parentheses).
NOASSMEM	Inhibits production of the Formatted Report.
NODEBUG	Disables the internal System Manager's trace.
NOFLOW	Inhibits production of the Flowchart Report.
NOLIST	Inhibits production of the PL/1 listing.
NOPROG	Inhibits generation of the PL/1 program.
NORNGTAB	Inhibits production of the Range Table report.
NOXREF	Inhibits production of the Cross Reference report.
RDEBUG	Activates a Run-time debugging mechanism which adds tracing statements to the generated PL/1 code (for severe debugging purposes in a MODEL specification utilizing advanced features of the system).
STCKOPT	Activates optimization of windowed variables and loop control. This option should normally be used.

Fig. 9.9: MODEL Control Parameters and their Effects.

9.4. THE TEST-DATA GENERATOR

For purposes of testing a developed MODEL specification, there is a mechanism for generating testing data. Before activating this mechanism, it is necessary to modify your specification slightly (for test-data generation only). First, delete all current Target and Interim data: file names, declarations, and defining assertions (don't worry about subscripts); next, make all current Source file names Target names keeping any range definitions and adding any range definitions required either with constants or assertions (for instance, source file records often have a (*) repetition indicating end-of-file; this will cause an error since that range will be undefined when associated with a target file - use of a constant or range-defining assertion will be needed); finally, add any equations to define the variables in the files. There are a series of functions in Figure 9.11 which are useful in the definition of data. If no equations are added, the compiler randomly creates values to assign to the new target data.

Figure 9.10 illustrates the "Sort" example as it might appear modified to generate testing data. Notice that the (*) originally specified for Datarec has been changed to a constant. If the range had not been specified in some way, an error message would have been issued and processing stopped. After the modifications are made, we must add the control parameter "DATA" to the SYSIN file and execute the MODEL compiler. The generated PL/1 code can then be executed to produce a file (or files) of test data.

```
Module: Sort;
Target: Datain;

1 Datain is file,
  2  Datarec (100) is record,
     3  key_fld is field (char (5));

(i,j) are subscripts;


/**NOTE:  This is actually enough for the generator, but below is an
          assertion illustrating the use of two controlling functions.  **/

key_fld(i)= if i<10 then randomchar(5)
                     else choose('JANRY,FEBRY,MARCH,APRIL');
```

Fig. 9.10: "Sort" modified for the Test-data Generator.

Function Name	Description
RANDOMINDEX(n)	Creates a random integer from 1 to n.
RANDOMCHAR(n)	Creates a random character string n characters in length (n must be between 1 and 100).
RANDOMBIT(n)	Creates a random bit string of length n (n being an integer between 1 and 16).
CHOOSE(s)	Chooses a phrase from s, where s is a sequence of characters separated by commas which form phrases of 32 characters maximum, with a limit of 32 phrases.

Fig. 9.11: Controlling functions for Test-data Generator.

10. RESTATING A SPECIFICATION TO IMPROVE EFFICIENCY OF PRODUCED PROGRAMS

10.1. USER'S GUIDANCE ROLE IN PRODUCING EFFICIENT PROGRAMS

The MODEL compiler generates highly efficient programs based on the specification provided by a user. However, the compiler does not have information on the environment in which the produced program is intended to operate, especially the values of the expected input data. The compiler must therefore assure that all the produced programs will operate correctly independently of the external environment. This limits the ability of the compiler to optimize the produced programs. The compiler must adhere closely to the submitted specification. Different statements of the same problem may produce programs with differing efficiencies. This chapter discusses how the user can modify the specification to provide the compiler with guidance that will result in further improving efficiency of the generated program.

As noted previously, composition of a MODEL specification can be carried out progressively, first composing individual parts, and then merging the respective parts. Further, each part can be developed in two steps: first obtaining a correct specification and only later proceeding with modifications which lead to improved efficiency.

The optimization of program memory is a major component of the MODEL compiler. This is where you can have most impact on efficiency by refining your specification. The other optimization activities of the MODEL compiler - input-output and computations - are impacted less by changes in the specification.

This chapter discusses three areas in which a specification can be further refined:

1. Through simplifying the specification and using fewer variables and equations, or variables with fewer dimensions or fewer elements.
2. Through choice of forms of subscript expressions and choice of types of control variables that allow a generated program to pass over dimensions of arrays in uniform order. In passing over the dimension the program can retain in memory, at any time, only a small window of the elements along the dimension. A dimension of a variable which is only partially retained in memory is shown in the Range Report as being *virtual*(V), otherwise it is shown as *physical*(P). Your role is to investigate, and if possible modify the specifications so that a dimension of a variable previously noted as physical is changed to virtual.
3. Finally, when a program must contain in memory large tables which require excessive space, it is possible to use external ISAM files for holding and updating the respective tables.

10.2. REDUCING THE NUMBER OF VARIABLES AND EQUATIONS IN A MODEL SPECIFICATION

Simplicity and elegance in stating a specification are important not only for ease of understanding and maintaining the specification, but also for its efficiency. A MODEL specification, in addition to source and target data, typically consists of chains of interim arrays which are transformed through equations. Your objective should be to make these chains as short as possible and specify transformations only when values of some elements are modified. Especially copying values is wasteful. The fewer variables and fewer equations the more readable and efficient is your specification. Also the less dimensions or smaller range of a dimension the more efficient is the produced program. On the first approach you would typically allow redundancy or excess of variables and equations in your specification and focus only on correctness. It is a good idea then to review your initial approach and reduce the number of interim variables and equations or their dimensions and ranges as much as possible.

10.3. ATTAINING VIRTUAL DIMENSIONS OF VARIABLES

10.3.1. USE OF APPROPRIATE SUBSCRIPT EXPRESSIONS

The MODEL compiler attempts to optimize memory only of those dimensions of array variables that are referenced with subscript expression of the following two forms

```
I-K
X(I - K1) - K2

I is a subscript variable
X(I) is a sublinear subscript variable
K, K1, or K2 are 0 or positive integers
```

Examples are A(I), A(I-5), A(X(I-2)-1).

The MODEL compiler does not attempt to optimize memory use of variable dimensions which are referenced with other forms of subscript expressions. The other forms are called *general subscript expressions*. Examples are V(I+2), V(5) etc.

Consider the following example.

```
X(10) IS FIELD (NUM(4));
X(1) = 1;
X(I+1)= IF I< 10 THEN (I+1) * X(I);
Y = SUM (X(I),I);
```

Y is the sum of ten products X(I) of all the positive integers from 1 to I. Thus X(1)=1, X(2)=2, X(3)=X(2)*3, etc., up to I=10.

Since X is referenced with the general subscript expression I+1, it will be allocated by the MODEL compiler 10 elements in memory and will be reported in the Range Report as having a physical dimension. However the above example can be restated without using a general subscript expression as:

```
X (10) IS FIELD (NUM(4));

X(I) = IF I=1 THEN 1
              ELSE I*X(I-1);
```

This will allow X to have a virtual dimension memory, allocating only two elements (reduced to one element in further optimization).

To obtain a more memory-efficient program you will have to examine your use of subscript expressions and wherever possible reform them to avoid use of general subscript expressions.

10.3.2. USE OF APPROPRIATE RANGE SPECIFICATIONS

You can define a variable range of Y(J) in the specification by composing an equation that defines either the variable END.Y(J) or the variable SIZE.Y. Use of SIZE.Y appears more efficient as it has one less dimension than END.Y(J). However, in some cases this may cause Y(J) to become physical. This is because Y(J) is dependent in its entirety on SIZE.Y, while individual elements of Y(J) are dependent only on corresponding elements of END.Y(J). The latter allows us a single pass on the elements of Y(J) to determine their value and range simultaneously.

This is illustrated in the following example.

```
        MOD: EXAMPLE;
        SOURCE: FIN;
        TARGET:  FOUT;

        1 FIN IS FILE,
          2 RIN IS REC,
            3 X(1:100) IS FIELD (CHAR(4));

        1 FOUT IS FILE,
          2 ROUT IS REC,
            3 Y (1:100) IS FIELD (CHAR(4));

/*Eq1/  Y(J)=X(J);
        I,J ARE SUBSCRIPTS;

Consider the alternatives

        /* Alt1*/   SIZE.Y = IF X(I) = 'LAST' THEN I;

        /* Alt2*/   END.Y(J) = X(J) = 'LAST';
```

Y is a vector with the last element containing the value 'LAST;. X is a vector of equal or greater range than Y. If we use the equation Alt1 that defines SIZE.Y, then it will be necessary to pass on X twice, first to find the value of SIZE.Y, which must precede evaluation of Y and then a second time to define the respective elements of Y from equation Eq1. This will force X to be physical. However this is not so if we use equation Alt2 to define END.Y(J). In this case in a single pass on X it is possible to define both the range of Y and its respective elements.

