A MODELING SYSTEM FOR MATHEMATICAL PROGRAMMING

Wilhelm F. Burger

May, 1981 TR 177

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

ABSTRACT

Mathematical programming has become an important tool for planning and decision making. Current technologies make the process of model construction and model solution slow and tedious. This thesis does not concern itself with optimization algorithms, though the models considered are restricted to LP models. Rather it focuses on the model formulation process and the data management aspects of the model data in order to improve the situation and to bridge the gap between end-user and machine. Mathematical programming software has evolved in a pragmatic, informal way, quite independent of related development in programming language design and data base design. Drawing on these areas a modeling language and data base facility is described which combined with man-machine interaction forms the basis of a modeling system.

In order to develop and solve models of significant size and complexity it is necessary to approach the problem in a modular fashion. A modeling task is broken down into sub-tasks that can be handled separately. In particular model formulation and model data management should be treated independently. The modeling language supports this concept by providing formulation modules and data modules as its basic building blocks.

The model equations are formulated in algebraic notation. This is considered to be a more natural representation of mathematical programming problems than for example the notation currently required for the matrix generators of commercial LP systems. Models are formulated as generalized models consisting of a set of equations which may cover serveral aspects of a problem. A particular model is obtained by selecting the relevant equations in a problem definition. This mechanism adds flexibility to the model formulation process.

Model formulations and model data are kept in a data base. The data handling facilities of the modeling system are based on the relational data model. A relation constructor is provided for expressing data access in a non-procedural way. This notation is uniformly applicable; it appears in index constructs of equations, and it is used for simple data base queries. Data are organized in modules. Model solutions are obtained by executing a problem formulation with a data module.

An interactive environment is needed to support the design process of a model, and to allow the user to experiment with alternative problem solutions. A command language is defined with which different modes of man-machine interaction can be initiated. Of importance is the browsing mode in which data definition and data manipulation facilities are available interactively. The browsing mode can be used to change or create data for different solutions, or it may be simply used to learn

more about the data at hand. These facilities provide the necessary support to handle 'what-if' questions in a decision making process.

The modeling language is the result of an integration of concepts developed in programming language design, data base design, and man machine interaction, applied to the area of Operations Research. The feasibility of a modeling system is demonstrated by discussing several implementation issues concerning the data base, the translator, and the interpreter of a modeling system.

Acknowledgements

I would like to express my gratitude to the many people who provided me over the years with inspiration, assistance, support, and understanding. In particular, I want to thank my advisor, Prof. Mani Chandy, for his guidance and encouragement. He will always be an example to me. I express my appreciation to Profs. J.C. Browne, A.G. Dale, U. Dayal, and D. Good for serving on my dissertation committee. My thanks are extended to Dr. R. Yeh for his support.

I am indebted and grateful to my wife, Pramilla: her confidence, love, and patience made this thesis possible. Special thanks are due to members of the faculty and my fellow students for their discussions and help, in particular: A. Araya, J. Bitner, R. Cohen, C. Dawson, H. Kunii, and M. Tyson. I would also like to thank Mrs. E. Arnakis for her hospitality during my stay in Austin.

Finally, I would like to thank A. Meeraus at the World Bank who introduced me to the problem area, and Prof. S. Seegmuller, Technical University, Munich, and Dr. D. von Lindern, Max-Planck Institute, Munich, who made it possible for me to continue with my work while I stayed in Germany.

Last, but not least, I thank my parents for their constant love, support, and encouragement.

This work was, in part, supported by the Air Force, Office of Scientific Research (AFOSR 77-3409) and by the National Science Foundation (MCS 79-25383).

CONTENTS

CHAPTER 1

Introduction

1.1 Goals of the Thesis

Mathematical modeling plays an increasing role in handling real-world problems. Mathematical modeling has become a tool for planning and decision making. This places new demands on how modeling is being carried out. Computers have, however, not yet played the important role in decision making as forecast by [Zannetos] due to the high resource requirements and the inability of decision makers to access the tools without skilled specialists.

This research is concerned with the design and implementation of a modeling system which bridges the gap between end-user and machine. The models to be considered are restricted to Linear Programming (LP) models. They constitute a very important section of mathematical modeling. The extensions to other mathematical programming models is straightforward. Moreover the solution algorithms for LP's have been so highly developed that we don't have to be concerned with them. Instead we shall concentrate on the model formulation process and the data management aspects of the model data.

In order to develop and solve models of significant size and complexity the following design goals emerged. The system should

- support a model notation which is easy to use and which can be easily communicated to other model builders.
- provide for incremental change to support the evolutionary design process.
- allow the user to browse through the data to identify relevant data and discover new relationships.
- support the ability to obtain alternate solutions and ask 'what-if' questions.
- provide a convenient, user-friendly interface. This includes guidance in the formulation process and information about the state of the system.

The system must be necessarily interactive to support the incremental and browsing facilities. Initial ideas on a modeling system were presented in [Burger1].

LP software so far has evolved in a pragmatic, informal way, quite independent of related development in programming languages and data base design. This study draws on the areas of programming languages and data base design for developing languages for model formulation, data definition, and for providing data base capabilities to support a modeling system. For example the module concept in programming languages is applied to structure models into managable parts, and the type concept is used to better classify and structure data. An attempt was made to integrate in a clean manner several languages which address the various aspects of modeling: a language to formulate models, a language to define and manipulate data, a language to present the results, and a command language to interact with the user and to tie the different parts of the system together.

The project was undertaken to identify and solve problems in the data structures and data management areas which specifically relate to modeling environment, language definition and man-machine interaction. Parts of a prototype modeling system have been implemented to demonstrate and test the ideas. A goal for the future is to develop the system into a production tool to handle complex models which can support timely decision making. Finally it is hoped that the modeling system can be used as an educational tool for teaching modeling based upon mathematical programming.

1.2 Organization of the Thesis

The remainder of Chapter 1 reviews the LP modeling problem. It looks at the currently available LP software, and the need for a new approach to support the modern demands on modeling. Chapter 2 gives an overview of the modeling system. The type concept, modules, and the relational data model are introduced. Language aspects for data manipulation and equation formulation are developed. Chapter 3 defines the language constructs for data declaration and data manipulation. Chapter 4 presents the language for model formulations. Formulation module definitions are introduced. Chapter 5 defines the command language. User interaction facilities, browsing facilities and data modules are described. Chapter 6 addresses implementation issues of the modeling system. The design and implementation of the data base, the translator, the interpreter, and the user interface are described. Chapter 7 concludes the thesis with a summary of the work presented and some suggestions for future research.

1.3 Linear Programming

Linear programming (LP) systems obtain optimal solutions to models composed of linear inequalities by finding the smallest (or largest) value for a linear function (the objective function) of the model's variables, such that the linear inequalities are all satisfied. The two main aspects of LP modeling we are concerned with are

- the model formulation process, design, and evolutionary development

- the data management problems: how the data underlying the model are assembled, transformed, and fed to the optimizer (LP solution algorithm) and how the results (optimal solution values) are retrieved, analyzed, summarized, and reported.

Model formulation and data management are handled together in traditional LP systems as the model formulation can be directly expressed in the way the data are supplied to the LP solution algorithm. The process of providing data for an LP system is generally referred to as 'matrix generation'. Matrix generation is only part of the overall LP data management which includes the management of solutions and variations of a model. LP data management has considerable practical importance. [Welch] reports that in current practice LP models easily reach a size of 2000 rows by 8000 columns. Assuming that the matrix is about 0.2% dense about 32000 data elements must be managed. This number increases drastically when variations of a model need to be considered and various solutions must be kept.

Given a particular LP optimizer two different approaches are taken to LP solution data management. The data are supplied to the LP algorithm either by employing a general purpose language, or by using a special purpose language specifically designed to support LP modeling. These two methods are treated in the next sections. We consider these methods as two different levels of sophistication in the development of modeling systems.

1.4 General Purpose Language

A general purpose language (e.g. FORTRAN) is used to transform data for a particular model into the desired form used by an LP optimizer. Another transformation step written in a general purpose language is usually applied to report the solution data. [Welch] points out that this approach is taken in organizations where 'programming' can be delegated to some data processing department.

The effects of this approach are as follows:

- The time span between model conception and solution is very large. Several people are involved in 'programming' the model. The final program representation of the model may differ from the intended model due to failures in communication between the people, and due to the time lag between conception and realization of the model.

- Changes to the model are difficult to incorporate as this may require major revisions to the program. Alternate formulations or further investigations are not carried out due to the high cost involved in making modifications.

- The final form of the model is the program itself. The model cannot be communicated to another model builder without substantial effort. Also the development process of the model is lost as no history can be kept.

A model consists of a series of steps which, if not integrated into a system leave ample room for human error. Computing time may be saved if a model is run on a routine basis. However, when models become older, they are also more likely to need revision. Thus when inherent limits of the program are reached, e.g. array dimensions, then major reprogramming is necessary.

The cost of building models this way is quite high, and the productivity of the model builder is very low. [Fromm] reports that the cost of building a mathematical model, averaged over 650 models (linear and other) surveyed, cost a staggering $154,000. Further it took an average of 17 month to make a model operational.

1.5 Specialized Languages

In an attempt to overcome the drawbacks of the previous method the various steps of the modeling process are integrated into a system and specialized languages are provided to handle the input and output of model data. Commercial LP systems such as OMNI [Haverly], GAMMA [Burroughs], and DATAFORM [Ketron] fall into this category. The following features are more or less common to these systems.

Data is provided in tabular form. Tables are one of the preferred forms of representing data. The tables, columns, and rows are named. These names and their manipulations are the basis for the LP matrix generation. The column and row names of the LP matrix are constructed from the column and row names of the data tables. The matrix generation statements describe how the elements from the data tables are to be placed into the LP matrix. The names (original or constructed) are then later used in the report generation phase.

The LP formulation is to a large extent data independent as the generation statements rely only on the table, column and row names. Characteristic of these languages is the ability to manipulate names and set of names, e.g. create subsets, form unions, etc. The model generation can be 'debugged' by first using small data tables. Later the presumably much larger actual data tables are used for the solution.

Having provided a systematic way to interface with the LP solution algorithm these systems feature also some capabilities to manage the data and the matrix and report generation statements. The data can be kept and modified in their original form. This is in general supported by a simple data file system. Similarly a model may be assembled from groups of matrix and report generation statements which are kept on a file. These features add to the flexibility of an LP system. Most importantly

it is now possible to understand and make modifications to the model and data by users other than the original model builder.

[Welch] points out that specialized languages are popular with LP analysts who have the implementation as well as the design responsibilities for their LP studies. As these languages are oriented to the problem of matrix generation requiring no 'programming' more attention can be paid to the model itself. Still, however, the LP analyst must translate the model into the form required by the LP system in use. Why not formulate the problem directly in a notation with which the model builder is familiar? The next section introduces some systems which employ 'model formulation' languages.

1.6 Model Formulation Languages

An early attempt was made by [Aigner] to formulate models to the computer in a notation similar to the one the model builder is familiar with. The equations which represent the model are used in a simplified form as input routine to the LP system.

Another approach was taken by the AMBUSH system [White]. Here the model formulation is customized to a particular problem area, namely process flow and transportation. The model is described in the jargon of this problem area.

A modern approach is taken by the GAMS system [Meeraus1] which is currently under development. Algebraic notation is employed for the model equations so that a large variety of optimization problems can be formulated. Algebraic notation is considered to be the 'natural' representation of mathematical programming problems. Algebraic notation is also used in LPMODEL [Katz]. Very simple equations can be expressed with the notation provided by this system.

The concept of a model formulation language brings with it the separation of model formulation and model data. It opens up the possibility to develop each aspect separately. Moreover the model formulation becomes independent of a particular solution system. This is the direction which leads to the development of a modeling system.

1.7 Modeling System Requirements

General aspects of computer based planning and modeling are discussed in [Wolters]. Problems are classified from well-structured to ill-structured. Well structured problems can be handled by formalized methods and it is possible to fully automate the decision making process. Less structured problems require the active participation of the user in a man-machine dialogue. Different alternatives based on the intuiton and experience of the user are compared with the help of the computer. An effective modeling system must be able to cover a wide range of problems.

Three closely inter-related aspects of a modeling system need to be addressed: model formulation, data management, and man-machine interaction. Man-machine interaction is an essential part of model formulation as well as data management.

The model formulation must be in a notation which can be easily mastered by a non-programmer so that the computer can become an efficient tool for the model builder. The model formulation is directly accepted by the computer. Changes to the model can be carried out easily. This is important during model development as the problem at hand may not be well understood, and the model formulation itself may serve as a learning device to improve the conceptual understanding of the problem. The model formulation must be embedded into a system which allows the incremental development of a model.

Changes to a model are also required when it is used to test different scenarios in order to answer 'what-if' questions. Here the modeling system must support a dialogue with the user so that answers can be obtained in a timely manner in a decision making process. Modifiability of a model is also necessary during the life time of the model.

The computer processable representation of a model brings several advantages with it. Without a computer a human has to carry out the transformation of the model equations into structures accepted by some solution system. As now the computer does all the work this source of errors is eliminated. Moreover the machine can be used to check the underlying semantics of the formulation and thus errors can be caught early in the model formulation.

The data management aspects of modeling must be given new attention. First the volume of data to be handled has risen drastically. Second it must be possible to select and group data under various aspects, for example for answering 'what-if' questions, or simply to learn more about data. This can be achieved by giving the user some means of 'browsing' through data. A facility to transform available data into the form needed by a model is especially important. Third the data management must be concerned with managing 'internal' data, e.g. the input/output data of a solution algorithm, or a sequence of solution data to produce a final report.

The data management requirements are met by a data base system which provides a high level of data and processing abstraction. The data base is the central part of a modeling system (besides the solution algorithms). Not only will it contain model data, but also the formulations so that incremental model development can be supported. A modular structure of model formulations and data will add to the flexibility of the modeling system.

The man-machine interaction facilities of the modeling system must be very user-friendly if the system is to be also employed successfully by a casual user. This can be achieved by providing 'help' features. Help may

be actively requested by the user, or the system may generate suggestions to guide the user.