The flowchart report of both cases is shown below.

Alternative 1:

```
        FIN
        RIN
+------ ITERATION ON I
|       X
|       Alt1
+------ END ITERTAION ON I
        SIZE.Y
+------ ITERATION ON J
|       Eq2
|       Y
+------ END ITERATION ON J
        ROUT
        FOUT
```

Alternative 2:

```
        FIN
        RIN
+------ ITERATION ON J
|       X
|       Eq2
|       Y
|       Alt2
|       END.Y
+------ END ITERATION ON J
        ROUT
        FOUT
```

10.3.3. FINDING THE EQUATIONS TO BE MODIFIED

After compiling a specification you should review the range report to verify which variable dimensions are physical. The checking for general subscript expressions was described above but there may be also precedence conditions that caused a variable dimension to become virtual. At this point you may investigate whether the equations in the specification can be changed to allow the respective variable dimension to become virtual. The problem is that the specification may be very large and it is difficult to find which equations should be modified. The Flowchart Report can help you. If the variable dimension is physical then the produced flowchart must have more than one iteration shown for that dimension. Locate these iterations in the flowchart, then one or more of the variables listed between the iterations are the ones that caused the splitting of the iterations and also the making of the respective variable dimension physical.

As an example examine the flowcharts above. In Alternative 1, the flowchart shows SIZE.Y is between the two iterations. In Alternative 2 obtained by replacing the equation Alt1 with Alt2 (that defines, END.Y(J)), we obtained a single iteration on J. Upon examining the Range Report we would find that the dimension of X has been changed from physical to virtual for alternative 2.

10.4. REPLACING LARGE PHYSICAL DIMENSION TABLES BY ISAM FILES.

Some applications require referencing variables that have a very large number of elements. Such a table sometimes can be referenced only through use of a general subscript expression, and therefore the respective dimension must by physical. The table may be so large as to make the entire specification impractical. In such cases it is generally possible to place the table in an ISAM file. References to this file are replaced by input/output operations.

This is illustrated in the example below. Consider source data consisting of item numbers of merchandize that must be priced. The price table is enormous, consisting of 100,000 prices in a file called PRICE_TABLE, order by item number.

Two alternatives are shown. The entire specification is shown under Alternative 1. In statement 5, the PRICE_TABLE.PRICE(ITEM_NUMI) is referenced with a general subscript expression. Therefore it will be allocated a physical dimension in memory (shown in the Range Report).

In Alternative 2 we make three changes. First, the PRICE_TABLE file is changed from SEQ to ISAM file organization. Second, we add a POINTER.RTAB variable to define the value of the key to the table. Then statement 5 can be modify to eliminate the need to use a general subscript expression.

This technique can be used quite generally, whenever the memory space for a physical variable dimension is not acceptable.

Alternative 1

```
        MOD:  PRICING:
        SOURCE: IN, PRICE_TABLE;
        TARGET: OUT;

STMT

 1      1  IN IS FILE,
           2 RIN(*)IS REC, IS FIELD
             3 ITEM_NUMI(NUM (5));

 2      1 PRICE_TABLE IS FILE,
          2 RTAB(100000)IS REC,
            3 ITEM_NUMR IS FIELD (NUM(5)),
            3 PRICE IS FIELD (PIC'$ZZZZ.V99');

 3       1 OUT IS FILE,
           2 ROUT(*)IS REC,
             3 ITEM_NUMII IS FIELD(NUM(5));
             3 PRICE IS FLD (PIC '$ZZZZ.V99');

 4       I IS SUBSCRIPT;

 5       OUT.PRICE(I) = PRICE_TABLE.PRICE(ITEM_NUMI);
```

Alternative 2

```
        Add to first line statement 2

                ORG = ISAM, KEY = ITEM_NUMR,

        Add statement

6       POINTER.RTAB(I) = ITEM_NUM(I);

        Change statement 5 to

5       OUT.PRICE(I) = PRICE_TABLE.PRICE(I)
```

11. MODEL FUNCTIONS

11.1. LIST OF FUNCTIONS IN MODEL

Section 4.10 discussed the syntax for using functions in MODEL. It also explained how users of MODEL can create their own functions for general use.

The present chapter lists the functions and gives a brief description of what each one does. The functions are discussed in two sections: The first describes functions which are also used in the PL/I programming language. These are known as "Built-in PL/I functions. (Further documentation on these functions may be found in a PL/I programming manual). Functions to be discussed in this section include:

FUNCTION	PURPOSE OF FUNCTION
ABS	Absolute value
ACOS	Arc Cosine
ADD	Addition
ASIN	Arc Sine
ATAN	Arc Tangent
BIT	Conversion from Character to Bit
CEIL	Next highest integer
CHAR	Conversion from Arithmetic to Character
COPY	Copy and concatenate strings
COS	Cosine
DATE	Returns current date, (yymmdd)
DECIMAL	Conversion from Arithmetic to Decimal
DIVIDE	Division
EXP	base e raised to a power
FIXED	Conversion from Arithmetic to Fixed
FLOAT	Conversion from Arithmetic to Float
FLOOR	Next lowest integer
HIGH	Returns string of High-values
INDEX	position of substring within string
LENGTH	Length of string
LOG	Natural Logarithm

MAX	Larger of two expressions
MIN	Smaller of two expressions
MOD	Modulo function
MULTIPLY	Multiplication
ROUND	Rounds fixed point decimal
SIGN	Returns +1, -1 or 0 for arithmetic
SIN	Sine
SUBSTR	Substring
TAN	Tangent
TIME	Time of day, (hhmmsshh)
TRANSLATE	Replace string occurrences with translation
UNSPEC	Conversion to binary
VERIFY	Compare two strings for inequalities

The second section describes those functions which were developed for use with the MODEL system, but which are not found in the PL/I language. These include:

FUNCTION	PURPOSE OF FUNCTION
AMAX	Maximum value in an array of elements
AMIN	Minimum value in an array of elements
CHOOSE	Random choice from a list
DEPENDS_ON	To indicate dependence on record arguments.
EXIST	Test for presence of condition along a dimension
FALSE	value '0'B
RANDOMBIT	To generate random bits
RANDOMCHAR	To generate random characters
RANDOMINDEX	To generate random integer
RUNSUM	Cumulative totals along dimension
SUBLINEAR	Define sublinear index
SUM	Sum of elements along a dimension

TRUE	value '1'B
WHICH	To identify index when condition is met along a dimension

The last section is this chapter discusses definition of additional function by the user.

11.2. BUILT-IN PL/I FUNCTIONS

11.2.1. ARITHMETIC FUNCTIONS

ABS(x)

ABS returns the absolute value of a given expression x. (it is the positive value of x)

ACOS(x)

The ACOS function returns a floating-point value that is the arc (inverse) cosine of an arithmetic expression x, where x is a number between -1 and +1. The result is a floating-point value in radians.

ADD(x1,x2,p,q)

The ADD function returns the sum of two arithmetic expressions, x1 and x2, with a specified precision p and a scale factor q.

```
x1  first value to be summed

x2  second value to be summed

p   An unsigned integer constant between 1 and 31.  This number represents
the total number of decimal digits used to represent the result.

q   An integer less than or equal to p.  This number represents the number
of fractional digits in the result.
```

ASIN(x)

The ASIN function returns a floating-point value that is the arc (inverse) sine of an arithmetic expression x, where x is a number between -1 and +1. The result is a floating-point value in radians.

ATAN(x)

The ACOS function returns a floating-point value that is the arc (inverse) tangent of an arithmetic expression x. The result is a floating-point value in radians.

CEIL(x)

CEIL returns the smallest integer greater than or equal to a given value x.

COS(x)

The COS function returns a floating-point value that is the cosine of an arithmetic expression x, where x represents an angle in radians.

DECIMAL(x)

DECIMAL returns the decimal representation of a given value x.

```
x   value to be converted to decimal base.
```

The precision of the result is determined from the rules for base conversion.

DIVIDE(x1,x2,p,q)

The DIVIDE function returns the quotiend of two arithmetic expressions, x1 and x2, with a specified precision p and a scale factor q. (Note: For greatest precision, it is recommended that this function be used for all non-floating point variables).

```
x1  first value to be divided

x2  second value to be divided

p   An unsigned integer constant between 1 and 31.  This number represents
the total number of decimal digits used to represent the result.

q   An integer less than or equal to p.  This number represents the number
of fractional digits in the result.  (Note: If x1 or x2 is floating point,
q must be 0).
```

EXP(x)

The EXP built-in function returns a floating-point value that is the base e to the power of an arithmetic expression x.

FIXED(x1,x2,x3)

FIXED returns the fixed-point representation of a given value x1 with a precision specified by x2 and x3.

```
x1  value to be converted to fixed-point scale.

x2  unsigned decimal integer constant specifying
    the precision.  Range, 0 to 31.

x3  decimal integer constant, specifying the
    scale factor of the result.  Range, 0 to 31.
```

FLOAT(x1,x2)

FLOAT returns the floating-point representation of a given value x1 with a precision specified by x2.

```
x1  value to be converted to floating-point scale.

x2  unsigned decimal integer constant specifying the
    total precision of the result.  Range, 1 to 34.
```

FLOOR(x)

FLOOR returns the largest integer less than or equal to a given value x.

LOG(x)

The LOG built-in function returns a floating-point value that is the base e (natural) logarithm of an arithmetic expression x. The expression x must be greater than zero.

MAX(x1,x2)

MAX returns, from a set of two arguments, the value of the argument with the larger value.

```
x1,x2  list of values from which the
       larger is to be returned.
```

MIN(x1,x2)

MIN returns, from a set of two arguments, the value of the argument with the smaller value.

```
x1,x2   list of values from which the smallest is to be returned.
```

MOD(x1,x2)

MOD returns the smallest non-negative value, R, such that:

(x1-R)/x2 = n where n is an integer.

R is the smallest non-negative value that must be subtracted from a given value x1 to make it exactly divisible by the given value x2.

If x1 is positive, R is the remainder of the division of x1 and x2; if x1 is negative, R is the modular equivalent of this remainder.

If x2 is zero, the ZERODIVIDE condition is raised.

MULTIPLY(x1,x2,p,q)

The MULTIPLY function returns the production of two arithmetic expressions, x1 and x2, with a specified precision p and a scale factor q.

```
x1  first value to be multiplied

x2  second value to be multiplied

p   An unsigned integer constant between 1 and 31.  This number represents
the total number of decimal digits used to represent the result.

q   An integer less than or equal to p.  This number represents the number
of fractional digits in the result.  (Note: If x1 or x2 is floating point,
q must be 0).
```

ROUND(x1,x2)

ROUND returns the given value x1 rounded at a digit specified by x2.

```
x1  the value to be rounded.

x2  decimal integer constant, specifying the
    digit at which rounding is to occur. If x2 is positive, it is
    the (x2)th digit to the right of the point; if negative, it
    is the (x2+1)st digit to the left of the point.
```

If x1 is floating-point, x2 is ignored; the rightmost bit of the mantissa is set to 1.

Note that the rounding of a negative value results in the rounding of its absolute value, then the sign is replaced.

SIGN(x) Arithmetic

SIGN returns a default-precision fixed-point binary integer that indicates whether a given value x is positive, zero, or negative. The value returned is as follows:

value of x	value returned
x > 0	+1
x = 0	0
x < 0	-1

SIN(x)

The SIN built-in function returns a floating-point value that is the sine of an arithmetic expression x, where x is an angle in radians.

TAN(x)

The TAN built-in function returns a floating-point value that is the tangent of an arithmetic expression x, where x is an angle in radians.

11.2.2. STRING-HANDLING FUNCTIONS

BIT(x)

Bit returns a bit string representation of a given value x.

x	expression to be converted.

CHAR(x)

CHAR returns a character string representation of a given value x.

x	expression to be converted.

COPY(x1,x2)

The COPY built-in function copies a given string x1 a specified number of times and concatenates the result into a single string.

x1	Any bit- or character-string expression. If the expression is a bit string, the result is a bit string. Otherwise, the result is a character string.
x2	Any expression that yields a nonnegative integer. The specified count controls the number of copies of the string that are concatenated, as follows:

Value of Count	String Returned
0	a null string
1	the string argument
n	concatenated copies of the string argument

Example

The function reference

```
    COPY('12',3)
```

returns the character-string value '121212'.

<u>HIGH(x)</u>

HIGH returns a character string of length x where each character is the highest character in the collating sequence (hexadecimal FF).

x   expression specifying the length.

<u>INDEX(x1,x2)</u>

INDEX returns a fixed-point binary integer indicating the starting position of a substring identical to string x2 within the string x1.

x1   string to be searched

x2   object of search

If x2 does not occur in x1, the value zero is returned.

<u>LENGTH(x)</u>

LENGTH returns a fixed-point binary integer specifying the current length of a given string x.

<u>SUBSTR(x1,x2[,x3])</u>

SUBSTR returns a substring of the given string x1.

x1   string from which the substring is to be extracted.

x2   an integer specifying the position of the first character of the substring in x1.

x3   an integer specifying the length of the substring to be extracted. If x3 is omitted, the substring returned is position x2 in x1 to the end of x2.

<u>TRANSLATE(x1,x2,x3)</u>

TRANSLATE returns a string the same length as a given string x1 where all or some of the characters may have been changed. Characters are changed according to a look-up table provided by strings x2 and x3.

The function operates on each character of x1 as follows:

If a character in x3 is found in x1, then the character in x2 that corresponds to the one in x3 is copied to the result; otherwise, the character in x1 remains.

x1   character string to be searched for possible translation of all or some of its characters.

x2   character string containing the translation values of characters.

x3   character string containing the characters that are to be translated.

Strings x2 and x3 should be the same length; otherwise x2 is padded with blanks, or truncated, on the right to match the length of x3.

`UNSPEC(x)`

UNSPEC returns a bit string that is the binary form of a given value x.

x expression of any data type

The length of the returned bit-string depends on the attributes of x.

The bit string is padded, if necessary, on the right with zeros to match the length of the variable.

`VERIFY(x1,x2)` `String-handling`

VERIFY returns a fixed-point binary integer indicating the position in the given string x1 of the first character or bit that is not in the given string x2. If all the characters or bits in x1 do appear in x2, a value of zero is returned.

x1	string to be scanned for any character not in x2.
x2	the verification string, consisting of a set of characters in any order.

11.2.3. SYSTEM FUNCTIONS

`DATE`

DATE returns a character string of length six, in the form yymmdd, where:

yy	the current year
mm	the current month
dd	the current day

`TIME`

TIME returns a character string of length nine, in the form hhmmssttt, where:

hh	the current hour
mm	number of minutes
ss	number of seconds
ttt	number of milliseconds

11.3. BUILT-IN MODEL FUNCTIONS

`AMAX(X1[,FOR__EACH.X2]*)`

Finds the maximum value of an array of elements. This function can be used across records. The optional FOR__EACH.X2 identifies the subscript on which the maximization is carried out. In the absence of any subscript the rightmost subscript of x1 is used.

`AMIN(X1[,FOR__EACH.X2]*)`

Finds the minimum value of an array of values. This function can be used across records. The optional FOR__EACH.X2 identifies the subscript on which the minimization is carried out. In the absence of any subscript the rightmost subscript of x1 is used.

`CHOOSE('string1,string2,...')`

The CHOOSE function will return a random selection from the choice of strings which it is given as arguments.

```
string1,2,etc  any string expression.  The maximal
               length of each section is 32.
```

For example, CHOOSE('MON,TUES,WED,THUR') will randomly return one of the first four days of the week.

DEPENDS_ON(R)

The argument R must be a RECORD structure. This function is used to indicate an external dependency (due to activities outside the specification) between records. It needs to be used when a target (output) record must precede a source (output) record. (This is the case for instance if a source (input) record. This is the case for instance if a source record is a response to a target record. Let SR be a vector of source records and TR be a target record vector, both subscripted by I.

```
SR(I)= DEPENDS_ON (TR(I-1))
```

means that an external receiver of TR(I-1) produces SR(I).

EXIST(BOOL1,FOR__EACH.x1)

The EXIST function will return a value or 1 or 0 based upon the presence of a condition in a certain dimension to be tested.

```
BOOL1          any boolean expression.  Will generally contain
               variables subscripted by 'FOR__EACH.x1'

FOR__EACH.x1   A global subscript. (does not have to use
               the 'FOR_EACH' prefix).
```

For example, the following specification tests whether any record in the source file has a numeric field greater than 0.

```
module:tryfun;
source:in;
target:out;
1 in is file,
  2 inr(*) is record,
    3 Num_fld is field (num);
1 out is file,
  2 our is record,
    3 existfld is field (num);
i is subscript;

existfld=exist(Num_fld(i)>0,i);
```

Note that the resultant variable has one less dimension than the condition.

FALSE

The FALSE function may be used to represent a value of '0'B. It may be coded in conditional assertions such as, for example:

```
COND(I)=IF A(I)=1 then '1'B ELSE FALSE;
```

This expression would be the same if the '0'B were actually coded.

RANDOMBIT(x1)

The RANDOMBIT function returns a string of random bits. x1 must be an integer less than 16. The function will return a random sequence of as many ones and zeros as are specified by x1.