Current technology in the LP modeling area as described in previous sections falls short of the desired goals. In order to fulfil the modern demands on modeling it is necessary to provide an integrated system. The next section looks at some systems which attempted to combine specialized applications with data base functions in a user-friendly environment.

1.8 Related Research

The systems described here address one or more of these areas: model formulation, data management and man-machine interaction. All these systems recognized the need to develop a very high level description language. Some systems combine a data base system with a specialized application package.

GAMS was developed at the World Bank to provide a tool for modeling in a strategic planning environment [Meeraus2]. In its current state of implementation it is used to formulate models. Algebraic notation is used to define model equations. Special emphasis is placed on the notion of sets. Sets are values which are represented by symbolic names. Sets are used e.g. as indexes. The system was designed, however, for a batch environment, and thus lacks incremental facilities. Data management by a data base system was not considered.

LABSTAT [Labstat] is an example of a data-oriented system. The system handles data for time-series problems. Data are stored in the commercial data base management system TOTAL. Several languages are provided to access data: a language to make simple queries, a language to prepare data for an external system, and a language to format data for reports. The languages are problem-oriented and non-procedural.

FINSIM [Klein] is an interactive decision support system for financial planning and engineering. It is used here as example for man-machine interaction. Its overall design is very attractive. A data base is kept for models and data. (There are four financial planning models which can be parameterized. Extensions to a model require modifications to the standard model). Great emphasis is placed on the help features of the system. The help features serve as learning aid and provide assistance in using the system.

LPMODEL [Katz] is a recent APL based system for constructing simple LP models. It employs algebraic notation for expressing equations. The underlying APL system supports interactive facilities and the storing of model formulations. A data base subsystem is provided to associate values with the constants in equations of a model when the LP matrix is generated.

There are many more systems which touch, in one way or another, on the problems of a modeling system. [Alter] surveyed a variety of

data-oriented and model-oriented decision systems. An attempt to integrate DBMS functions and specialized applications into one user-friendly environment is also made by the ELLIPSE system [Suvorov]. ELLIPSE is developed at the Central Economics Mathematical Institute, USSR, to provide a support system for economic-oriented research.

CHAPTER 2

The System

In this chapter we give an overview of the system and look into some of the components making up the system. We are only concerned here with the overall architecture and logical aspects of the system. Details of the languages and implementation considerations are handled in later chapters.

2.1 Systems Overview

Figure 2-1 illustrates the high level design of the system. This is also the user view of the overall system. The main parts of the system are the user interface, task components, the data base and the solution systems.

The modeling system is an integrated system combining model formulation and model solution into one system. The user interface provides a command language with which a particular mode of interaction with the modeling system can be initiated. The following tasks can be carried out:

- definition of models and data structures
- modifications of formulations
- execution of formulations
- browsing through data
- input and output of data
- interrogation of the system

Each mode is described briefly. In the definition mode the user makes various formulations known to the system. Formulations are for example, model equations, data structures for model data, or formulations for executing a model. The various formulations are packaged into modules. They can be defined independently. Formulations may be obtained from a file or entered from a terminal.

In modification mode formulations can be altered. The system keeps track of modules which are affected by a modification. If necessary the user is prompted to make changes to these modules. In execution mode the user solves a model. Here data from the data base are formatted according to the model equations and sent to the solution system. The results are stored in the data base. Optionally solution reports may be prepared.

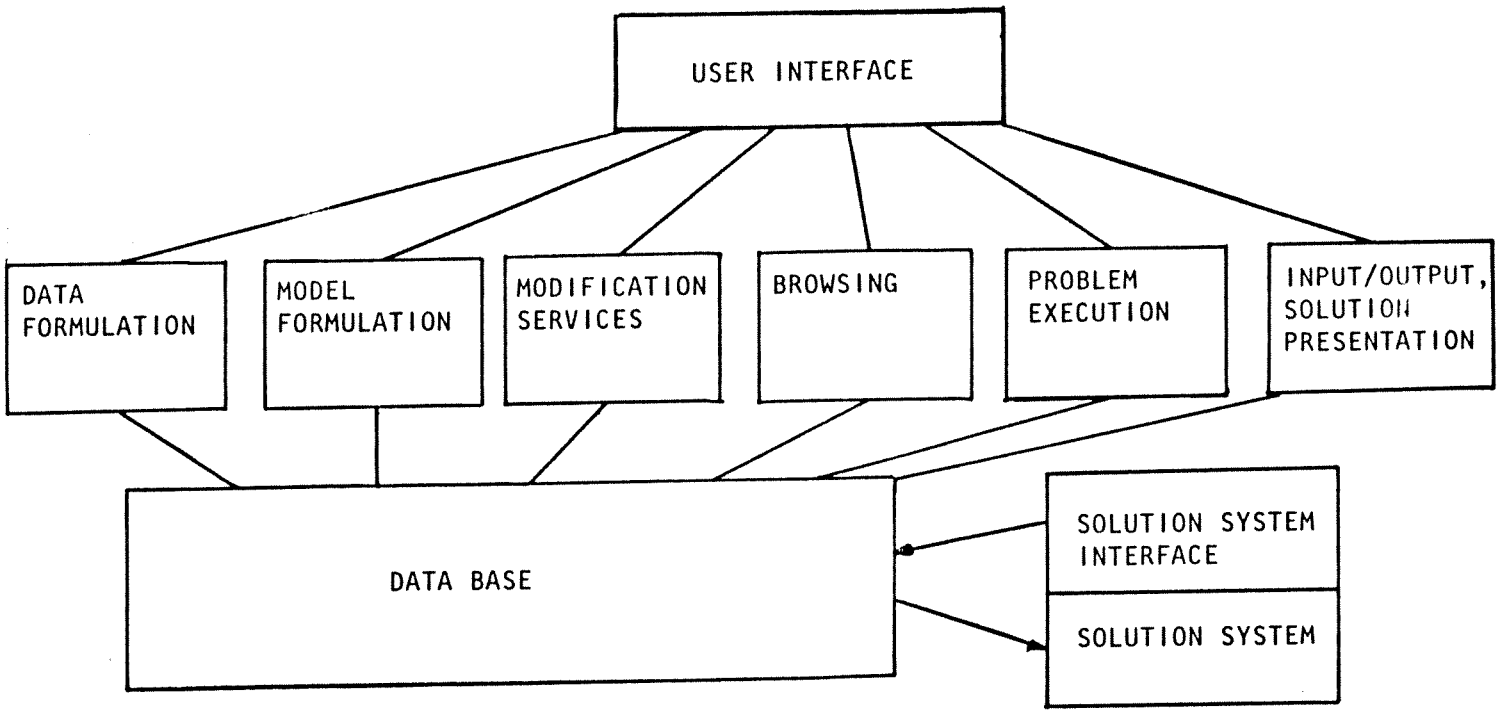

Figure 2-1 High Level Structure of Modeling System

In browsing mode the user can interactively create, change, or investigate data. Input/output mode is concerned with getting data into or out of the data base. This may involve external files or the generation of reports. Input/output is in general under the control of one of the other tasks. Finally, at any time the user may interrogate the system to obtain pertinent information. For example the user can list directories of modules and data, or ask for help explaining a command.

The structuring of formulations into modules is a basic concept of the modeling system. The ability to combine modules gives the user great flexibility in handling complex problem. Modules are also the basic unit of interaction with the system. Depending on the kind of module the interaction may be simply modifications to a module, or as in the case of browsing, it may involve data manipulations.

The data base part of the modeling system combines the data management for several components of the system. The data base consists of three parts: a data base for external data, a data base for modules, and a data base for internal data. Model data is considered to be external data. The structure of the data is also stored in the data base. Model formulations, model solutions, and report formulations are stored as modules in the data base. Internal data is generated by a solution execution, either as input to the solution system or as output from the solution system.

The solution system consists of a standard optimization algorithm and an interface for adapting the data to and from the modeling system. The solution system must not necessarily be part of the modeling system or even be on the same machine. In an environment where several machines are combined in a network it is feasible to delegate the optimization to a very fast machine (e.g. a CDC Cyber) while the rest can be done on a smaller machine (e.g. a DEC-10).

Several languages are provided by the modeling system to formulate the various parts of a modeling problem. Languages are available to formulate data structures, models, reports, model solutions, and data base queries. An attempt was made to design these languages in a uniform way. The design of the languages for the modeling system is strongly influenced by the concept of modules and by the need for interactivity.

The support for user interaction is an important aspect of the modeling system. During model development the user can interactively modify or create modules. In browsing mode or input/output mode the user can interactively manipulate data. The system combines in a balanced way batch requirements (model formulation/solution) with interactive requirements (data manipulation/modifications).

Finally the system was designed to support a casual user as well. In particular, the user can get help from the system explaining where he is and what he can do in the state he is in.

2.2 Modules

One of the basic assumptions is that data and models can be defined separately. The same data may be used with different models, or a model may be solved with several data sets. Also parts of one model may be useful in other models. To handle these situations we introduce modules. Modules are a powerful tool for structuring complex problems.

Modules come in two varieties: modules for formulations, and modules for data. The first kind is static in nature while the second kind has dynamic aspects. Formulations, though they may be interactively modified, have well defined data structures and statements when they are ready for execution. Data modules on the other hand can be modified resulting in new data structures. The interfacing of static and dynamic modules brings with it additional problems to the problem of interfacing modules and the separate translation of modules.

Formulation modules can be handled in a similar way as modules in programming languages like ADA [ADA] and Modula-2 [Wirth2]. Modules contain interface declarations which are used to create and share a common environment. Separate translation of modules imposes a chronological sequence of translation. For example environment information must be translated first before a module using it can be translated. In the modeling system we have the benefit of a data base which can store the environment information and the modules.

Formulation modules are stored in the data base in two forms: an external form and an internal form. The external form of a module is a block of text. The internal form is obtained by translating the module. The translation process establishes the syntactic and semantic correctness of a module. The internal form of a module consists of a symbol table in case of declarations, and a sequence of instructions in case of statements. The external form of the module is used for modifications, for example for incremental development. Changes to a module affect all modules which depend on it. The system (under the control of the user) re-translates these modules (which in turn may require further modifications). The internal form of a module will be consistent with its external form when the modification mode of the modeling system is terminated.

Data modules can be created by declarations initially. This is done by a data creation module which has the form of a formulation module. A data module consists of two parts: data structures and data. Only the data structure descriptions are established when a data module is created. Data modules exist only in internal form. Data modules can be also created interactively by the user supplying declarations, or they can be modified by creating new data structures in browsing mode. An external form of a data module is therefore not feasible. The data creation module though an external form is merely used to get a data module with that particular structure established.

The solution of a problem requires the coordination of several modules. A typical case involves four kinds of modules: a data module, a model module, and a report module. The operational aspects of a problem solution are formulated in an execution module. While the formulation modules (in this case the model module and the report module) can be interfaced by sharing environment information, it is necessary to interface the data module by a formal parameter mechanism. This way all interfaces can be checked at translation time. When a data module is bound to an execution module at run-time a check is made that the actual data structures of the data module conform with the formal data structures of the execution module. The concept of an execution module makes it attractive to use the modeling system in an applied environment: Model and execution modules can be prepared by an 'expert', while the data preparation, model execution and the analysis of the results can be delegated to an 'assistant'.

The following modules can be defined by the user:

- declaration modules
- model modules
- data creation modules
- execution modules
- routine modules
- data modules

All modules with the exception of data modules are formulation modules. Routine modules are defined to handle data transformations and input/output. A report for example is formulated by a routine module. The modules are described in more detail after the relevant language constructs have been introduced.

2.3 Data Base

The data base of the modeling system is used to store formulation modules, data modules, and internal data. Directories are maintained for the various kinds of modules. The data base is also used for temporary data which for example may arise in browsing mode.

The storage of formulation modules poses no particular problems. Relationships between the modules are recorded so that the effects of modification to a module can be followed. The initial form always corresponds to the external form of a module. If necessary a module is translated to obtain the corresponding internal form. The internal form of declaration modules is used when modules are translated separately.

More sophistication is needed to store data modules. Data are a projection of the real world and their description should reflect the objects and relations between objects in a natural and understandable way. We must choose therefore a basic data model which is most suited for our purpose.

Basic data models are the hierarchical model [Date], network model [DBTG], and relational model [Codd1]. The hierarchical model may be considered a special case of the network model. The relational model represents data as a collection of tables while the network model stores the data and relationships between the entities explicitly. The data formulation in a network model has the advantage that relations between entities can be dynamically created.

The relational model has an appealing conceptual simplicity. Tables are the natural representation of data in the LP modeling area. Table statements to provide data for an LP solution algorithm are present in all commercial available LP systems. One of the basic activities of model data management is the creation and manipulation of tables. Matrix generation in the conventional sense can be seen as creating a new table (the input to the LP solution algorithms) from a number of small tables according to 'queries' represented by the model equations. The basic data model we choose therefore for the management of data is the relational model.

Table structures are defined by giving column names and the type of of the values which may be contained in a column. In the relational model columns of a table are also referred to as attributes, and the value set from which the values are taken as domain. Rows of tables are also referred to as tuples. In the modeling system this concept of table is slightly extended. Several columns can be grouped together to form a so-called array column which is referenced by one attribute name only. An index mechanism is provided to select the individual columns. An attribute of this kind will be referred to as array attribute. This extension allows us to deal with tables of similar structure in a uniform way; in particular it takes care of the problems of multiple right-hand sides in an LP model.

It is important to note a fundamental difference between our data base and a 'standard' relational system. The data structure information ('data dictionary') is not supplied to other modules when they are translated, but it is provided at run-time as part of the parameter mechanism. This is a consequence of the dynamic nature of the data base. While in conventional systems a 'data base administrator' defines the data structures ('schemas') which then are supplied as static structures ('sub-schemas') to user applications here the user has the ability to dynamically create new structures. This ability is necessary for transforming or extracting data while experimenting with a model. 'Administrative' and applicative interaction with the data base are, however, carried out in different modes of the modeling system. Another difference is that the data base of the modeling system is not designed for concurrent access. It is assumed that only one person at a time will use the system.

2.4 Type System

In order to better classify and structure data we use the type concept from programming languages. A type is an abstraction of a class of similar values. A type can be described by the properties shared by all values of its corresponding class. Besides the conceptualization of data values, data types are used to maintain certain properties when data are manipulated. To a large extent this can be already done at translation time through type checking. The success of the programming language Pascal [Wirth1] is in part due to the added security in formulation and better abstraction facility by a type system.

We apply the type system to model formulation and to data module definition. One of the major problems with data bases is getting control over the validity of data which enter the data base. The typing mechanism can be used to define consistency constraints. Early relational languages (ALPHA [Codd2], SEQUEL [Chamberlin]) did not have such a mechanism to support the integrity of the data base. Type systems for data base integrity have been suggested by [Prenner], [Schmidt1], [Brodie] and [Turnherr]. BETA [Brodie] is a relational language based on the type system, and PASCAL/R [Schmidt2, Meyer] is an extension of Pascal with relational facilities.

The type concept from programming languages can be directly carried over to formulation modules. Types are introduced in variable declarations or they are explicitly assigned a name in type declarations. The use of a named type could simply be considered an abbreviated notation, however, type checking may take the type name also into account. In weak type checking two objects are type compatible if their underlying types are the same. In strong type checking the type name must be the same too. Types for strong type checking are called strict types by [Schmidt2] and interpreted types by [Brodie]. We introduce strict types by domain definitions. Strict types add to the integrity of formulation.

The type concept is also applied to data modules; however, we must pay special attention to data independence and type checking. Data (and their structures) exist independently from the way they were created. We therefore have two universes of types: the types of data in data modules and the types of data in applications (e.g. model modules). When data of a data module are bound to an application then the relevant types of the application are considered to be formal parameters for which the actual types of the data module are substituted. This is supported by so-called external defined types. External defined types are incomplete type specifications for which details are provided when a particular data module is interfaced with an application. Therefore, an application can be written independently of the data. The types contain enough information to do the type checking of static properties at translation time. The dynamic aspects of data are only relevant when the data are bound to the application. Run-time checks need only be provided for those features which cannot be covered at translation time. External defined types are particularly useful for dealing with array attributes of tables.

In this case they provide information on the size of the array at run-time.

The use of typing information for models and data may seem to increase the complexity of the languages contrary to our goal of simplicity and ease of use of the modeling system. Types, however, contribute greatly to the clarity and correctness of the formulations. Furthermore the type system can be used as design aid in structuring and classifying data of the real world.

2.5 Data Manipulation

The data manipulation facilities of the modeling system are based on the relational model. The user can retrieve data from the data base by queries. New data may be computed from retrieved data. Additional facilities are provided to add, modify or delete data. Data manipulations can be found in routine or execution modules, or they are done interactively in browsing mode. The facilities handled here do not include input/output.

It was our goal to provide a data manipulation language which is simple and convenient to use. The most important features are the table manipulation capabilities in a form which a model builder is familiar with, like categorizing a table into sub-tables, or selecting rows and columns of a table according to some condition. The complexity of formulation increases with the complexity of the data manipulation requirement; thus for most needs simple formulations are sufficient.

The basic construct of the data manipulation language is the relation constructor. A new table (relation) is formed by extracting information from other tables and possibly transforming the information into new data. Relation constructors in relational languages are either descriptive (non-procedural) or prescriptive (procedural). The descriptive form of the relation constructor specifies the result and the tables participating in it but not how the result is obtained. Descriptive constructors are suited for simple queries [Huits]. Our approach for handling more complex queries is by combining several simple queries in a sequence. This approach is also taken in TAMALAN [Vandijk] and Regis [Joycel]. The modeling system also provides, however, language constructs which permit the formulation of queries in a procedural form.

The notation chosen for a relation constructor in the modeling system is based on the form of an operational set constructor in algebra:

{ <result table> | <input table> : <condition> }

(Quantifiers are not used in the formulation of conditions). The relation constructor can be seen as a loop construct: for each tuple from the input table the condition is checked, and if it holds the tuple is further processed and the result is included in the result table. A higher level loop construct is provided by the ability to partition a table according

to one or more attributes. This results in sub-tables (with the same value in one or more columns of the sub-table). The conditions are then applied to the sub-tables. The descriptive notation of a loop is also applicable in a similar fashion to the formulation of model equations and report formulations, thus contributing to the uniformity of the language.

In the following we describe the relation constructor in more detail. Examples are given in Section 3.3.1.4. Queries come in three levels of complexity: simple queries, queries which involve more than one table, and-queries which compute new values. We assume that most queries are simple, that is they are projections or selections. Queries involving two tables can be expressed by a join operation. The descriptive relation constructor is restricted to these kinds of queries. More complex queries either must be decomposed by the user into a sequence of simple queries, or the procedural query facilities of the language must be used.

Queries which require the computations of new values from a selected tuple could be handled by extending the relation constructor with a compute clause:

{ <result table> | <input table> : <condition> <computation> }

Though this extension fits naturally into the frame work of our relation constructor it was not included in the language. Computations fit the procedural nature of query processing better; furthermore the results can be already achieved by the tuple-at-a-time facilities of the language.

Simple queries are selections and projections of a table. The conditions are expressed by logical expressions. Logical expressions consist of boolean combinations (with operators 'and', 'or', 'not'), comparison terms (with operators =, <=, >=, <, >, <>) and set comparison terms (with operators 'in', 'contains', 'partof').

The next level of queries realizes the join operator of relational algebra. Two tables are combined to form the join of the two tables according to the join condition. Only those rows are retained for further processing for which the condition holds. As the result table may contain two columns with the same name a facility for renaming column names of the result table is provided.

The relation constructor is tied in with the type system. When possible, the attribute names and types of the result table are derived from the input table. The type of the result of a relation constructor must be compatible e.g. with the type of the variable it is assigned to. In browsing mode a new variable may be created by the user with a relation constructor. The type of the new variable is inherited from the relation constructor.

There are two more ways to create new tables. The first uses the set-theoretic operations 'union', 'intersection', and 'difference'. The two tables involved must be type compatible. The other way is by building up a table a tuple at a time by insert operations. The descriptive loop

construct of the relation constructor is also provided as statement and may be used for this purpose.

The descriptive relation constructor (and the construction by set-theoretic operations) always produces a new table which is temporary unless it is assigned to a variable which is connected to the data base. Special statements are provided which work directly on a table (temporary or in the data base). These statements let the user insert, delete, or update tuples in a table. The descriptive loop construct can be used for the selection of tuples by conditions; the selected tuples can then be modified or deleted. The update and insert operations maintain the uniqueness property of tuples in a table if the table is defined with that property.

The data manipulation facilities of the modeling system provide the user with powerful analytical capabilities. The ability to select items from one table based on conditions of another table, and the grouping of items in a table combined with computations are considered to be among the most important. The data manipulation language is complete in the sense of [Codd1] as every operation of the relational algebra can be expressed in this language.

2.6 Model Formulation

Before introducing the model formulations in the modeling system we give a brief review of the formulations found in LP modeling. We use a small transportation problem. Plants with certain capacities ship goods to markets with certain requirements. The objective is to select shipments in such a way as to minimize the transportation costs from plants to markets. Figure 2-2 defines the objects we are talking about.

i	plant i in the set of plants I
j	market j in the set of markets J
k_i	capacity of plant i
r_j	requirements of market j
x_{ij}	amount of goods shipped from i to j
c_{ij}	cost of a unit shipped from i to j

Figure 2-2 Objects of Transportation Model

Note that names fall into two categories: names for indexes and names for data objects. This distinction will be important later when we talk about the 'execution' of model equations.

The transportation model is described by the equations in Figure 2-3. Equation (1) states that the goods shipped to the markets from one plant cannot exceed the capacity of that plant. Equation (2) states that the goods received at a market must at least fulfill the requirements of that market. Equation (3) defines bounds on the amount of goods shipped. Finally equation (4) states the objective of the model: to minimize the cost of transportation. Implicit in these equations is that the amount of goods shipped (x_{ij}) is a variable (in the modeling sense) while capacity requirements and unit costs are constants (parameters in the modeling sense). Characteristic of LP programming is that the equations are linear forms in their variables.

$$(1) \qquad \sum_{j \varepsilon J} x_{ij} \leq k_i \qquad i \varepsilon I$$

$$(2) \qquad \sum_{i \varepsilon I} x_{ij} \geq r_j \qquad j \varepsilon J$$

$$(3) \qquad x_{ij} \geq 0 \qquad i \varepsilon I, j \varepsilon J$$

(4) Minimize

$$\sum_{i \varepsilon I} \sum_{j \varepsilon J} c_{ij} x_{ij}$$

Figure 2-3 Equations of Transportation Model

The above equations are in their simplest form. They can be made more complex by adding so-called row ranges, and by introducing bounded variables. We revise the transportation model with some additional conditions to demonstrate the more general case. First we impose a minimum shipment for each plant. This results in a row range (equation (1')). Then we put a limit on the amount of goods which can be shipped from a plant to a market, e.g. due to a limitation in transportation facilities. This results in a bounded variable (equation (3')). In its most general form bounded variables specify a lower and an upper limit. The modified equations are shown in Figure 2-4.

A further restriction can be placed on model variables. While they are usually continuous, they may be restricted to discrete values. With this restriction we enter the realm of mixed integer programming. Commercial LP systems may also provide solution algorithms for this problem category.

The formulations of models in the modeling system follow closely the mathematical notation of LP problems. A model is realized by a model module. The model module consists of three parts: a declaration part, an equation part, and a problem section. In the following we describe these parts further; details of the language constructs, and a formulation of

m_i minimum shipment from plant i

l_{ij} limit in shipment from i to j

$$(1') \quad m_i \leq \sum_{j \varepsilon J} x_{ij} \leq k_i \qquad i \varepsilon I$$

$$(3') \quad 0 \leq x_{ij} \leq l_{ij} \qquad i \varepsilon I, j \varepsilon J$$

Figure 2-4 Revised Transportation Problem

the transportation problem in the language of the modeling system are found in Section 4.1.

The declaration part corresponds to the definition of names which usually precede the model equations. Additional variables and routines for computations can be declared. Some of the variables are given the role of formal parameters of the model module when they appear in the interface specification of a problem in the problem section. The interface specification is used to connect the data structures of the model module to the data structures of the environment when the model module is executed.

The equation part defines the equations of the model. Equations are defined by declarations similar to routine declarations: they consist of a name, possibly formal parameters, and an expression. Equation declarations have the advantage that different aspects of the phenomena to be modelled can be all formulated in one place.

Equations are not executable in the sense of data manipulations; they are a higher level parameterization of the data transformation process of model data to model solution and vice-versa. The instantiation of the equations is controlled by indexes. The distinction between indexes and data is important: Indexes are the active part of an equation; they define which data elements are accessed. Data are the passive part: the equations only give the rules on how data elements are being selected and formatted for the solution algorithm. Index sets can be used to tailor a model to specific needs, e.g. data dependence of equations can be expressed by the values supplied for the index sets with which the equations are formulated. The formal parameters of an equation declaration are intended for this purpose.

Equations have the same form as their mathematical counter part. The left side of an equation is a linear form, usually an expression. The summation sign is represented by 'sigma' followed by the controlling indexes. The equation operator is one of the following: 'leq', 'geq', 'eq', and 'range'. The right side of an equation is a simple expression which must evaluate to a constant. In case of a range equation first the minimum and then the maximum of the range are given. Bounded variables

are treated in the same way as equations. If no bounds are declared for model variables then bounds are derived from the ranges of their basic type. A model variable is usually of type real. Discrete model variables, like zero-one variables, are defined with type integer. The objective function of a model is also formulated in the equation part similar to an equation. The equation operator is 'objective', but there is no right side. Several objective functions can be defined but only one may be selected in a given problem.

Array attributes may be used in equations (without indexes). They are useful for handling multiple righthand sides for LP modeling. In order to employ them successfully the index types of those array attributes which will create the righthand side must be all the same. This implies that the model variables which will contain the results must also use array attributes with the same index type.

The problem section of the model module consists of one or more problem formulations. The first part of a problem formulation defines the actual problem to be solved. The second part defines what to do with the model solution. This part consists of data manipulations. The problem formulation contains an interface specification. The interface specification defines the data input and data output of the problem. It lists the variables which are to be used as formal parameters of the problem.

The actual model to be solved is obtained by calls to the relevant equations defined in the equation section of the model module. Model variables, constraints, bounds, and the objective of the optimization are specified. The model description is embedded into the solve statement. When a problem is executed the calls to the equations generate the matrix which is then sent to the solution system. The solve statement defines in addition implicit variables which contain information on the solution. For an LP system implicit variables are provided for the status of the solution (e.g. feasibility), cost of the objective, and additional information on rows and columns like slack activity or input cost of the model variables. The variables which contain this information are declared implicitly because their structures are beyond the capability of the type system, and because the information returned also depends on the underlying solution system. However, in general the implicit variables are treated like table structures with relaxed access requirements: e.g. to get the slack activity of an equation the equation name (and index) can be used to access it. (A typical example of a slack activity would be the unused capacity of a production unit). If multiple righthand sides were present then these structures can be further indexed by values of the index type of the array attribute with which the righthand side was created. The solve statement may be preceded by a computation part which is intended for index computations.

The model formulations of the modeling system are derived from the mathematical notation used for models. Its descriptive character makes the formulation easy to use; it is also attractive for non-programmers. We simplified the formulation of equations by separating the index

computations with which data dependencies can be handled. The introduction of equation declarations makes it possible to formulate generalized models. While the formulations have been restricted to linear forms to deal with the LP programming area it is obvious that based on the mathematical notation these formulations can be easily expanded to cover other areas of mathematical programming as well. In particular the solution system interface formulations were made very flexible by employing implicit variables. This facility can be used to expand the modeling system with other solution systems without having to provide language elements to deal with particular features of those systems.

CHAPTER 3

Data Declarations and Data Manipulations

In this chapter we handle the language constructs for data declarations and data manipulations. The constructs are defined by grammar rules. A complete grammar is given in Appendix A.