RANDOMCHAR(x1)

The RANDOMCHAR function returns a string of random characters. x1 must be an integer less than 100. The function will return a random sequence of as many characters as are specified by x1.

RANDOMINDEX(x1)

The RANDOMBIT function returns a random integer from 1 to x1.

RUNSUM(X1[,FOR__EACH__X2]*)

This function is identical in syntax and execution to that of SUM. The difference between them is that SUM accumulates the sum of the elements over the complete range of the subscripts implied while RUNSUM yields at any point the cummulative sum so far. Consider the two following examples:

I. For an input file containing a single field A in each record it is required to return a single record containing a field B which is the sum of all the fields A in the input file:

```
        IN IS FILE(RECORD IS IN_R)(*));
         IN_R IS RECORD (A);
          A IS FIELD;
        OUT IS FILE (RECORD IS OUT_R);
         OUT_R IS RECORD (B);
          B IS FIELD;
        B = SUM (A);
The last assertion can also be written as:
        B = SUM(A,FOR_EACH.IN_R);
        B = SUM(A(FOR_EACH.IN_R)),
        B = SUM(A(FOR_EACH.IN_R), FOR_EACH.IN_R);
```

II. consider now the case that for the same input file we wish an output file with an output record for each input record. The fields in this record are C which is a copy of A, and D is the cumulative sum of all the A fields to this point.

```
        IN IS FILE (RECORD IS IN_R)(*));
         IN_R IS RECORD (A);
          A IS FIELD;
        OUT IS FILE (RECORD IS OUT_R)(*));
         OUT_R IS RECORD (C,D);
          C IS FIELD;
          D IS FIELD;
        C = A;
        D = RUNSUM(A)
The last assertion can also be written as:
        D = RUNSUM(A, FOR_EACH.IN_R)
        D (FOR_EACH.IN_R) = RUNSUM(A,FOR_EACH.IN_R)
etc.
```

SUBLINEAR(BOOL1,BOOL2)

The SUBLINEAR function may be used to have the system automatically genearate sublinear index equation in a specification.

PAR1	The conditional expression which will determine whether the sublinear variable will begin from a value of 1 or 0.

PAR 2	The conditional expression which will determine whether the sublinear variable will remain at current value or be incremented by 1.

If you write

```
name(exp1,exp2,...,ind)=SUBLINEAR(par1,par2)
```

the MODEL system sees

```
name(exp1,exp2,...,ind)= IF ind=1 THEN
                            IF par1 THEN 1 ELSE 0
                         ELSE IF par2 THEN name(exp1,exp2,...ind-1)+1
                                      ELSE name(exp1,exp2,...ind-1);
```

If you wish the sublinear to alway begin from 1, then the key-word, TRUE, may be used for par1.

SUM(X1[,FOR__EACH.X2]*)

X1 may be a variable or a subscripted variable. The X1 are summed. FOR__EACH.X2 is a subscript. In the absence of any FOR__EACH.X2 parameters the summation is performed on the rightmost subscript of X1. Note that in the presence of several subscripts as parameters a multiple level summation is performed.

TRUE

The TRUE function may be used to represent a value of '1'B. It may be coded in conditional assertions such as, for example:

```
COND(I)=IF A(I)=1 then TRUE else '0'B;
```

This expression would be the same if the '1'B were actually coded.

WHICH(BOOL1,m,I)

The EXIST function will return the number of the index value along a certain dimension at the time of the first occurrence of a condition.

BOOL1	any boolean expression. Will generally contain variables subscripted by 'FOR__EACH.x1'
m	starting value for the search. If specified, the test for the condition will begin when the global subscript has a value of m.
I	A global subscript.

For example, the following specification returns a value of the first record in the source file which has a numeric field greater than 0. It begins testing right at the start of the source file, (i.e. m=1).

```
module:tryfun;
source:in;
target:out;
1 in is file,
  2 inr(*) is record,
    3 Num_fld is field (num);
1 out is file,
  2 our is record,
    3 whichfld is field (num);
i is subscript;
```

```
whichfld=which(Num_fld(i)>0,1,i);
```

Note that the resultant variable has one less dimension than the condition.

11.4. FUNCTIONS DEFINED BY USERS

In addition to the above enumerated built-in functions, a user may define additional function in MODEL, have the respecitve programs generated automatically and placed in the MODEL compiler library. See section 4.10 for details of syntax and description of method.

Appendix I
ERROR/WARNING MESSAGES IN MODEL

Message Identifier	Message
L	APD1: MODEL COMPILER ERROR: DICTIONARY ENTRIES EXCEED LIMIT OF 20,000.
E	ASS1: INCONSISTENT USE OF LEFT-HAND-SIDE VARIABLE IN A CONDITIONAL ASSERTION AT LINE *line-number-within-statement* OF STATEMENT *statement-number*. THE ORIGINAL LEFT-HAND-SIDE IS "*name-1*" BUT "*name-2*" IS USED LATER.
F	ASS2: MULTIPLE ASSERTION ANALYSIS FAILURE. UNABLE TO BUILD THE PARSING TREE FOR ASSERTION IN STATEMENT: *statement-number*.
W	BKT1: IN STATEMENT *statement-number*, MAXIMUM ITERATION HAS BEEN CHANGED TO *maximum-iteration-number*.
W	BKT2: NO OPEN BLOCK IN STATEMENT *statement-number*. END STATEMENT IS IGNORED.
W	BKT3: END BLOCK LABEL "*name*" IN STATEMENT *statement-number* DOES NOT MATCH CURRENT BLOCK LABEL "*name*" DEFINED IN STATEMENT *block-statement-number*.
F	BTC1: MODEL COMPILER ERROR: GENIOCD (BYTE_CALC) - ILLEGAL TYPE IN CALL OF PROCEDURE "BYTE_CALC".
W	CDG1: *variable-name* IS INITIALIZED TO ONE IN BLOCK *block-name*.
E	CDG2: THE SUFFIX OF "SUBSET" IN "*name*" IS INVALID.
F	CDG3: MODEL COMPILER ERROR: NO SUBSCRIPT FOUND FOR VARIABLE "*name*" AT DIMENSION *position-number*.
F	CDG4: NO MODULE NAME FOUND FOR THIS SPECIFICATION.
L	CDG5: MODEL COMPILER LIMIT: IN GENERATION OF "FINFO" REPORT, INDEX OF "*name*" EXCEEDS 7 DIGITS.
L	CDG6: MODEL COMPILER LIMIT: IN GENERATION OF "FINFO" REPORT, REPETITION OF "*name*" EXCEEDS 5 DIGITS.
L	CDG7: MODEL COMPILER LIMIT: IN GENERATION OF "FINFO" REPORT, LEVEL OF "*name*" EXCEEDS 3 DIGITS.
L	CDG8: MODEL COMPILER LIMIT: ASSERTION TEXT LENGTH EXCEEDS 5000 CHARACTERS.

W CHK1: CONDITIONAL STATEMENT "*assertion-number*" HAS A NON-BOOLEAN EXPRESSION.

E CHK1: CONDITIONAL STATEMENT "*assertion-number*" HAS A NON-BOOLEAN EXPRESSION.

E CHK2: CONDITIONAL STATEMENT "*assertion-number*" HAS DIFFERENTLY TYPED RIGHT-HAND-SIDES TO BE EQUATED TO A SINGLE LEFT-HAND-SIDE.

E CHK3: INVALID OPERAND(S) FOUND FOR OPERATOR "*operator*" IN ASSERTION *assertion-number*. THEIR TYPES ARE: "*data-type*" AND "*data-type*".

W CHK4: A "*data-type*"-TYPED VARIABLE IS CONVERTED TO "FIXED BIN" IN ASSERTION *assertion-number*.

E CHK5: INCOMPATIBLE OPERAND(S) FOUND FOR OPERATOR "*operator*" IN ASSERTION *assertion-number*.

E CHK6: WRONG NUMBER OF ARGUMENTS FOUND FOR FUNCTION "*function-name*" IN ASSERTION *assertion-number*.

E CHK7: FUNCTION "*function-name*" IS INVOKED WITH AN INVALID DATA-TYPE ARGUMENT IN ASSERTION *assertion-number*. THE DECLARED TYPE IS "*data-type1*", THE INVOKED TYPE IS "*data-type2*".

W CHK8: VARIABLE "*name*" TYPE "*data-type*" IS CONVERTED TO "FIXED BIN" IN ASSERTION *assertion-number*.

W CHK9: IMPLICIT CONVERSION BETWEEN VARIABLES OF TYPE "*data-type*" AND "*data-type*" IN ASSERTION *assertion-number*.