3.1 Syntax and Representation

3.1.1 Notation for Syntax

The syntax is described in an extended Backus-Naur formalism which is also used in the definition of Modula-2. Terminal symbols of the language are either words written in capital letters or strings enclosed in quote marks. Example:

```
MODEL   "+"
```

The nonterminal symbols are words formed mostly by lower case characters. The words are chosen so that some meaning can be conveyed. Example:

```
RelationConstructor
```

A syntactic rule has the form

```
S = E
```

where S is a nonterminal and E a syntax expression. E may be a sequence of terminal or nonterminal symbols. Several alternatives are separated by the | character. Example:

```
number = integer | real
```

Square brackets are used to denote 0 or 1 occurrences of the enclosed syntactic expression, curly brackets denote 0 or more occurrences. Round brackets may be used to group alternatives together. Example:

```
letters =  letter {letter}
SignedNumber = ["+"|"-"] (integer | real)
```

3.1.2 Representation

Certain terminal symbols of the language are formed by lexical rules. We refer to them as lexemes. Lexemes in the modeling language are identifiers, numbers, strings, and one form of table constant. They are denoted in the grammar by the symbols ident, number, string, and TableConst respectively. The usual notation for identifiers, numbers and strings may be given by:

```
ident = letter {letter | digit}
integer = digit {digit}
real = digit {digit} "." {digit} [ScaleFactor]
ScaleFactor = "E" ["+"|"="] digit {digit}
number = integer | real
string = """" {character} """"
```

A quote symbol which is part of a string must be written twice.

The representation of identifiers is extended in the modeling language to allow also character sequences of the form integer, reserved word, and string to be used as identifiers. This is accomplished by applying an @ sign operator to the character sequence. Example:

```
@1980    @type    @"x-part"
```

The rules for table constants are handled in 3.3.1.3.

It is assumed that the ASCII character set is available for the representation of formulations. A smaller character set, however, is sufficient: upper case characters can be used instead of lower case characters, and special symbols can be replaced by the symbols on the right below:

```
"{"          "("
"}"          ")"
"|"          "%"
```

Blanks, end-of-line, and comments are ignored except as they serve to separate tokens of the language, e.g. to separate otherwise adjacent identifiers. (Blanks and end-of-line are significant in table constants, however). Comments start with the characters --. A comment is terminated by the end of the line. Example:

```
-- This is a comment
```

In the following for the convenience of the reader all grammar rules of the language are marked by a $ sign on the left margin.

3.2 Data Declarations

The data structures of formulation modules and data modules are defined by type and variable declarations. In the context of a data base system this part of the language is referred to as data definition language. Some differences exist between declarations in formulation modules and in data modules. They will be dealt with when modules are introduced.

3.2.1 Types

A data type determines a set of values which variables of that type may assume. A type also determines the operations which may be performed with values of that type. The type declaration associates an identifier with the type. We distinguish between simple types and structured types.

```
$    TypeDeclaration = SimpleTypedecl | StrictTypedecl |
$                      StructTypedecl
$    SimpleTypedecl = TYPE ident "=" basicType
$    StrictTypedecl = DOMAIN ident "=" strictType
$    StructTypedecl = TYPE ident "=" structType
```

Simple Types

Simple types are the types 'integer', 'real', 'boolean', 'string', and subrange type.

```
$    basicType = standardType ["[" range "]"]
$    standardType = INTEGER | REAL | BOOLEAN | STRING
$    range = rangeConst ".." rangeConst | POSITIVE | NEGATIVE
$    rangeConst = ["+"|"-"] (number | INF)
```

Subranges of a type are defined by giving the smallest and the largest value of the subrange. Example:

```
type AGE = integer[1..100]
type posint = integer[positive]
```

The range 'positive' is defined as 0..+inf, the range 'negative' is defined as -inf..0.

Strict Types

Strict types have the form of simple types. The type name is a property of the type. It is also used for checking type compatibility. Enumerated types are defined as strict types.

```
$    strictType = (basicType | enumeration | externaldefined)
$                 [constraint]
$    enumeration = "(" ident {"," ident} ")"
```

```
$    externaldefined = ENUMERATED | SUBRANGE
$    constraint = ORDERED | UNORDERED | RESTRICTED
```

The enumerated type is of particular importance: it specifies a set of values by introducing identifiers which stand for each value in the set. Example:

```
type COLOR = (RED,GREEN,BLUE)
```

Enumerated types are usually not ordered. An ordering can be imposed on an enumerated type by:

```
type color = (RED,GREEN,BLUE) ordered
```

The order is defined by the sequence in which the values are written down. The same identifier may be used as value descriptor in different enumerated types. The denoted values are different, however.

Constraints can be placed on strict types to control the applicability of comparison operators or arithmetic operators on values of that type. The comparison operators <, <=, >, >= can be applied to types whose values are ordered. They cannot be applied to types which are unordered. Example:

```
domain employernumber = integer[positive] unordered
```

The arithmetic operations can be taken away by the constraint 'restricted'. For example dates can be compared but not added to each other.

```
domain date = integer[601010..991231] restricted
```

Constraints were introduced to give the user some means for avoiding meaningless operations on the data.

Structured Types

Structured types are aggregations of simple types [Smith]. Structured types are tuples and relations. We refer to relations also as tables.

```
$    structType = tupleType | tableType | setType
```

Tuples

An entity in the 'real word' is described by a tuple definition. It consists of an aggregation of attribute names together with their types.

```
$    tupledef = "[" attributeSequence "]"
$    tupleType = TUPLE tupledef
$    attributeSequence = attribute {";" attribute}
```

```
$    attribute = ident [extension] |
$                ident {"," ident} [extension] ":" simpleType
$    simpleType = ident | basicType
$    extension = "<" ident ">"
```

A person for example may be described by three attributes: name, age, and weight. Assuming 'name' is declared as string type one can define the following type:

```
type persontype = tuple [name   : name;
                         age    : integer[1..100];
                         weight : integer[1..300] ]
```

The types of the attributes age and weight are anonymous types, type equivalency is done without considering the type name. In order to take advantage of the additional security offered by strict types one can define the following:

```
domain name   = string;
domain age    = integer[1..100];
domain weight = integer[1..300];

type persontype = tuple [name   : name;
                         age    : age;
                         weight : weight ];
```

In order not to burden the user too much with types, abbreviated notations can be used if this is allowed by the context. This makes it possible to approximate formulations in languages which do not use a type system.

```
type persontype = tuple [ name; age; weight ]
```

If two attributes take their values from the same domain then attribute names must be defined, e.g.

```
type persontype = tuple [ firstname : name;
                          lastname  : name;
                          age;
                          weight ]
```

It is possible to use one attribute name for a group of things of the same type, e.g. we may want to record the weight of a person for several years without having to invent several different attribute names. An attribute in this case is referred to as an array attribute. The attribute is extended by the name of a subrange type or enumerated type. Values of this type are used to identify a particular element of the group. Example:

```
type year = (@1960,@1970,@1980)
type persontype = tuple [ firstname: name;
                          lastname : name;
```

```
                                    age;
                                    weight<year> ]
```

Tables

Collections of entities of the same type are defined by a relation type. The relation type is a powerset structure.

```
$     tableType = TABLE columndef [WITH (keydef | DUPLICATES)]
$     columndef = ident | tupledef
$     keydef = KEY ident {"," ident}
```

An instance of a relation type is dynamic: the number of instances of the underlying type is not known. Instances are created and removed during the life time of the relation. An additional mechanism must be provided to identify the instances of a relation. This is accomplished by keys. A key consists of one or more attribute names of the underlying tuple type. Only one key per relation may be defined (even if other attribute combinations qualify as key). Names of array attributes cannot be used in the definition of a key. Example:

```
type persontable1 = table [ name; age; weight ]
                          with key name,age
type persontable2 = table persontype
                          with key firstname,lastname
```

A key definition assumes that all entities of a relation are uniquely identifiable and that this property is maintained by data manipulations. An entity thus consists of two parts: a key part which identifies the entity and a value part. A relation may be defined without key. It is then assumed that all attributes (except any array attributes) participate in the definition of a key. No duplicate entries may be contained in the relation.

If duplicates are allowed in a relation then obviously no key property can be defined. A declaration looks then as follows:

```
type persontable = table persontype with duplicates
```

Sets

One-column tables defined with a simple attribute can be considered to be set structures. For ease of notation we also introduce set structure declarations.

```
$     setType = SET simpleType
```

Example:

```
type colortype = set color
```

External Defined Types

The value set of an enumerated type does not have to be explicitly defined but may be bound to a set of values in the data base when e.g. a data module is interfaced with a problem module for execution. (Problem modules and data modules are introduced in the next chapters). The type name then can be declared as follows:

```
domain color = enumerated
```

Comparison operators can be made available to this type by specifying 'ordered'. The type name is sufficient to do the necessary type checking at translation time.

Similarly the fixing of bounds of integer subranges can be deferred until data from the data base are actually made available. Example:

```
domain partno = subrange
```

This facility is used when tables with array attributes of unknown dimension must be handled in formulation modules.

Implicit Types

Shorthand notations for tuple or relation definitions take advantage of the possibility to define types implicitly. For example

```
type production = tuple [plant; product]
```

adds automatically the two definitions:

```
domain plant   = enumerated
domain product = enumerated
```

This assumes, of course, that these identifiers were not declared previously.

3.2.2 Variables

Variable declarations serve to introduce variables and associate them with a unique identifier and a fixed data type and structure. Variables are used in formulation modules and data modules. The data type determines the values which the variable may assume and the operators which are applicable. The type of a variable is given by a type name or a type definition.

```
$    VariableDeclaration = VAR ident {"," ident} ":" typedef
$    typedef = ident | basicType | structType | derivedType
```

Example:

```
        var person: persontype;
        var persontable: table [name; age; weight]
                         with key name;
```

Implicit declarations of enumerated type names are possible.

For convenience of notation the declaration of a variable may be also done by derived types. The type of the variable may be defined by referring to a previously declared variable (or type) without repeating type information.

```
$    derivedType = TYPE ident ["." ident] | TUPLE ident
```

Example:

```
        var newpersons: type persontable
```

Variables may be defined having the same type as an attribute of a table or tuple structure. Example:

```
        var prsname: type persontable.name
```

If an array attribute is used then the variable is declared with the type of the columns.

Variables which are to contain tuples of a table can be declared in a similar way. Example:

```
        var nperson: tuple persontable
```

The type or variable used must be a table structure.

3.3 Data Manipulation

Data manipulations are defined in routine modules and problem modules. In the context of a data base this part of the language is referred to as data manipulation language. The constructs can be also used in browsing mode. Data manipulations are expressed by statements and expressions. Of particular interest are the relation constructor and the insert, update and delete operations.

3.3.1 Expressions

Expressions obtain or compute new values. Expressions consist of operands and operators. Operands may be constants, variables, designators, function calls, and relation constructors. The operations can be categorized into arithmetic, logical, comparison, and set operations. The applicability of an operator depends on the type of its operands.

```
$    primary = designator | number | sconst | tconst |
$              StructConst | FunctionCall | RelationConstructor
$    factor = primary | "(" expression ")" | N-operator factor
$    term = factor {M-operator factor}
$    simpleExpr = ["+"|"-"] term {A-operator term}
$    expression = simpleExpr [R-operator simpleExpr]
$    N-operator = NOT | TAKE
$    M-operator = "*" | "/" | DIV | MOD | AND
$    A-operator = "+" | "-" | OR | UNION | INTERSECT | MINUS
$    R-operator = "=" | "<>" | "<" | ">" | "<=" | ">=" | IN |
$                 CONTAINS | PARTOF
```

The precedence of the operators is given in decreasing order by:

N-operator, M-operator, A-operator, R-operator

3.3.1.1 Primaries

Simple primaries are designators and simple constants. A designator is an identifier which refers to a variable. It may be followed by an access list if the designated object is to be accessed by a key, and it may be followed by a field selector to access part of a structure, e.g. the field of a tuple, or the column of a table.

```
$    designator = ident [accessList] [fieldSelector] [property]
$    accessList = "[" expression {"," expression} "]"
$    fieldSelector = "." ident ["[" expression "]"]
$    property = "'" ident ["[" expression "]"]
```

Tuples of a table can be accessed by key if the table is defined with the key property. A key uniquely identifies the tuple. Parts of a tuple can be further selected. Given the declaration

```
var x: table [city; sz,pop:integer] with key city;
var y: tuple x;
```

a tuple of table 'x' is accessed by key as follows:

```
x[AUSTIN]
```

For obtaining the value of a composite key the expressions in the access list are evaluated according to the attribute sequence of the key definition. A field of the selected tuple is obtained by:

```
x[AUSTIN].sz
```

The type of the primary is derived from the table or tuple structure.

In the context of a descriptive loop construct tuples are not accessed by keys but in a system-defined manner. A designator (without access list) refers here to a tuple of the table, or a sub-table depending on the loop set-up. A selector refers to a field of a tuple or to a column of a sub-table.

If it is necessary to access a particular element of an array attribute then the field name can be further indexed by an expression which selects the element. The expression must evaluate to a value of the appropriate index type and within the bounds of the index range. With the declaration

```
var x: table [city; sz,pop<year>]
           with key city;
```

we can obtain the population of Austin in the year 1980 by:

```
x[AUSTIN].pop[@1980]
```

Some entities may have in addition to a 'value' other 'properties'. Properties consist of an identifying name and a value. Only for certain variables pre-defined property names exist; these also determine the type of their value (see Section 4.1.3). Properties are accessed by the property operator "'" followed by the property name. Example:

```
status'name
```

Other primaries together with examples are handled in later sections.

3.3.1.2 Operations

Simple Operations

Under this heading fall all operations except set operations. The definitions of arithmetic, logical, and comparison operators are the same as in Pascal. Example:

```
(x.pop>1000) or (x.sz<500)
```

All operations require scalar operands except the equality operators. Equality operators are also applicable to tuples and tables.

The 'take' operator is provided for tables which consist of one tuple. The result of the operation is the tuple. In case of a one column table (or set) the element of the table is returned as scalar.

Set Operators

Tables of the same structure and attribute types can be operands of the union, difference, and intersection operator. Tables may contain duplicates. The operations are defined by:

```
x in (S1 union S2)       iff (x in S1) or (x in S2)
x in (S1 minus S2)       iff (x in S1) and not (x in S2)
x in (S1 intersect s2)   iff (x in S1) and (x in S2)
```

The operators 'contains' and 'partof' denote improper set inclusion. The operands must be type compatible tables (or sets). The operations are defined by:

```
S1 contains S2      iff  for all x in S2: x in S1
S1 partof S2        iff  for all x in S1: x in S2
s1 = s2             iff  for all x in S1: x in S2 and
                         for all x in S2: x in S1
```

Set membership is denoted by the operator 'in'. The type of the tuple (or for sets the type of the scalar) must be compatible with the type of the table.

3.3.1.3 Constants

Constant values exist for simple types and structured types. If necessary a type operator must be applied to a constant to avoid ambiguities or to provide additional structure information.

```
$    sconst = string | TRUE | FALSE | UNKNOWN
$    slconst = ident | ["+"|"-"] number | sconst
$    tconst = typeop slconst
$    StructConst = [typeop] (SET setstruct | TUPLE tuplestruct |
$                  TableConst | EMPTY)
$    setstruct = "[" setElement {"," setElement} "]" | "[" "]"
$    setElement = expression [".." expression]
$    tuplestruct = "[" tupleElement {"," tupleElement} "]"
$    tupleElement = [expression]
$    typeop = "$" typeref ":"
$    typeref = ident | basicType | derivedType
```

Constant values of simple types like numbers are represented by their usual notation. Values of enumerated types may need a type operator for uniquely identifying a value. Example:

```
$color: RED
```

Constant values for structured types usually require a type operator to define the type of its attributes. Otherwise the type is derived from the constants (or expressions) used in the structure. Example:

```
set[1..3,7,9]
```

For tables it is convenient to provide the data in table form. The table form is defined by lexical rules and thus only the lexeme TableConst appears in the grammar. A declaration describing e.g. the number of telephone lines between two cities and the distance between the cities could be given by:

```
type telph = table [FROM,TO:city;
                    distance:real; lines:integer]
```

In the table formulation the attribute names are used as column names. The column names define the layout of the table. Column 1 within the table form is reserved for special purposes. A line containing column names is indicated by a plus symbol in the first column. Example:

```
$telph:table
+       FROM      TO       DISTANCE    LINES
          A         B          3.0         55
          X         Y          2.0         70
end
```

If necessary the table can be separated into parts if the column names do not fit onto one line. Extra lines containing column names are marked by a plus symbol in column 1. Example:

```
$telph:table
+                FROM     TO
                   A       B
                   X       Y
+     DISTANCE   LINES
        3.0        55
        2.0        70
end
```

Table constants with array attributes can be also formulated. A column name in this case consists of the attribute name followed by the index value which is used to identify the column. For the example above one might have three choices for the number of telephone lines. This could be formulated by:

```
  LINES.1       LINES.2       LINES.3
    30            33            37
```

The language provides two typeless constants 'unknown' and 'empty' to handle unknown values and empty structures. These constants are only useful for assignments and comparisons. Key attributes in tables cannot have the value unknown. In tuple constants and table constants values can be omitted. These values are assigned the value unknown.

3.3.1.4 Relation Constructor

The relation constructor creates a new table from existing tables. Based on the assumption that most data base operations are quite simple the relation constructor is restricted to provide the basic table operations of selection, projection, join, or a combination thereof. More complex operations can be realized either by decomposing the operation into a sequence of simple operations creating intermediate tables along the way, or by using the tuple-at-a-time facilities of the language (Section 3.3.2).

The relation constructor consists of three major parts: the result part, the input part, and the selection part. The descriptive nature of the relation constructor derives from the looping mechanism by which the tuples of the relation are accessed: no specific access sequence is defined.

```
$    RelationConstructor = "{" resultPart rc-op inputPart
$                          [selectionPart] "}"
$    rc-op = "|" | "||"
```

The resulting table may contain duplicates if the "||" form of the table constructor is used, otherwise duplicates are removed from the result table. We introduce first the three parts of the relation constructor. Then a variety of examples is given.

Definition

The result part of the relation constructor specifies the structure and the column names of the result table. All column names of the result table must be different; column names can be renamed if necessary.

```
$    resultPart = telement {"," telement} [WITH keydef]
$    telement = tident | ident "=" tident
$    tident = ident ["." ident]
```

The names in the target list are usually attribute names. They can be qualified by the table from which they are taken. A target list may consist of a table name if all attributes of the table participate in the result. For array attributes all columns are transfered to the result table.

The input part defines one or two tables from which the tuples are taken. The tuples are selected in a system-defined way. A table can be

associated with an auxiliary variable which serves as loop variable. The type of this implicit declared variable is derived from the type of the table. The scope of the variable is the relation constructor only.

If two tables are given then first a 'join' table is constructed according to the join condition. The loop of the relation constructor obtains tuples from this table for the selection and projection operations. The loop variables associated with the input tables are coupled to this loop; they refer to the part of the 'join' tuple which was obtained from the respective table.

```
$    inputPart = itable [BY bySelector] |
$                itable "," itable "(" tident J-operator tident ")"
$    itable = [ident IN] ident
$    J-operator = "=" | "<>" | "<" | ">" | "<=" | ">="
$    bySelector = ident ["[" expression "]"]
```

A table may be categorized into sub-tables according to some attribute with the 'by' construct. For each distinct value of the respective attribute a sub-table is formed. The loop in this case runs over the sub-tables, and the loop variable refers to a sub-table instead of a tuple. The attribute for the categorization of a table may not be an array attribute. However, a specific column of an array attribute can be used in the 'by' construct.

The selection part defines the conditions under which the tuple (or sub-table) obtained in the loop is to be included into the result table.

```
$    selectionPart = ":" expression
```

The expression used in the selection part may not contain further relation constructors or other table building operations. This restriction limits the complexity of the data base operations and is in line with our goal to match the complexity of notation with the complexity of the operation.

The treatment of sub-tables requires some further explanation. The conditions of the selection part are applied to a sub-table. Standard functions are available to obtain values or properties of a sub-table (or column thereof), e.g. sum, min, max, average, or size. If a sub-table qualifies for further processing then the whole sub-table is used. Usually only one tuple per sub-table will be added to the result table because duplicates are removed if only some of the columns are projected out for the result. In general, however, the whole sub-table is added to the result.

Examples

The relation constructor is demonstrated by examples. We introduce the familiar 'supplier' and 'part' relations. The relation constructors are first formulated as queries. For the following we assume the existence of these tables

```
var suppliers: table [name; suppno; city];
var part: table [suppno; partno; quantity];
```

with the appropriate attribute types. In some cases an auxiliary variable called 'temp' of the appropriate type is necessary to hold intermediate results.

Simple queries involve only one table. Example of a simple projection:

```
Get the names of suppliers.
{ x.name | x in supplier }
```

In most cases the loop variable is not necessary. The above query can be formulated shorter by:

```
{ name | supplier }
```

Example of a simple selection:

```
Get the suppliers located in Austin.
{ supplier | supplier: city=AUSTIN }
```

A table of supplier tuples is returned in this case.

The selection condition may depend on some properties of another table. A query of this kind can be usually formulated by a join operation. In many cases it is advantageous to break down the query into two parts: first an auxiliary table is created which is then used in the simplified query. This way the more expensive join operation can be avoided. Example:

```
Get the names of suppliers who supply part 3.
temp:= { suppno | part: partno=3 };
{ name | supplier: suppno in temp }
```

The query formulated as join operation is given by:

```
{ name | supplier,part(suppno=suppno): partno=3 }
```

The join of the two input tables is built according to the join condition on the identified columns. The columns must be type compatible. The resulting table may contain columns with the same name: if these columns are to be used in the result table then the renaming facility must be used. Note that an input table identifier can be dropped if the attribute names can be unambiguously identified within the context of the table constructor.

A result table which contains attributes of both tables must be formulated as join operation. Example:

```
Get the names of suppliers and the part numbers
```

```
they supply.
{ name;partno | supplier,part(suppno=suppno) }
```

Tables can be categorized into sub-tables with the 'by' construct. Tuples of a table can be selected according to the properties of sub-tables from another table. Example:

```
Get the names of suppliers who provide at least
10 different parts.
temp:= { suppno | part by suppno: size(part)>10 };
{ name | supplier: suppno in temp }
```

Finally a table constructor may have the same outcome in selecting a tuple from a table as accessing the tuple by key. If we assume that 'suppliers' was defined with the key 'suppno' then the following two constructs will retrieve the same tuple:

```
Get supplier with suppno=7.
take { supplier | supplier: suppno=7 }
supplier[7]
```

The limits of the relation constructor are reached by a query of the kind:

```
Get part numbers together with their total
quantity at hand.
```

As the resulting table contains a column of computed values this query must be formulated with a for-statement (Section 3.3.2.2). The input table is categorized according to the part numbers. For each sub-table a tuple is created which contains the part number together with the sum of the quantities. This tuple then is inserted into the result relation. The procedural formulation of the query is illustrated by:

```
ans:=empty;
for part by partno do
  insert tuple[take partno, sum(quantity)]
  into ans;
end;
```

3.3.1.5 Function Calls

An identifier followed by an expression list activates the respective function and stands for the value resulting from its execution. The declaration of functions is handled in Section 4.3.1.

```
$    FunctionCall = ident "(" [ActualParameters] ")"
$    ActualParameters = expression {"," expression}
```

Several standard functions are predefined. Some of these functions allow parameters from a class of types, or the type of the result may depend on the type of the parameter. These functions are called generic functions as for each context the proper function is available. The arguments of functions may be scalars or structures. The following is a list of functions provided:

Functions with Scalar Argument

ABS(x)	absolute value; result type is integer or real.
SIN(x),COS(x),TAN(x),ARCTAN(x), EXP(x),LN(x)	usual functions; result type is real.

Functions with Structure Argument

SIZE(x)	x is a table or a column of a table, returns the number of elements in the structure; result type is integer.
SUM(x)	x is a column of a table, returns the sum of the elements; result type is real or integer depending on type of column.
AVG(x)	x is a column of a table, returns the average sum(x)/size(x); result type is real.
MIN(x)	x is a column of a table, returns the smallest element of the column; result type is same as argument type.
MAX(x)	x is a column of a table, returns the largest element of the column; result type is same as argument type.

3.3.2 Statements

Statements exist for assigning values to variables or structures, for specifying the order in which computations are done, for input/output, and for solving a model. Statements are simple or structured. Structured statements contain statements themselves.

```
$    statement = [assignment | insertion | deletion |
$                 ProcedureCall | IfStatement |
```

```
$                LoopStatement | ExitStatement |
$                ForStatement | WithStatement |
$                Modification | InputOutput |
$                ExecuteStatement]
$    statementlist = statement {";" statement}
```

The statements are described briefly. Of importance are the data management statements and the for-statement. Input/output and the execute-statement are deferred to Section 4.4. For the following examples these declarations are assumed:

```
var persons:  table [name; salary:integer]
                    with key name;
var person:   tuple persons;
var rperson:  type persons;
var k,x:      integer;
```

3.3.2.1 Simple Statements

Assignment

Assignment is used to overwrite the current value of a variable by the newly computed value.

```
$    assignment = reference ":=" expression
$    reference = ident ["." ident ["[" expression "]"] ]
```

If an assignment is made to a variable connected to the data base then the old values in the data base are deleted and new values are created. Example:

```
persons := {persons | persons: salary>500}
```

The syntax does not allow the assignment to a field of a tuple which is obtained by key from a table. The modification statement must be used for this purpose.

Insertion

Tuples are added to tables by the insert statement.

```
$    insertion = INSERT expression INTO ident
```

The expression must evaluate to a tuple which is type compatible with the table into which it is to be inserted. If the table has a key property then the tuple is only inserted if there are no key conflicts. Example:

```
insert person into persons;
```

```
insert tuple[SMITH,10000] into person;
```

Deletion

Tuples are deleted from a table by the delete statement.

```
$    deletion = DELETE [ALL] expression [FROM ident]
```

The expression must evaluate to a tuple which is type compatible with the table. (In case of duplicate entries only one tuple is deleted from the table unless the ALL construct is used). The table from which the tuple is to be deleted may be omitted if the tuple is produced by key access or by a descriptive loop construct. Example:

```
delete persons[SMITH];
delete person from persons;
```

Procedure Call

An identifier followed by an expression list activates the respective procedure. The declaration of procedures is handled in Section 4.3.1.

```
$    ProcedureCall = ident ["(" ActualParameters ")"]
```

Two standard procedures are predefined which select the current input file or current output file: inputfile and outputfile. The argument of these routines is a string. Example:

```
inputfile("mod.data")
```

3.3.2.2 Structured Statements

If-Statement

The if-statement is used to make decisions. The expression must evaluate to a boolean value.

```
$    IfStatement = IF expression THEN statementlist
$                  [ELSE statementlist] END
```

Example:

```
if k>3 then x:=5 end;
if k=0 then x:=3 else x:=x+5 end;
```

Loop-Statement

The loop-statement specifies the repeated execution of a statement sequence. The loop is terminated by the execution of an exit-statement.

```
$    LoopStatement = LOOP statementlist END
$    ExitStatement = EXIT WHEN expression
```

An exit-statement may only be used within a loop-statement. When an exit-statement is executed then only the immediately enclosing loop is terminated. Example:

```
k:=5;
loop f(k); exit when k=0; k:=k-1 end;
```

Other looping constructs like the 'while' and 'repeat' constructs found in Pascal can be expressed by placing the exit-statement respectively as the first or last statement in the loop.