D CHK10: MODEL COMPILER ERROR: INVALID TYPE ASSIGNMENT IN TYPE PROPAGATION.

W CHK12: CONFLICT IN ARITHMETIC, CHARACTER, AND/OR BIT STRING USAGE OF VARIABLE "*name*". FLOAT BIN(53) HAS BEEN SELECTED FOR ITS DATA-TYPE.

W CHK13: CONFLICT IN ARITHMETIC AND/OR BIT STRING USAGE OF VARIABLE "*name*". FLOAT BIN(53) HAS BEEN SELECTED FOR ITS DATA-TYPE.

E CHK14: UNARY OPERATOR "*operator*" IS USED WITH AN INVALID DATA-TYPE VARIABLE IN ASSERTION *assertion-number*.

W CHK15: CONFLICT IN ARITHMETIC AND/OR CHARACTER USAGE OF VARIABLE "*name*". FLOAT BIN(53) HAS BEEN SELECTED FOR ITS DATA-TYPE.

E CHK16: A NON-FIELD VARIABLE WAS USED IN ASSERTION *assertion-number*.

F CHK17: MODEL COMPILER ERROR: MALFUNCTION IN THE "CHECKER" PHASE.

W CHK18: USER FUNCTION "*function-name*" HAS NO PARAMETER CHECKS.
ARGUMENT & DATA TYPE CHECKING WILL BE BYPASSED.

W CHK19: PARAMETER CHECK STATEMENT MAY BE INVALID IN USER FUNCTION
"*function-name*".

W CHK20: INVALID PARAMETER CHECK STATEMENT FOUND IN USER FUNCTION
"*function-name*". CHECKING WILL BE BYPASSED.

W CHK21: PARAMETER CHECK STATEMENT INVALID FOR "RETURN" VALUE IN USER
FUNCTION "*function-name*". "RETURN" CHECKING WILL BE
BYPASSED.

F CLS1: MODEL COMPILER ERROR: MALFUNCTION IN MSCC ANALYSIS.

W CRD1: "*name*" IS CONTAINED IN MORE THAN ONE PARENTAL STRUCTURE.

W CRD2: THE FOLLOWING QUALIFIED NAME-*name* HAS AN UNDEFINED SUFFIX.

F CRD3: "*node-name*" IS MULTIPLY DECLARED WITHIN A DATA STRUCTURE.

E CRD4: ILLEGAL ISAM KEY DEFINITION "*name*". A KEY MUST BE IMMEDIATELY
UNDER A SINGLE "RECORD" STRUCTURE IN AN ISAM FILE.

F CRD5: MEMBER "*name*" IN FILE "*file-name*" IS UNDECLARED.

E CRD6: "*name*" SHOULD NOT BE DECLARED AS A GLOBAL SUBSCRIPT.

E CRD7: THE ISAM FILE "*file-name*" HAS AN UNMATCHED KEY. "*key-name*"
IS NOT FOUND WITHIN THE FILE STRUCTURE.

D CRD8: MODEL COMPILER ERROR: "*type*" IS AN ILLEGAL STATEMENT TYPE.

W CRD9: A DECLARATION HAS BEEN SUPPLIED FOR THE UNDECLARED VARIABLE
"*name*".

E CRE1: INVALID QUALIFIED NAME "*name*". WRONG FILE PREFIX SPECIFIED.

E DMP1: CONTRADICTION IN DIMENSIONALITY

W EED1: INVALID SUFFIX FOR "FOUND". "*node-name*" DOES NOT BELONG TO A

SOURCE FILE.

W EED2: INVALID SUFFIX OF "SUBSET" *node-name*. IT SHOULD BELONG TO A TARGET FILE.

E EED3: INVALID USE OF CONTROL VARIABLE "*node-name*". IT SHOULD CONTAIN A ".".

E EED4: "*node-name*" REFERS TO AN UNRECOGNIZED VARIABLE.

E EED5: INVALID CONTROL VARIABLE "*node-name*".

F EED6: ERROR(S) OCCURRED IN ENTERING EXPLICIT DEPENDENCIES. COMPILATION DISCONTINUED.

E EED8: INCOMPATIBLE LEFT-HAND-SIDES FOUND IN ASSERTION *assertion-number*.

E EED9: MODEL BUILTIN FUNCTION USED AT AN INTERNAL LEVEL IN ASSERTION *assertion-number*.

E EED10: VARIABLE NAME OR SUFFIX "*variable-name*" IS UNRECOGNIZED IN ASSERTION *assertion-number*.

E EED11: A FUNCTION "*function-name*" IS NOT RECOGNIZED IN ASSERTION *assertion-number*.

E EED12: "SIZE" OR "END" SUFFIX "*node-name*" IS NON-REPEATING.

E EED13: THE "POINTER" SUFFIX "*node-name*" IS A NON-KEYED RECORD.

E EED14: THE "NEXT" SUFFIX "*node-name*" IS NOT AN INPUT FIELD.

E EED15: THE "SUBSET" SUFFIX "*node-name*" IS NOT A VIRTUAL RECORD.

E EED16: THE "INITIAL" SUFFIX "*node-name*" IS NOT A FIELD.

W EED17: THE CONTROL VARIABLE "NEXT" REFERS TO FIELD "*target-name*" WHICH DOES NOT HAVE A CORRESPONDING FIELD IN EACH RECORD.

W EED17: THE CONTROL VARIABLE "NEXT" REFERS TO FIELD "*target-name*" WHICH DOES NOT HAVE A CORRESPONDING FIELD IN EACH RECORD.

W EED18: THE "MALDATA" SUFFIX "*node-name*" DOES NOT BELONG TO A SEQUENTIAL SOURCE FILE.

D EED19: NUMBER OF TARGET FILES EXCEEDS 100. PLEASE INFORM IMPLEMENTERS.

D EED20: NUMBER OF SOURCE FILES EXDEEDS 100. PLEASE INFORM IMPLEMENTERS.

D EED21: UNDEFINED FIELDS OVER 2000. PLEASE INFORM THE INPLEMENTERS.

W EHR1: RESERVED PREFIX "NEXT" APPEARS AS A DATA NAME: "*name*".

E EHR2: "ENDFILE" PREFIXES A NON-EXISTENT INPUT RECORD:"*record-name*".

E EHR3: A STARRED DIMENSION WITHIN A FIXED REPETITION DIMENSION OCCURRED IN VARIABLE "*name*".

F EHR4: MODEL COMPILER ERROR: "*name*" HAS A STATEMENT TYPE OF "*statement-type*", WHICH IS AN ILLEGAL TYPE.

E EHR5: NON-STANDARD FILE STRUCTURE. NO RECORD HAS BEEN FOUND BETWEEN "*name*" AND ITS FIELD(S).

E EHR6: NON-STANDARD FILE STRUCTURE. MORE THAN ONE RECORD HAS BEEN FOUND BETWEEN "*name*" AND ITS FIELD(S).

W EHR7: INAPPROPRIATE DATA STRUCTURE. MORE THAN ONE DIMENSION WITH UNKNOWN REPETITION BETWEEN "*name*" AND ITS FIELD(S). THIS MAY CAUSE AN EXCESSIVE AMOUNT OF MEMORY TO BE USED BY THE GENERATED PROGRAM.

E EHR8: THERE ARE NO FIELDS BELOW "*name*".

E EHR9: THE ISAM FILE "*name*" HAS AN INVALID STRUCTURE. ONLY ONE RECORD STRUCTURE IS ALLOWED.

E EHR10: THE ISAM FILE "*name*" HAS AN INVALID STRUCTURE. THE RECORD CANNOT BE REPEATING.

W EHR11: INVALID "MALDATA" SUFFIX "*name*". IT WILL BE IGNORED.

W EHR12: INVALID "MALDATA" SUFFIX "*name*". IT WILL BE IGNORED BECAUSE IT IS AN INTERIM VARIABLE.

W EHR13: GENERIC VARIABLE "*name*" WAS FOUND WITH AN UNDEFINED DATA-TYPE.

W EMD1: INCOMPLETENESS BETWEEN "*source-field-name*" AND "*target-field-name*".

F EMT1: MODEL COMPILER ERROR: "*predecessor-name*" IS NOT IN THE DICTIONARY.

F EMT2: MODEL COMPILER ERROR: "*successor-name*" IS NOT IN THE DICTIONARY.

E ENP1: THE "POINTER" SUFFIX "*name*" DOES NOT POINT TO A RECORD NAME.

E ENP2: THE "FOUND" SUFFIX "*name*" DOES NOT REFER TO A KEYED INPUT RECORD.

E ENP3: ARGUMENT "*name*" TO FUNCTION "EXIST" IS NOT A REPEATING GROUP OR RECORD.

W EVL1: THE RANGE FOR DIMENSION *range-number* OF "*name*" NEEDS AN UPPER BOUND AND HAS BEEN ASSUMED TO BE 9999.

E EVL2: MODEL PROCESSOR ERROR: THE *position-number* DIMENSION OF "*name*" SHOULD NOT USE WINDOW SCHEME.

E EVL3: NO RANGE DEFINITION FOUND FOR RANGE *range-number* IN THE RANGE TABLE.

E EVL4: MISSING RANGE DEFINITION FOR MAJOR RANGE *range-number* IN THE RANGE TABLE.

W EVL5: SUB-RANGE *sub-range number* OF RANGE *range-number* IS INVOLVED IN BOTH MAJOR-RANGE AND SUB-RANGE CALCULATIONS.

W FED1: END STATEMENT GENERATED FOR BLOCK "*name*" AT STATEMENT *statement-number*.

E FIR1: INCOMPLETENESS: NEED TO KNOW HOW TO OBTAIN "*target-field-name*".

W FIR2: ASSERTION GENERATED: "*target-field-name = source-field-name*;"

W FIR3: AMBIGUOUS SOURCE FIELD FOR TARGET FIELD "*target-field-name*". SOURCE FILES WITH THIS FIELD ARE: "*source-files*".

F FLM1: MODEL COMPILER ERROR: ERROR MESSAGE STACK UNDERFLOW. COMPILATION DISCONTINUED.

E FLM2: END-OF-FILE ENCOUNTERED WHILE LOOKING FOR SEMICOLON. LAST STATEMENT NUMBER ENCOUNTERED WAS *statement-number*

E FPK1: NO LENGTH DEFINITION FOR THE FIELD "*file-name*".

F FPK2: ISAM FILE INVOLVED INSIDE OF AN MSCC.

E FSB1: "*node-name*" IS DEFINED AS HAVING *dimension-number* DIMENSION(S), BUT DOES NOT HAVE THAT MANY REPEATING ANCESTORS.

E FSB2: "*node-name*" OF TYPE "*type*" SHOULD HAVE DIMENSION 0 BUT HAS BEEN ASSIGNED DIMENSION *dimension-number*.

E FSB3: EDGE OF ILLEGAL TYPE "*edge-type*" ENCOUNTERED BETWEEN "*source-name*" AND "*target-name*".

E FSB4: EDGE OF TYPE "*edge-type*" CONNECTS "*source-name*" OF DIMENSION *dimension-number* TO "*target-name*" OF DIMENSION *dimension-number*.

E FSB5: EDGE OF TYPE "*edge-type*" CONNECTS "*source-name*" TO "*target-name*", BUT "*target-name*" IS PART OF A PHYSICAL STRUCTURE

E FSB6: EDGE OF TYPE "*edge-type*" CONNECTS "*source-name*" TO NODE "*target-name*", BUT "*source-name*" IS PART OF A PHYSICAL STRUCTURE.

F FSB7: MODEL COMPILER ERROR. INVALID "ENDFILE" SUFFIX "*target-name*". THE LAST RECORD OF THE FILE SHOULD BE USED.

E GAS1: VARIABLE "*name*" IN THE ASSERTION "*node-name*" CONTAINS AN INAPPROPRIATE SUBSCRIPT EXPRESSION IN POSITION *position-number*.

E GDL1: "POINTER" SUFFIX "*name*" IS NOT AN ISAM RECORD NAME.

E GDL2: "POINTER" SUFFIX "*name*" HAS NO ISAM KEY VALUE.

E GIO1: NO POINTER DEFINITION FOR THE KEYED RECORD "*node-name*".

E GIO2: WRONG I/O CASE AT RECORD "*node-name*".

E GIO3: KEYNAME "*key-name*" IS NOT IN THE ISAM FILE "*file name*".

W GIO5: OUTPUT RECORD "*record name*" IS TOO LARGE. THE MAXIMUM IS 32767 BITS.

W GIO6: OUTPUT RECORD "*record name*" IS TOO LARGE. USER DEFINED SIZE *rec_size* WILL BE USED.

W GIO7: USER DEFINED "REC_SIZE" IS VALID ONLY WHEN MAXIMUM RECORD LENGTH EXCEEDS 32767 BITS.

F HSR1: MODEL COMPILER ERROR: "*target-field-name*" IS NOT IN DICTIONARY.

W IIX1: SOME SUBSCRIPTS APPEAR ON THE RIGHT-HAND-SIDE BUT NOT ON THE LEFT-HAND-SIDE OF AN ASSERTION. SELECTION IS IMPLIED FOR "*subscript-list*" IN ASSERTION *assertion-number*.

W IIX2: A GENERAL EXPRESSION APPEARS AS A LEFT-HAND-SIDE SUBSCRIPT IN ASSERTION *assertion-number*.

E IIX3: THE ASSERTION *assertion-number* IS NEITHER OF THE SIMPLE NOR OF THE IF ASSERTION TYPE.

F IIX4: MODEL COMPILER ERROR: INVALID TARGET FIELD IN ASSERTION *assertion-number*.

F IIX5: MODEL COMPILER ERROR: INVALID CONSTRUCT FOUND IN ASSERTION

assertion-number.

W IIX6: *Number found of* SAWTOOTH ARRAYS. THIS EXCEEDS THE LIMIT OF 20.

F INA1: MODEL COMPILER ERROR: DICTIONARY OVERFLOW.

F INI1: MODEL COMPILER ERROR: MAXIMUM NUMBER OF FUNCTIONS (*100*) EXCEEDED.

E ISF1: SOURCE FILE NAME "*name*" IS LONGER THAN 7 CHARACTERS.

L ISF2: MODEL COMPILER LIMIT: NUMBER OF SOURCE FILES GREATER THAN 20. PLEASE USE MULTIPLE "SOURCE" STATEMENTS.

E ITF1: TARGET FILE NAME "*name*" IS LONGER THAN 7 CHARACTERS.

L ITF2: IMPLEMENTATION LIMIT REACHED. NUMBER OF TARGET FILES GREATER THAN 20. PLEASE USE MULTIPLE TARGET STATEMENTS.

W LEX1: TEXT "*token...*" AT LINE *line-number-within-statement* OF STATEMENT *statement-number* HAS BEEN TRUNCATED TO *maximum-qualified-name-length* CHARACTERS.

F LEX2: MODEL COMPILER ERROR: END OF SPECIFICATION REACHED WITH AN OPEN COMMENT. LAST VALID STATMENT WAS NUMBER *statement-number.*

F LEX3: MODEL COMPILER ERROR: END OF SPECIFICATION REACHED WHILE LOOKING FOR SEMICOLON. THE LAST VALID STATEMENT WAS NUMBER *statement-number.*

F LEX4: MODEL COMPILER ERROR: WRONG MODEL SPECIFICATION FILE NAME.

F LEX5: MODEL COMPILER ERROR: END OF SPECIFICATION REACHED WHILE LOOKING FOR SINGLE QUOTE MARK. THE LAST VALID STATEMENT WAS NUMBER *statement-number.*

E MIN1: RECORD OR GROUP NAME "*name*" IS RESERVED. USED IN LINE *line-number-within-statement* OF STATEMENT *statement-number.*

E MIN2: THE FIRST LEVEL OF A HIERARCHIAL DECLARATION STATEMENT CANNOT REPEAT. PLEASE ADD AN ADDITIONAL LEVEL OF DECLARATION. USED IN LINE *line-number-within-statement* OF STATEMENT *statement-number.*

E MIN3: ISAM FILE "*name*" HAS NO KEY DEFINED.

E MIN4: ILLEGAL FILE NAME. "*name*" IS RESERVED.

E MIN6: INVALID DECLARATION STATEMENT. THE LAST LINE IS *line-number-within-statement* OF STATEMENT *statement-number*. NOTE THAT RECORDS, GROUPS, AND FIELDS AT THE SAME LEVEL SHOULD HAVE THE SAME LEVEL NUMBER.

W MIN7: FIELD "*field name*" HAS A (*) REPETITION WHICH MAY REQUIRE EXCESSIVE MEMORY AT RUNTIME. IN STATEMENT *statement-number*.

E MIN8: INVALID REPETITION SPECIFICATION OF (*minimum:maximum*), AT LINE *line-number-within-statement* OF STATEMENT *statement-number*.

L MIN9: MODEL COMPILER LIMIT: FIRST LEVEL MEMBERS OF A *hierarchial* STRUCTURE "*name*" EXCEEDS 100.
PLEASE SEPARATE BY ADDING A DUMMY NON-REPEATING GROUP STRUCTURE.