For-Statement

The for-statement specifies a descriptive loop construct. It has essentially the same form as the selection part of a relation constructor. The expression which represents the condition must evaluate to a boolean value.

```
$    ForStatement = FOR ForSelection DO statementlist END
$    ForSelection = selection [":" expression]
$    selection = [ident IN] ident [BY bySelector] |
$                ident IN basicType
```

Elements are either selected from a table or implicitly from a range of values given by a simple type. An auxiliary loop variable may be declared for the loop. Its type is derived from the associated table or type. The scope of the loop variable is the for-statement only. For tables (or sets) the for-statement obtains in a system defined way a tuple of the table (or element of the set), checks the conditions associated with it, and if the conditions hold, executes the statement list of the for-statement. Tables may be categorized into sub-tables with the 'by' clause. The loop variable then runs over the sub-tables. The same restrictions for array attributes apply as for the selection part of a relation constructor. Note that the syntax allows only a table identifier in the selection part. This means that an expression, e.g. a set constant, must be assigned to a table variable for use in the for-statement. If no loop variable is specified then the table name itself can be used if this does not cause any ambiguities.

The second form of the for-statement uses a simple type as value generator. The simple type must be discrete and restricted by a range. A simple type in the context of this construct is treated as a set of

values. The values are available either implicitly, e.g. integer[1..10] or explicitly e.g. an enumerated type. A loop variable must be specified if the type is not declared in a type declaration. Example:

```
for p in persons: p.salary<300 do ... end;
for name do ... end;
```

With-Statement

The with statement allows attribute identifiers to be used in its statement sequence without qualifying them by the name of the table or tuple variable.

```
$    WithStatement = WITH ident DO statementlist END
```

Example:

```
with person do
  name:= SMITH;
  salary:= 5000
end;
```

Modification

Tuples in a table are modified by the modification statement.

```
$    Modification = MODIFY ident [accessList] ( WITH | DO )
$                   statementlist END
```

Access of a tuple is either by key or in the context of a descriptive loop construct. The 'with' form of the modification statement allows the use of attribute names in the statement list without the need of qualifying them with the tuple name. Modifications to values of key attributes are not allowed. If key attributes must be changed then it is necessary to create a new tuple. The old tuple is then deleted from the table, and the new tuple is inserted. Example of modification statement:

```
modify persons[SMITH] with salary := salary+100 end;
```

Further Examples

The for-statement is useful for insertion, deletion, and modification of data in the data base. The for-statement is also necessary in formulating more complex queries which cannot be expressed by a table constructor. The procedural form of a table constructor is demonstrated by the following example:

```
-- Get persons with salary>5000
rperson:=empty;
for p in persons: salary>5000 do
  insert p into rperson;
end;
```

Tuples of a table for which a certain condition holds can be deleted with a for-statement. As the tuple to be deleted is readily identified through the loop construct the short form of the delete-statement can be used. Example:

```
-- Delete persons with salary>5000
for p in persons: salary>5000 do
  delete p;
end;
```

Finally tuples of a table for which certain conditions hold can be modified. Example:

```
-- Increase the salary of all persons with
-- salary<500 by 10
for p in persons: salary<500 do
  modify p with salary:= salary+50 end;
end;
```

Model Formulation and Formulation Modules

In this chapter the language constructs for model formulations are introduced. Model modules, declaration modules, routine modules, execution modules, and data declaration modules are described. The language constructs are defined by grammar rules. (The notation for grammar rules is defined in Chapter 3).

4.1 Model Module

A model module consists of three parts specific to this module: a declaration part, an equation part, and a problem formulation part. Model modules are named.

```
$    ModelModule = MM-header M-declarations M-equations
$                  problemsection END
$    MM-header = MODEL MODULE ident ";"
```

In the following the parts of the model module are described further. To demonstrate the language structures we use the transportation problem as introduced in Section 2.6. A complete formulation of a model module is given in Appendix B.

4.1.1 Model Declaration Part

The model declaration part consists of type, variable, and routine declarations.

```
$    M-declarations = {D-declaration} {R-declaration}
```

The declared variables fall into two categories: variables which are used for model variables and model parameters of the equations, and variables which are used for data manipulations, e.g. index computations. All variables in the model declaration part are considered to be local variables. However, they may be used in the interface specification part of a problem in the problem section. In this case (and only in the context of the particular problem) they are considered to be formal parameters of the model module.

We use as example the declaration part of the transportation model. Goods are produced at plants and shipped to markets. Data structures are needed for the capacity of plants, the market requirements, the amount

shipped, and the shipping cost. The declaration part is formulated as follows:

```
-- declarations
var I: set plant;
var J: set market;
var capacity:     table [plant; k:real]
                        with key plant;
var requirement:  table [market; r:real]
                        with key market;
var cost:         table [plant; market; c:real]
                        with key plant, market;
var amount:       table [plant; market; x:real[positive]]
                        with key plant, market;
```

Note that the sets I and J in this example will be used for index computations.

4.1.2 Equation Part

The equation part contains the equations of the model. This includes bounds and objective function. An equation is defined with a name and possibly parameters. The parameters are usually set structures. They can be only used for indexes.

```
$    M-equations = EQUATIONS M-equation {M-equation}
$    M-equation = equationHeader IS equation ";"
$    equationHeader = ident ["(" Q-formal {";" Q-formal}")"]
$    Q-formal = ident {"," ident} ":" typeref
```

An equation may be simple, or it may represent a group of equations. A group of equations is generated by a descriptive 'each' construct which is similar to a descriptive loop. The 'each' construct consists of a selection part and a condition part. An auxiliary loop variable may be defined. The scope of this variable is the equation only. The values for the loop may be taken from an index set, or a simple type may serve as value generator.

```
$    equation = [EACH eachpart] Q-leftpart Q-rightpart
$    eachpart = Q-selection {"," Q-selection} ":"
$               [expression] ":"
$    Q-selection = [ident IN] ident | ident IN basicType
```

When an equation is executed it generates one or more rows of the LP matrix. If it is necessary to refer to one of these rows then the equation name together with the index which generated it can be used.

The 'each' construct is also used for the running index of a summation sign. The summation is represented by the 'sigma' construct.

```
$    summation = SIGMA "(" eachpart Q-expression ")"
```

The condition part of the 'each' construct is treated in a similar way as the condition part of a for-loop: only if the conditions hold is the rest of the construct processed further. The summation

$$\sum_{i \in I} \sum_{\substack{j \in J \\ i \neq j}} x_{ij}$$

is expressed in the modeling system by

```
sigma (i in I,j in J: i<>j: x[i,j].x)
```

Equations consist of a left part and a right part. The left part has the form of an (equation) expression. It must be linear in the model variables. The right part for inequalities, equalities, and ranges must evaluate to a constant, or to a lower and upper bound respectively. For objective functions the right part consists only of the equation operator.

```
$    Q-primary = designator | number
$    Q-factor = Q-primary | summation | "(" Q-expression ")"
$    Q-term = Q-factor {QM-op Q-factor}
$    Q-expression = ["+"|"-"] Q-term {QA-op Q-term}
$    QM-op = "*" | "/" | MOD | DIV
$    QA-op = "+" | "-"
$    sQ-primary = ["+"|"-"] Q-primary
$    Q-leftpart = Q-expression
$    Q-rightpart = Q-op sQ-primary | OBJECTIVE |
$                  RANGE sQ-primary "," sQ-primary
$    Q-op = EQ | LEQ | GEQ | = | <= | >=
```

The equations of the transportation problem with the declarations given above can now be formulated:

```
equations
  -- a plant cannot ship more than it produces
  cpconstraint is
  each i in I::
    sigma (j in J:: amount[i,j].x) <= capacity[i].k;

  -- a market must at least receive what it requires
  mkrequirement is
  each j in J::
    sigma (i in I:: amount[i,j].x) >= requirement[j].r;

  -- shipments must be positive
  nonnegbound is
  each i in I, j in J::
    amount[i,j].x >= 0;
```

```
-- cost function
totcost is
  sigma (i in I,j in J:: amount[i,j].x*cost[i,j].c)
  objective;
```

4.1.3 Problem Section

The problem section consists of one or more problem definitions. A problem describes the actual model to be solved. It consists of five parts: a declaration part for local variables, an interface specification, a computation part for indexes, the solve statement, and another computation part for general data manipulation.

```
$    problemsection = problem {problem}
$    problem = problemHeader {TV-declaration} [interfaceSpec]
$              [statementlist] SolveStatement [statementlist]
$              END
$    problemHeader = PROBLEM ident ";"
$    interfaceSpec = INTERFACE ["*"] ident {"," ["*"] ident} ";"
```

The interface specification defines the formal parameters of a problem. The variables mentioned in the interface specification list may refer to variables declared in the problem declaration part as well as to variables declared in the module declaration part. If an asterisk precedes the formal parameter then the parameter is an input parameter. The corresponding actual parameter is expected to supply a value with which the local variable of the model module (or problem section) is initialized when the problem is executed.

The solve statement consists of the model solution system interface and the description of the actual model. The actual model is defined by calls to the relevant equations of the model formulation. Model variables, constraints, and bounds are identified in separate sections. It is also indicated if the objective function is to be minimized or maximized.

```
$    SolveStatement = SOLVE SS-interface actualmodel END
$    actualmodel = variables objective constraints [bounds]
$    variables = VARIABLES ":" v-ident {"," v-ident } ";"
$    v-ident = ident ["." ident]
$    objective = OBJECTIVE ":" option equationcall
$    option = MINIMIZE | MAXIMIZE
$    constraints = CONSTRAINTS ":" equationcall {equationcall}
$    bounds = BOUNDS ":" equationcall {equationcall}
$    equationcall = ident ["(" expression {"," expression}")"] ";"
```

The equations called in the different sections must be of the appropriate kind, e.g. an objective equation cannot be called from a constraint

section. If the bounds section is omitted then the bounds are derived from the basic type with which the model variables are declared.

The model solution system interface is specific to the solution system in use. It has the general form: name of solution system followed by actual parameters. The actual parameters are identified by the name of their formal parameter to establish the proper correspondence. Parameters are used in two ways: they declare implicit variables which contain information on the solution, or they are used to pass information along to the solution system.

```
$    SS-interface = "(" ident ":" SS-parameter
$                   {"," SS-parameter} ")"
$    SS-parameter = ident ["=" expression]
```

The right side of a declaration parameter can be omitted; the parameter name is then used by default for declaring a variable. The scope of implicit declared variables is the computation part following the solve statement of the current problem formulation.

It is assumed that the three parameters 'status', 'cost', and 'parm' are always provided by the solution systems. The first two are used to declare variables implicitly. 'Status' is a variable of type integer which returns the status of the solution. 'Cost' is a variable of type real which returns the value of the objective function. Parameters can be passed along to the solution system with the parameter 'parm'. Its argument is of type string.

Implicit declared variables of the model solution system interface have additional properties which can be accessed by the property operator "'". For a given solution system the translator is aware of the possible implicit variable declarations and their available properties. Thus checks on their correct use can be made at translation time.

The 'status' variable has the property 'name' which contains a string of the name of the status of the solution. This information could be used like in this example:

```
if status'name = "optimal" ...
```

If an array attribute was involved for providing several righthand sides for the model solution then solution information for a particular column can be obtained by using the same index value with which the righthand side of the LP model was generated. Example:

```
if status'name[@1980] = ...
```

For an LP solution system two additional interface parameters for declaring implicit variables are defined: 'row' and 'column'. These variables contain the relevant information usually available from an LP system. Both are table structures. Access to a tuple is accomplished

either by giving the equation name or the name of the model variable. Additional indexes are usually required. As the tables are declared implicitly the value of a field is obtained by the property operator, e.g. the input cost of a variable in the final LP solution could be obtained by

```
column[amount,i,j]'inpcost
```

or the value of a slack variable for an equation (if it exists) by

```
row[mkrequirement,i]'slack
```

Solution information for multiple righthand sides is obtained by indexing further with the appropriate righthand side index value.

The problem formulation for the transportation problem looks now as follows:

```
problem P;
  -- problem interface
  interface capacity, requirement, cost, amount;

  -- index computations
  I:= {plant | capacity};
  J:= {market | requirement};

  solve (PDQLP: status)
    variables:   amount.x;
    objective:   minimize totcost;
    constraints: mkrequirement;
                 cpconstraint;
    bounds:      nonnegbound;
  end;

  if status'name="optimal" then .... ;
end;
```

The problem is to be solved with the LP system PDQLP, and the only implicit declared variable is 'status'. The bounds section could have been omitted as these bounds are also derivable from the type of the model variable.

4.2 Declaration Module

Logically related variables and types can be collected in a declaration module. A declaration module can be included in other modules instead of declarations.

```
$    DeclarationModule = DM-header {D-declaration} END
$    DM-header = DECLARATION MODULE ident ";"
$    TV-declaration = TypeDeclaration ";" |
$                     VariableDeclaration ";"
```

```
$    D-declaration = TV-declaration |
$                    INCLUDE ident {"," ident} ";"
```

The data structures of the transportation model (Section 2.6) can be formulated as follows:

```
declaration module TM;
var capacity:     table [plant; k:real]
                       with key plant;
var requirement:  table [market; r:real]
                       with key market;
var cost:         table [plant; market; c:real]
                       with key plant, market;
var amount:       table [plant; market; x:real[positive]]
                       with key plant, market;
end;
```

In addition to the variable names also the type names 'plant' and 'market' are declared in the above module.

A declaration module is included into another module by the 'include' construct. Example:

```
include TM;
```

The inclusion of a declaration module is handled as if the text of the module was inserted. A declaration module is not used for sharing variables like the COMMON declaration in Fortran; the interface specification is provided for this purpose. All names of a declaration module are visible in the scope of the module where it is included. The scope of a declaration module, however, is closed in the sense that all names like default type names are defined.

4.3 Routine Module

The routine module consists of a collection of functions and procedures. Variables which are global to these routines can be declared in a declaration part.

```
$    RoutineModule = RM-header {D-declaration} {R-declaration}
$                    END
$    RM-header = ROUTINE MODULE ident ";"
```

4.3.1 Routine Declarations

The declaration of functions and procedures follows the established pattern of programming languages. The declaration specifies the name of

the routine, its formal parameters (if any), and in the case of a function the type of the returned value. This is followed by the body of the routine.

```
$    R-declaration = routineheader {TV-declaration}
$                    BEGIN statementlist END ";" |
$                    use-declaration ";"
$    routineheader = PROCEDURE ident [formalPart] ";" |
$                    FUNCTION ident [formalPart] ":" typeref ";"
```

A routine may contain local variables; however, no nested routine declarations are allowed.