L MIN10: MODEL COMPILER LIMIT: FIRST LEVEL MEMBERS OF FILE "i[file name]" EXCEEDS 100. PLEASE SEPARATE BY ADDING AN ADDITIONAL FILE DECLARATION.

E MIN11: INVALID FIELD STATEMENT AT LINE *line-number-within-statement* OF STATEMENT *statement-number*. ITS DEPENDENT(S) ARE "*field names*".

E MIN12: INVALID STRUCTURE DECLARATION IN STATEMENT *statement-number*. A FILE MAY ONLY BE DECLARED AS THE TOP MOST LEVEL IN THE STRUCTURE.

F MNT1: NO CODE GENERATED DUE TO DATA TYPE ERROR(S).

F MNT2: MODEL SPECIFICATION FILE NOT FOUND.

W NRC1: "*token*" AT LINE *line-number-within-statement* OF STATEMENT *statement-number* IS TRUNCATED TO *maximum-name-length* CHARACTERS.

F PND1: MODEL COMPILER LIMIT: ASSERTION TEXT LENGTH EXCEEDS 5000 CHARACTERS.

F PRS1: MODEL COMPILER LIMIT: ASSERTION TEXT GREATER THAN 5000 CHARACTERS.

W RGP1: DIMENSION *dimension-number* OF "*name*" IN RANGE SET NUMBER *range-set-number* DOES NOT HAVE AN EXPLICIT RANGE.

W RGP2: DIMENSION *dimension number* OF "*name1*" AND DIMENSION *dimension number* OF "*name2*" HAVE INCOMPATIBLE RANGES, THUS THEY CANNOT SHARE THE SAME SUBSCRIPT.

E RGP3: INVALID SUBSCRIPT EXPRESSION IN A VIRTUAL DIMENSION. SUBSCRIPT IS "*name*" IN A DEPENDENCY OF "*target-name*" ON "*source-name*".

E RGP4: MULTIPLE RANGE CRITERIA FOR VARIABLE "*name*". BOTH "*name*" AND "*name*" HAVE BEEN SPECIFIED.

E RGP5: NO RANGE DEFINITION NEEDED FOR THE SUBLINEAR RANGE DESCRIBED BY "*name*".

F RGP6: MODEL COMPILER ERROR: "*name*" IS REFERENCED BUT NOT DEFINED.

W RGP7: "*Name1, dimension-number*" AND "*name2, dimension-number*" USED AS LEFT-HAND-SIDE VARIABLES, HAVE INCOMPATIBLE RANGES.

E RTB1: NO IMPLICIT RANGE HAS BEEN FOUND FOR RANGE NUMBER *range-set-number* IN THE RANGE TABLE. PLEASE CHECK RANGE DEFINITION(S).

W RTB2: DEFINITION FOR SUBRANGE *range-set-number* MAY BE DISCARDED IF THE STRUCTURE IS VIRTUAL.

W RTB3: NO DEFINITION FOUND FOR SUBRANGE *range-set-number*. NOT NEEDED IF THE STRUCTURE IS VIRTUAL.

L RTB4: MODEL COMPILER LIMIT: NUMBER OF RANGES EXCEEDS 160. THE RANGE TABLE WOULD BE PRINTED INCORRECTLY.

E RTB5: INVALID RANGE DEFINITION FOR SUBRANGE *range-set-number*. A CONSTANT WAS SPECIFIED IN "*name*" DEFINING THIS RANGE.

F RTV1: MODEL COMPILER ERROR: INVALID LOGICAL EXPRESSION "*symbol*".

L RTV2: MODEL COMPILER LIMIT: MAXIMUM RETRIEVALS EXCEEDED *array-bound*. LAST LOGICAL EXPRESSION WAS "*symbol*".

E SAP001: BIT STRING CONTAINS CHARACTER OTHER THAN 0 OR 1 AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP002: COLON MISSING AFTER THE WORD "BLOCK" AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP003: BADLY FORMED BOOLEAN EXPRESSION AFTER IF IN-STATEMENT AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP004: MISSING OR INVALID NUMERIC CONSTANT IN ITERATIVE COUNT SPEC AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP005: MISSING OR INVALID NUMERIC CONSTANT IN RELATIVE ERROR SPEC AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP007: ORGANIZATION TYPE MISSING OR ILLEGAL IN DISK STATEMENT AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP009: TYPE DISK MISSING OR ILLEGAL IN DISK STATEMENT AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP012: MISSING ELSE IN CONDITIONAL EXPRESSION AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP014: ASSERTION MISSING AFTER THE KEYWORD "THEN" AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP018: NO BOOLEAN EXPRESSION AFTER THE KEYWORD "IF" AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP022: NO EXPRESSION AFTER LEFT PARENTHESIS AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP023: KEYWORD "=" IS MISSING AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP024: RIGHT PARENTHESIS MISSING AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP026: STRING MISSING AFTER QUOTE AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP033: ERROR IN RECOGNITION OF RIGHT HAND SIDE OF AN ASSERTION AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP038: KEYWORD "THEN" IS MISSING AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP039: RECORD OR GROUP KEYWORD EXPECTED AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP042: RECORD NAME MISSING OR ILLEGAL IN FILE OR REPORT STATEMENT AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP044: MEDIUM NAME MISSING OR ILLEGAL IN FILE OR REPORT AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP045: KEYNAME MISSING IN FILE OR REPORT STATEMENT AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP046: MAXIMUM LENGTH MISSING OR ILLEGAL IN VARIABLE LENGTH IN FIELD STATEMENT AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP047: INVALID OR MISSING FIELD TYPE IN FIELD/INTERIM STATEMENT AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP048: MISSING OR INVALID LENGTH IN FIELD/INTERIM STATEMENT AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP049: MISSING RIGHT PARENTHESIS AFTER FIELD-TYPE IN FIELD/INTERIM AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP050: MINUS SIGN IS NOT FOLLOWED BY AN INTEGER AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP051: MISSING/INVALID MAX NUMBER OF OCCURRENCES OF ITEMS. AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP052: NAME MISSING OR ILLEGAL IN ITEM LIST AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP053: MISSING LEFT PARENTHESIS IN LINE SPEC AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP054: MISSING INTEGER IN LINE SPEC AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP055: MISSING RIGHT PARENTHESIS IN LINE SPEC AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP056: MISSING/INVALID FILE NAME AFTER KEYWORD FILE AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP057: FORMAT MISSING/MISSPELLED AFTER RECORD IN STORAGE STATEMENT AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP058: MISSING/INVALID TAPE LABEL AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP059: KEYWORD "RECORDSIZE" MISSING OR MISSPELLED AFTER "MAX" AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP060: MISSING/INVALID VOLUME NAME (EXTERNAL OR INTERNAL) AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP061: MISSING/INVALID DEVICE TYPE AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP062: MISSING/INVALID ITERATIVE SOLUTION METHOD AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP063: COLON MISSING AFTER KEYWORD "MODULE" AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP064: NAME MISSING OR ILLEGAL IN MODULE STATEMENT AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP065: ERROR IN ASSEMBLY OF A NUMBER CONSTANT AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP066: TAPE SPEC PARAMETER MISSING OR ILLEGAL AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP067: ERROR IN PICTURE SPEC AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP068: QUALIFIED NAME ILLEGAL AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP069: RECORD FORMAT MISSING OR ILLEGAL AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP070: KEYWORD "BLOCKSIZE" MISSING IN RECORD FORMAT SPEC AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP071: BLOCKSIZE VALUE MISSING/ILLEGAL IN RECORD FORMAT SPEC AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP072: RECORD SIZE VALUE MISSING/ILLEGAL IN RECORD FORMAT SPEC AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP074: SEMICOLON MISSING AT END OF STATEMENT AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP075: COLON MISSING AFTER KEYWORD "SOURCE" AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP076: NAME MISSING/ILLEGAL IN SOURCE FILE LIST AT LINE

line-number-within-statement OF STATEMENT
statement-number AT SYMBOL "*symbol*".

E SAP077: COLON MISSING AFTER KEYWORD "TARGET" AT LINE
line-number-within-statement OF STATEMENT
statement-number AT SYMBOL "*symbol*".

E SAP078: NAME MISSING/ILLEGAL IN TARGET FILE LIST AT LINE
line-number-within-statement OF STATEMENT
statement-number AT SYMBOL "*symbol*".

E SAP079: MISSING "THEN" IN CONDITIONAL EXPRESSION AT LINE
line-number-within-statement OF STATEMENT
statement-number AT SYMBOL "*symbol*".

E SAP080: UNRECOGNIZABLE STATEMENT AT LINE *line-number-within-statement*
OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP081: BADLY FORMED ARITHMETIC EXPRESSION AT LINE
line-number-within-statement OF STATEMENT
statement-number AT SYMBOL "*symbol*".