Formal parameters of a routine may be declared in one of three modes: by reference, by value, and as constant.

```
$    formalPart = "(" parameterDeclaration
$                 {";" parameterDeclaration} ")"
$    parameterDeclaration = [mode] ident {"," ident} ":" typeref
$    mode = VALUE | CONST
```

If the formal parameter is declared without an explicit mode then the actual parameter is passed by reference. This means that any data manipulations carried out on the formal parameter directly affect the actual parameter. This kind of parameter corresponds to the VAR parameter in Pascal. If the formal parameter is declared as VALUE parameter then a copy of the actual parameter is passed to the routine. The formal parameter is handled like a local variable initialized with the value of the actual parameter. If the formal parameter is declared as CONST parameter then the actual parameter can be accessed only but may not be modified. Tables cannot be passed as value parameters.

4.3.2 Global Variables

Variables declared for a routine module have the role of 'static' variables (e.g. like in the programming language C), that is they are privat global variables of that module. The global declarations of a routine module are useful if a large number of variables must be shared among the routines; it provides an alternative to function and procedure parameters for communicating data.

4.3.3 USE Declaration

A routine module is included into the current module by a use-declaration. Usually all routine names declared in a routine module are visible in the scope where the module is used. The visibility can be limited by importing only specific names.

```
$    use-declaration = USE ident [IMPORT ident {"," ident}]
```

Use-declarations may appear where functions or procedures can be declared. The types of the parameters (and function results) must be defined in the module which uses the routine module. These definitions must be compatible with the definitions in the routine module.

4.3.4 Example

The following example shows the skeleton of a routine module:

```
routine module RM;
  include TM;

  procedure report1(const mm: type amount);
  begin ... end;

  procedure line;
  begin ... end;
end;
```

The routine 'report1' may be used in another module if the module RM is included in it by a use-declaration. Example:

```
use RM  import report1;
```

The visibility of the routines of the routine module RM is restricted to the routine 'report1'; the routine 'line' for example is not accessible in this module.

4.4 Execution Module

The execution module defines the operational aspects of a model. It consists of input/output statements, data manipulation statements, and problem execution calls. Input/output usually is limited to the definition of the interface between data base and execution module.

```
$    ExecutionModule = EM-header {D-declaration} [interfaceSpec]
$                      {R-declaration} statementlist END
$    EM-header = EXECUTION MODULE ident ";"
```

The execution module is used for data transformations. Data transformations may be required to prepare data for a problem in the appropriate form or for storing data obtained from a problem solution in the data base. An execution module does not have to contain a call to a problem execution: the execution module then can be used to enter data into the data base or to present data from the data base.

4.4.1 Execution Statement

The execution of a problem of a model module is accomplished by the execute-statement.

```
$    ExecuteStatement = EXECUTE ident OF ident sharedVariables
$    sharedVariables = "(" ident {"," ident} ")"
```

The execution of problem P of the transportation model might be formulated by:

```
execute P of
  transportation(capacity,requirement,cost,amount);
```

The actual parameters of the model module are usually variables which define the interface between the execution module and the model module. For those formal parameters which have been declared with an asterisk the actual parameters are evaluated. Thus for these parameters also expressions may be used.

The execute statement can be part of a statement sequence in the execution module, e.g. it can be part of a loop. This may be useful when trying to solve a problem in an iterative fashion.

4.4.2 Input/Output Statements

As the execution module is also concerned with input/output we handle input/output statements here. Input may be obtained from a file or from the user. Output may be sent to a file or to the terminal. The current input file is selected by the standard routine 'inputfile', the current output file by the standard routine 'outputfile' (see Section 3.3.2.1). Input for the command processor is always expected from the terminal (or a command module). Thus the terminal is automatically selected as input file when a command finishes, or when errors occur during the execution of a command (and input was read from a file).

Two routines READF and WRITEF are provided for formatted input/output. The corresponding routines for unformatted input/output are READ and WRITE.

```
$    InputOutput = WRITE "(" expression {"," expression} ")" |
$                  WRITEF "(" expression {"," expression } ")" |
$                  READ "(" [sentinel ","] reference [sentinel]
$                  {"," reference [sentinel]} ")" |
$                  READF "(" expression [sentinel] "," reference
$                  [sentinel] {reference [sentinel]} ")"
$    sentinel = ":" expression
```

The first parameter of the routine WRITEF is a string which contains the control characters on how to convert the following expressions. The

format layout and conversion characters have been adapted from C [Kernighan]. Conversion specifications are separated by commas. A conversion specification starts usually with a digit sequence which gives the field width of the item to be printed. This is followed by the conversion character. The field width may be preceded by a minus sign indicating that the item is to be displayed left-adjusted in the given field instead of right-adjusted. Furthermore a second digit sequence separated by a period may be included in some cases. The conversion characters and their meaning are:

d The argument is displayed as integer.

e The argument is displayed as floating point number. A second digit sequence is required; it gives the number of digits in the mantissa.

f The argument is displayed as fixed point number. A second digit sequence is required; it gives the number of digits following the decimal point.

s The argument is a string, an enumerated type, or a boolean.

t Before the next argument is handled, tab to the absolute position given by the digit sequence.

n Write an end-of-line.

Tables may be directly used as arguments of WRITEF. For each attribute a format specification is required. For array attributes the format of the attribute is applied to each column. The whole table is displayed according to the format string; end-of-lines are inserted automatically after each tuple of the table has been displayed. The table 'amount' of the transportation problem could be printed by

```
writef("10s,10s,10.2f",amount);
```

The routine READF is very similar to WRITEF. The first parameter is a control string giving the format on how to read the data for the following parameters. With the exception of the first parameter all parameters of READF are variables. The conversion characters are very similar to the ones of WRITEF, however, only one digit sequence to define the field width for floating point numbers is necessary. The end-of-line character is used to get the next input from a new line. Input is read according to the type of the argument; the format specification must be compatible with the type. String format is used to read values of enumerated types as well as string types. Leading and trailing blanks are eliminated of the character sequence obtained from the given field width. Real numbers may appear in the defined field width as integers, fixed point numbers or floating point numbers. Zeroes must be written except that a field consisting of spaces is interpreted as the value zero.

A table may be directly used as parameter of READF. It is assumed that each tuple of the table begins at a new line. A tuple may be spread over a few lines. The format specifications are matched according to the sequence of attributes of the table. A particular problem arises with finding the end of a table. Two methods are provided for this purpose: a predefined number of tuples can be read, or tuples are read until an end-of-table marker is found. The table parameter is extended with an integer argument in the first case, and with a string argument in the second. The integer argument is used as counter of an internal loop. The string argument defines the pattern which must appear in the input (starting at column 1) which indicates the end of the table. The table 'capacity' could be read by:

```
readf("10s,10f",capacity:"*");
```

with the input

```
plant1        3.7
plant2        1.8
*
```

Input can be also obtained within a loop. A sentinel pattern may be defined which can be used to terminate the loop implicitly. This sentinel pattern is defined as extension of the control string. Example:

```
loop
  with capacity do
    readf("10s,10f,n":"*",plant,c);
  end;
end;
```

Note that here the change to a new line must be explicitly defined. The loop is exited when the defined pattern is encountered in the input stream.

Simple output is performed with WRITE. The arguments of WRITE are printed in a standard way according to their type. The sequence printed by the argument list of WRITE is automatically terminated by an end-of-line. Table arguments can be used with WRITE; each tuple starts at a new line.

Input is read format-free by READ. Each token is read according to the type of the argument. The input tokens are assumed to be separated by blanks (or end-of-line). This way it is possible to read values of enumerated types and strings without special indicators, though quote symbols are necessary if a string is to contain blanks. End-of-lines are not significant, except that sentinel character sequences start in column 1. Sentinel patterns are defined in a similar way as in the formatted case. A '0-th' parameter can be extended by a sentinel definition if READ is used within a loop. Values for the table 'capacity' could be obtained by:

```
read(capacity:"*");
```

with the input

```
plant1 3.7 plant2 1.8 plant3 9.3
*
```

4.5 Data Declaration Module

The data declaration module is necessary to define the initial data structures of a data module. The data declaration module is very similar to a declaration module.

```
$    DataDeclModule = DDM-header D-declaration {D-declaration}
$                     [INITIALIZE statementlist] END
$    DDM-header = DATA DECLARATION MODULE ident ";"
```

The index types of array attributes may still be left unspecified. When a data module is created by the data declaration module these types will be completely defined: Values for enumerated types and bounds for subrange specifications are supplied. The creation of data modules is handled in Section 5.1.2.

The initialization part of a data declaration module is optional. It consists of assignments of constant values, like table constants. The assignments are carried out when the data module is created.

4.6 Translation Unit

One or more formulations can be stored on a file and presented to the modeling system as a translation unit.

```
$    TranslationUnit = module ";" { module ";"}
$    module = ModelModule | DeclarationModule | RoutineModule |
$             ExecutionModule | DataDeclModule | CommandModule
```

The modules are translated sequentially. Though they are compiled separately attention must be paid to their sequence so that inter-module dependencies can be dealt with. A declaration module for example must be already defined in the data base before it can be included in another module; thus it must precede those modules on the file.

In addition to formulation modules handled in this chapter a translation unit may contain command modules.

```
$    CommandModule = CM-header commands END
$    CM-header = COMMAND MODULE ident ";"
```

A command module is a collection of commands which a user might issue in a dialogue to accomplish a task, but which now can be used on a routine basis. Command modules are taken up again in Section 5.1.5.

Command Language and Browsing

This chapter deals with the user interaction facilities of the modeling system. The command language for defining, altering, and executing modules is introduced. Data modules and the facilities to browse through data are described.

5.1 Command Language

The command language is employed by the user to interact with the modeling system. Each command starts with a keyword that identifies its main function. Some commands put the user into different modes of interaction, and further commands may be available in this context. At each level the user can obtain a menu of available commands. The commands at the top level of the system are summarized in Figure 5-1. Each command is usually associated with some arguments. Command words can be shortened to the first few letters as long as they can be recognized uniquely.

Commands come in two varieties: short commands and long commands. A short command together with the arguments must fit onto one line; the end-of-line acts as terminator. Most of the above commands are short commands. The exceptions are 'create' and 'run' which are long commands. A long command usually requires a few lines for its arguments. It is terminated by 'end'.

BROWSE	enter 'browsing' mode
CATALOG	give directory of modules
CREATE	create a data module
DO	execute a command module
EDIT	interactively edit the text of a module
PROCESS	process a file of formulation modules
QUIT	quit the modeling system
REMOVE	remove a module
RUN	run an execution module
SHOW	display a module
USAGE	list module dependence
?	give menu of commands
??	describe command further

Figure 5-1 Top level Commands

The ?? command can be used as help command in general. It gives information about the current interaction mode the user is in. The user

may be parsing a command or a module, or he may be in command mode, browsing mode, or in data input mode. Depending on the context more specific information is given, e.g. in data input mode the user is informed for which particular item input is currently expected. The help command may be followed by a command name. In this case the command is further described, e.g. ?? RUN gives help with the form and usage of the run command. Only if the ?? starts a line is it recognized as the beginning of a help command.

5.1.1 Simple Commands

We describe first the commands which return the user back to top level command mode. A catalog of all modules which are stored in the data base is obtained by the 'catalog' command. The command may be restricted to a particular category of modules. A category is identified by one or two letters as follows:

c	command module
d	declaration module
e	execution module
m	model module
r	routine module
dd	data declaration module
dm	data module

A catalog of model modules and data modules could be obtained by:

```
catalog m,dm
```

New module formulations are entered into the system with the 'process' command. It specifies a file which contains one or more modules. The modules are translated and entered into the data base. Example:

```
process "model.1"
```

The file name is given by a string. A module can be entered from the terminal if "tty" is specified.

A formulation module can be displayed on the terminal with the 'show' command. As modules in the various categories may have the same name the module category must be used to uniquely identify the name. Example:

```
show m.transportation
```

The interdependency of modules can be listed by the 'usage' command. Example:

```
usage d.TD
```

This will list all the modules which "included" the declaration module TD.

Any kind of module may be removed from the data base by the 'remove' command. Example:

```
remove d.TD
```

The user is reminded of the existence of other modules which may depend on this module, and in this case an explicit confirmation must be given by the user to remove the module.

Finally a session with the system can be ended by the 'quit' command. All other commands of the top level enter a different interaction mode. The commands together with their sub-commands are described in the following sections.

5.1.2 'Create' Command

The 'create' command is used to create a data module from a data declaration module. The data module DM1 is created with the data declaration module DDM by:

```
create DM1 with DDM
end
```

If the data declaration module contains incomplete type declarations for index types of array attributes or subranges then details for these must be supplied now. This is accomplished by a re-declaration of these types. The user is either prompted interactively for the type declarations, or the type declarations can be supplied in a parameter-like fashion. Example:

```
create dm1 with ddm

  type year = (@1971,@1980,@1981);
  type lgt = integer[1..10];
end
```

After the data structures are defined the initialization part of the data declaration module is executed. Variables not initialized have the value 'unknown', tables the value 'empty'.

Each data module defines its own encoding of its enumerated types and strings. This usually does not matter as user interaction or model solutions are limited to one data module at a time; also it is more efficient to use their internal representation. Only when data are moved from one data module to another is it necessary to go through their external representation.

5.1.3 'Run' Command

The 'run' command is used to execute an execution module. It binds the data of a data module to the execution module. Parameters for the interface declarations of the execution module are provided on a name basis instead of position. The execution module EX could be run with the data module DM1 by

```
run EX with DM1
  requirement = r1;
  amount      = a1;
  cnt         = 3;
end
```

This assumes that the interface definition in EX is given by

```
interface *cnt,requirement,amount;
```

and that the data structures 'a1' and 'r1' are available in the data module DM1.