E SAP082: BADLY FORMED BOOLEAN EXPRESSION AT LINE
line-number-within-statement OF STATEMENT
statement-number AT SYMBOL "*symbol*".

E SAP083: BADLY FORMED BOOLEAN TERM AT LINE
line-number-within-statement OF STATEMENT
statement-number AT SYMBOL "*symbol*".

E SAP084: BADLY FORMED CONCATENATION OF EXPRESSIONS AT LINE
line-number-within-statement OF STATEMENT
statement-number AT SYMBOL "*symbol*".

E SAP085: BADLY FORMED FACTOR AT LINE *line-number-within-statement* OF
STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP086: BADLY FORMED PRIMARY AT LINE *line-number-within-statement* OF
STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP087: BADLY FORMED TERM AT LINE *line-number-within-statement* OF
STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP090: LEFT PARENTHESIS MISSING IN COLUMN SPEC AT LINE
line-number-within-statement OF STATEMENT
statement-number AT SYMBOL "*symbol*".

E SAP091: INTEGER MISSING IN COLUMN SPEC AT LINE
line-number-within-statement OF STATEMENT
statement-number AT SYMBOL "*symbol*".

E SAP092: RIGHT PARENTHESIS MISSING IN COLUMN SPEC AT LINE
line-number-within-statement OF STATEMENT
statement-number AT SYMBOL "*symbol*".

E SAP101: LENGTH OF PICTURE SPECIFICATION IS TOO SMALL AT LINE
line-number-within-statement OF STATEMENT
statement-number AT SYMBOL "*symbol*".

E SAP102: SPECIFIED LENGTH IS INAPPROPRIATE FOR SPECIFIED TYPE OF DATA AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP104: SPECIFIED MAXIMUM LENGTH IS INAPPROPRIATE OR TOO SMALL AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP105: FRACTION POINT OFFSET IS OUTSIDE OF BOUNDS -128<P<127 AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP106: BAD REPETITION SPECIFICATION AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP107: ILLEGAL CHARACTER IN PICTURE SPECIFICATION AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP108: EXPECTING A LEVEL NUMBER IN A STRUCTURED DATA DESCRIPTION STATEMENT AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP109: LENGTH OF PICTURE SPECIFICATION IS TOO BIG AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP110: ILLEGAL BIT STRING IN "ON_CERR" CLAUSE AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP111: INCONSISTENT USE OF "ON_CERR" CLAUSE AND THE ATTRIBUTE OF THE FIELD AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP112: INVALID SPECIFICATION IN "ON_CERR" CLAUSE AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP113: ILLEGAL B2 CONSTANT AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP114: ILLEGAL B3 CONSTANT AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP115: ILLEGAL B4 CONSTANT AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP120: MORE THEN ONE SOURCE FILE IN FUNCTION SPECIFICATION AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP121: MORE THEN ONE TARGET FILE IN FUNCTION SPECIFICATION AT LINE *line-number-within-statement* OF STATEMENT *statement-number* AT SYMBOL "*symbol*".

E SAP122: MORE THEN ONE RECORD IN FUNCTION FILE DEFINITION AT LINE *line-number-within-statement* OF STATEMENT

statement-number AT SYMBOL "*symbol*".

E SAP123: GROUPS ARE NOT ALLOWED IN FUNCTION FILE DEFINITIONS AT LINE
line-number-within-statement OF STATEMENT
statement-number AT SYMBOL "*symbol*".

W SCD1: THE FOLLOWING ASSERTIONS ARE CONSIDERED AS SIMULTANEOUS EQUATIONS:

W SCD2: ALL OLD RECORDS IN THE ISAM FILE "*file-name*" ARE REFERENCED IN
ONE OPERATION.

W SCD3: CYCLE CONTAINS THE FOLLOWING ELEMENTS:

E SCD4: NO RANGE DETERMINED FOR LOOP VARIABLES AT LEVEL
loop-level-number IN CYCLE:

E SCD5: NO RANGE DEFINITION FOUND FOR RANGE *range-number*.

E SCD6: MORE THAN ONE OCCURRENCE OF THE "DEPENDS_ON" FUNCTION HAS BEEN
FOUND IN THE SAME LEVEL OF AN MSCC:

F SCD7: MODEL COMPILER ERROR: (SIM_BLK) THIS MSCC DOES NOT FORM A SET OF
SIMULTANEOUS EQUATIONS.

E SCD8: (SIM_BLK) THIS MSCC DOES NOT FORM A SET OF SIMULTANEOUS EQUATIONS.

W SCD9: RANGE OF "*name*" NEEDS AN UPPER BOUND AND HAS BEEN ASSUMED TO BE
9999.

F SCD10: MODEL COMPILER ERROR: NO SUBRANGE CAN BE FOUND.

W SCD11: SUB-RANGE *sub-range number* OF RANGE *range number* IS
INVOLVED IN BOTH MAJOR-RANGE AND SUB-RANGE CALCULATIONS.

E SFD1: INVALID FIELD NAME AT LINE *statement-number*. "*name*" IS
RESERVED.

E SFD1: INVALID FIELD NAME AT LINE *statement-number*. "*name*" IS
RESERVED.

W SFL1: DEVICE=TAPE, FILE ORGANIZATION IS DEFINED AS ISAM, BUT HAS BEEN
CHANGED TO SAM AT LINE *line-number-within-statement* OF
STATEMENT *statement-number*.

W SFL2: DEVICE=DISK. TAPE ATTRIBUTE(S) IGNORED, AT LINE
line-number-within-statement OF STATEMENT
statement-number.

W SFL3: DEVICE=TAPE. DISK ATTRIBUTE(S) IGNORED, AT LINE
line-number-within-statement OF STATEMENT
statement-number.

E SFL4: UNIDENTIFIED FILE TYPE FOR FILE "*name*", AT LINE *line-number-within-statement* OF STATEMENT *statement-number*.

W SFL5: FILE NAME "*token*" IN DATA DESCRIPTION IS LONGER THAN 7 CHARACTERS. TRUNCATED TO "*name*".

W SPS1: NO SOURCE FILE DECLARED, FUNCTION IS WITHOUT PARAMETERS.

E SPS2: NO TARGET FILE DECLARED, FUNCTION WILL RETURN NO VALUE.

E SPS3: ONLY ONE RETURN PARAMETER ALLOWED, BUT MORE THAN ONE FIELD IS DECLARED IN THE TARGET FILE.

E SPS4: SUBSCRIPTED PARAMETERS IN FUNCTIONS ARE NOT ALLOWED.

E SPS5: SUBSCRIPTED RECORD OF PARAMETERS IN FUNCTIONS IS NOT ALLOWED.

E SPS6: MAXIMUM OF 20 PARAMETERS PER FUNCTION ALLOWED.

E SVF1: ISAM FILE "*name*" HAS NO KEY DEFINED.

E SVF2: INVALID FILE NAME AT LINE *line-number-within-statement* OF STATEMENT *statement-number*. "*name*" IS RESERVED.

W SVF3: FILE NAME "*name*" LONGER THAN 7 CHARACTERS.

F SVF4: IMPLEMENTATION LIMIT REACHED. FIRST LEVEL MEMBERS OF FILE "*name*" EXCEEDS 100. PLEASE SEPARATE BY ADDING AN ADDITIONAL FILE DECLARATION.

E SVT1: THE SUM, EXIST, AND WHICH FUNCTIONS CAN ONLY BE USED AS A SINGLE EXPRESSION ON THE RIGHT-HAND-SIDE OF AN ASSERTION. STATEMENT *statement-number* VIOLATES THIS RULE.

L SVT2: MODEL COMPILER LIMIT: NUMBER OF DESCENDENTS EXCEEDED 500.

L SVT3: MODEL COMPILER LIMIT: ASSERTION *statement-number* IS TOO LONG.

L SVT3: MODEL COMPILER LIMIT: ASSERTION *statement-number* IS TOO LONG.

L SVT4: MODEL COMPILER LIMIT: NUMBER OF MULTIPLE TARGETS EXCEEDS 26**2.

E WID1: WINDOWING ERROR FOR VIRTUAL DIMENSION *dimension-number* OF "*name*". WINDOW HAS A WIDTH OF *width*.

W WIO1: MODEL COMPILER LIMIT: ONLY 1000 WINDOW VARIABLES CAN BE ANALYZED.STACK OPTIMIZATION DISABLED.

F WPL1: MODEL COMPILER ERROR: BAD PARAMETER FOR ROUTINE WRT.

E XRF1: INCONSISTENCY. CONTRADICTORY DESCRIPTIONS OF "*key-name*".

L XRF2: MODEL COMPILER LIMIT: ITEM "*key-name*" IS DEFINED IN MORE THAN
12 FILES.

Index