Type compatibility between actual and formal parameters of an execution model needs special attention. As data modules are independent from formulation modules we have two universes of types which must be matched when the data structures of a data module are connected to the execution module. For this purpose attribute names of structures of the interface, and external declared enumeration type names are considered implicit formal parameters of the execution module. The structures of actual and formal parameters must be the same. A type definition for tables includes the key definition. The types in the execution interface are defined to be compatible with the types of their actual parameters if the underlying type is the same, except for enumerated types explicitly declared in the execution module; here names and values must match.

The example given below defines a declaration module and an execution module which is used for demonstrating parameter compatibility of the 'run' command. Note that first a data module is created with the data declaration module. Data for the data module could be obtained in several ways: through the initialization part of the data declaration module, by entering data in data input mode, or by creating values in browsing mode. Browsing mode and data input mode are described later.

```
data declaration module DDM;
  type year=(@1980,@1981);
  type mon =(a,b,c);
  var part: table [partno; k<year>:integer]
              with key partno;
  var ssn:  table [ssno; q<mon>:integer]
              with key ssno;
  ...
end;
```

```
execution module EX1;
  type span = enumerated;
  var mp: table [no; cnt<span>:integer]
            with key no;
  interface mp;
end;
```

A command sequence might be:

```
create DM2 with DDM end
run EX1 with DM2
  mp = part
end
run EX1 with DM2
  mp = ssn
end
```

In the first 'run' command the tables 'mp' and 'part' are structure compatible. Also the types 'partno' and 'year' are automatically provided for the 'formal' types 'no' and 'span' of the execution module. The table 'ssn' is similar to 'part' in structure and type and thus can be also used as an actual parameter.

5.1.4 'Edit' Command

The 'edit' command is used to interactively edit the text of formulation modules and command modules stored in the modeling system. It is a very simple line-oriented text editor. Lines are accessed by line number. Lines are always numbered relative to the first line. Deletion and insertion operations automatically adjust the line numbering. A command consists of a line number range and a command character. The following edit commands are provided:

```
d          delete
i          insert
r          replace
p          print
```

A line range consists of one or two numbers. The last line can be referred to by '*'. Examples:

```
1,5p
*d
```

The insert command (and replace command) enters insert mode. All lines entered by the user are collected. A single period on a line (or an end-of-file condition) terminates insert mode.

Edit mode is terminated either by the commands 'end' or 'process'. In the first case a flag is set to indicate that the internal form of the module is now different from its external form. In the second case the

module is translated. If there exist any modules which depend on this module then the user is asked to re-translate these modules.

5.1.5 'Do' Command

The 'do' command starts a sequence of commands stored in a command module. A command module is to a limited extent comparable to a control card macro; it can be used with parameters. The parameters may be identifiers, numbers, or strings which are used in the body of the command module for text substitution. Example:

```
do try1 "x",1
```

The formal parameters of a command module are imbedded in the text where the substitution is to take place. The parameters are referred to by $1, $2, etc. Example:

```
command module TRY1;
process $1
run EX with DM$2 ...
end
```

With the 'do' command above this results in the commands:

```
process "x"
run EX with DM1 ...
```

5.2 Browsing Mode

In browsing mode the user has direct control over a data module. The user enters browsing mode by attaching to a data module with the 'browse' command:

```
browse DM2
```

In browsing mode the user is prompted for entering declarations, expressions, statements, or further commands. Each construct entered must be terminated by a semicolon.

DESCRIBE	describe variables or types
ENTER	enter data
KEEP	keep newly created data structures
LIST	list items in data module
REMOVE	remove item from data module
RETURN	return to command mode
SHOW	display contents of data item
?	give menu of commands
??	get help

Figure 5-2 Commands of Browsing Mode

A variety of commands is available with which the user can look at the data structures and data, and with which the data can be manipulated. The commands are summarized in Figure 5-2.

5.2.1 Simple Commands

Simple commands return the user back to browsing mode. Most of these commands provide information about a data module.

The items in the attached data module can be listed by the 'list' command. The command can be further modified by 'new' and 'types' or 'variables'. Example:

```
list new variables;
```

In this case only the variables which were created during browsing mode are listed. The names of the items and their structure, e.g. if the item is a scalar, tuple or table, are displayed.

A more detailed description of an item is obtained by the 'describe' command. For a table for example all attribute names and their types are given.

The contents of a data item is displayed by the 'show' command. The attribute names of tables and tuples are also given.

Items of a data module are removed by the 'remove' command. Types can be only removed if there exist no current data structures which use them.

Finally browsing mode can be ended by the 'return' command. This puts the user back into command mode.

5.2.2 Expressions and Statements

The main feature of the browsing facility is the possibility of evaluating expressions and statements interactively. The data manipulation facilities are a sub-set of those available for model formulations.

Of particular importance is the relation constructor with which the queries are formulated. When an expression is entered then the result is displayed on the terminal. Statements on the other hand are processed without creating output. However, the user is informed when possible that an action has been carried out. This information is in the form of how many tuples have been affected by the action. After an expression or statement is evaluated the user is prompted for the next input.

The available statements are the assignment, delete, insert, and modify statement. The table handling statements can be used in

conjunction with the for-statement. The following example shows a short dialogue with the user in browsing mode (the prompt character is '>'):

```
 browse DM2
>list variables;
   part   ssn
>show part;
   partno    k.1980    k.1981
   1123      10        10
   3725      11        12
>{partno | part: k[980] > 10};
   3725
>p:={partno | part};
   2 tuples
>describe p;
   var p: table [partno:partno]
>delete part[3725];
   1 tuple deleted
>
```

Of interest here is the assignment statement. In browsing mode the assignment statement can be also used to declare a new variable. The type of the variable is derived from the right-hand side, e.g. from the type of the relation constructor in the example above. This facility is useful for creating temporary data structures while the user is browsing through the data. If the user forgot a declaration, or wants to know of what type the newly created variable is then this information can be obtained by the 'describe' command. The user is reminded of the temporary variables when browsing mode is terminated. These variables and their associated values are deleted at that time unless they are kept with the 'keep' command. The newly created variable p in the example above is made part of the data module by:

```
keep p;
```

5.2.3 Declarations

As the user has control over the data module it is possible to declare new types and variables. The user can enter type or variable declarations in the same manner as expressions. Example:

```
>var k: table[x,y:integer] with key x;
>
```

Variable declarations entered in this way create temporary variables. They are deleted from the data module when browsing mode is ended unless they are explicitly kept.

5.2.4 Data Input

Data can be entered into tables with the 'enter' command. This command is also used for adding data to a table. The 'enter' command puts the user into data input mode. The prompt character in this mode is '#'. Data input mode is terminated by a single period on a line, or by an end-of-file condition. Example:

```
>enter part;
#   33372   11 12
#   57268    3  0
#.
>
```

Input is free format, each token must be separated by a blank or an end-of-line. Input is read according to the type of the current item to be read. The ?? command can be used to ask for the type of the expected input, and in case of a table the user is also informed about the current attribute name.

An alternate form of the modify statement is also available in browsing mode which allows the user to enter data directly. The grammar rule for this form is given by

```
$    modification = MODIFY ident [accesslist]
```

When this form of the modify statement is used then the user is prompted for input for each attribute. Only those fields are changed for which values are supplied. This mode can be terminated by a single period on a line, or by an end-of-file condition. Example:

```
>modify part[1123];
k.1980 :
k.1981 :  22
>part[1123];
  1123   10   22
>
```

This form of the modify statement can be used also within the for-statement. Before the user is prompted for values for the fields of the tuples the tuple is first identified by its key.

CHAPTER 6

Implementation Issues

The implementation of the modeling system requires a major programming effort. It is important to divide the task into manageable parts which can be developed independently and then later integrated into a system. This modular development has also the advantage that parts of the system can be replaced when necessary. The major components of the system to be developed are:

- the data base management system for storing data and modules
- translators for data definition, data manipulation, and model formulations
- interpreters for carrying out the data manipulations and the data generation for the solution system
- the user interfaces for supporting the browsing facilities, help facilities and editing facilities.

The components of the system are highly interdependent, e.g. the translator requires the services of the data base. It is therefore necessary to isolate the dependencies of the modules into interfaces so that the modules can be developed separately.

The design of the system could be carried out without regard to any programming language. However, it must be recognized that a design is strongly influenced by the actual language chosen and by the environment in which the system is implemented. One of the design goals of the system was also the abilitity to develop a model on perhaps a small machine and then solve the model on a larger machine. This requires that the software must be written in a language which allows it to be easily transported.

Several languages were considered for implementing the modeling system. As a fair amount of 'system' work is required the choice was narrowed down to Fortran and Pascal [Wirth1]. The final choice was Pascal due to its superior data structuring facilities and language constructs. Also experience had been gained in transportablity with a parser generation system [Burger2] written in Pascal. By adhering to a subset of the language this system could be directly moved to a variety of machines.

This chapter describes some of the design considerations and problems in implementing a modeling system. The first section describes the data base management part. The next section deals with the translator. The

last section handles the run-time storage organization and the input generation for the LP solution system.

6.1 Data Base Management System

The model system requires a data base which must be able to store the relational data structures as well as the internal and external formulations. Directories must be kept to find the objects in the data base and to determine how these objects are stored. Furthermore 'internal' structures, such as symbol tables which are created during the translation of formulations must be stored.

The design of the data base proceeds in hierachical levels in a bottom-up fashion. This also corresponds to the actual implementation of the data base subsystem. A similar approach is taken in [Schmidt3].

The lowest level of the system is concerned with providing bulk storage. The next level manages 'objects' in the data base. Objects fall roughly into four categories: data objects, text objects, compilation objects, and system objects. The third level provides object-specific operations, e.g. data manipulation operations for data objects. Finally the last level implements the operations necessary to interface the data base with other components of the modeling system.

The data base storage of the modeling system is realized by two random access files. One file is used to hold 'permanent' objects, the other file is used for temporary objects. Only the file with permanent objects is retained when the modeling system ends. Both files, however, are treated the same way by the routines of the data base management system.

6.1.1 Page Manager

The basic 'external' storage unit available from an operating system is a 'page' which can be accessed in a random fashion. By building on this storage unit all data base operations can be implemented in a machine independent way. The lowest level implements a page manager which makes a page available when it is needed by a higher level. Pages are kept in a page pool in main memory. When a new page is obtained from secondary memory then the least recently used page in the pool is removed, and if it was altered written back to secondary memory.

6.1.2 Objects

There are four basic kinds of objects which are stored in the data base: data objects, text objects, compilation objects and system objects. A directory for example is a system object. An object may be composed of several 'primitive' objects. We will refer to such an object as a 'composite' object. A primitive object consists of one or more pages in the data base. The primitive object is identified by an internal address.

This address is the page number of the first page used for the object. The system maintains a directory of all objects (primitive and composite). This directory is itself an object. (The first entry in this directory contains information about itself). The directory entry of a primitive object records also the last page allocated to the object. This information is used when the object is extended with further pages. Pages belonging to the same object are connected by links (see Figure 6-1). Some words on each page are reserved for organizational information. The links use part of this space. Links are necessary for processing an object sequentially.

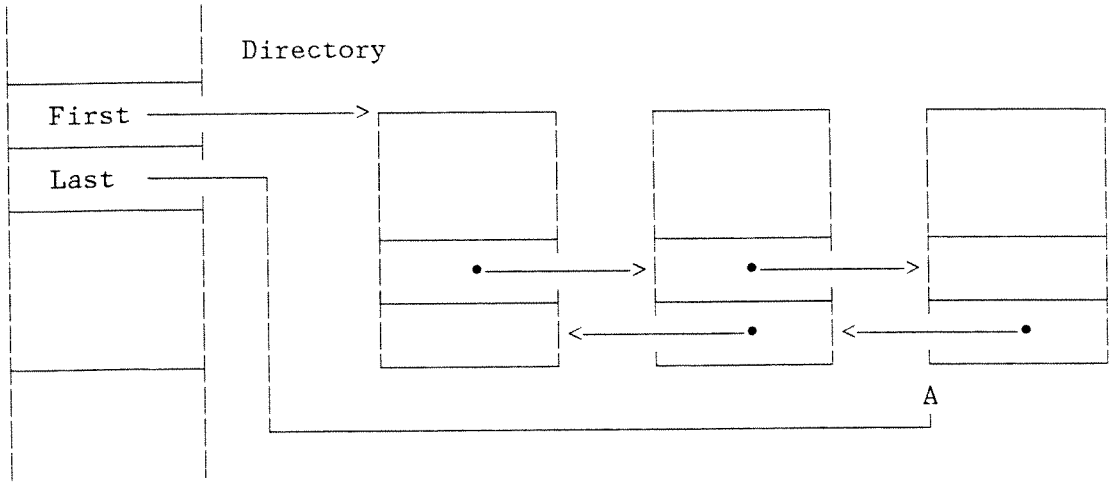

Figure 6-1 A Primitive Object in the Data Base

There are three basic storage structures in which information is stored in the data base: indexed structures, linear structures, and blocked structures. An object is represented by one of these structures. The structure is chosen according to the information content of the object and the particular access requirements. Data for example is usually stored in an indexed structure. The storage structures are described further in the next sections.

Object management is mostly concerned with managing the object directory. An object is created by making the appropriate directory entry. The allocation of space is handled by the access and manipulation operations specific to the storage structure of the object. An object is deleted from the system by removing the directory entry. Any pages which the object occupied are made available for re-use.

To speed up the access to objects a limited number of directory entries is kept in main memory in a directory pool. When a new entry is placed into the pool then the least recently used entry is discarded. If this entry has been modified then the system directory entry is updated.

6.1.3 Data Objects

Data objects implement the table structures of the modeling system. A table structure consists of a collection of tuples which may need to be accessed by key. Other operations to be carried out are insertion, deletion, and modification of tuples in the table structure. Several organizations are discussed in [Ullman] to handle these operations efficiently. The storage organization chosen here for tables is an indexed structure.

An indexed structure consists of an index and the tuple structure (see Figure 6-2). The tuples are stored in sorted order according to their key. The index is also a tuple structure. The tuples consist of the key values of the first tuple on each page and the page number. The index is sorted. It is possible to create several levels of indexes, however, a one level index seems to be adequate for the needs of the modeling system.

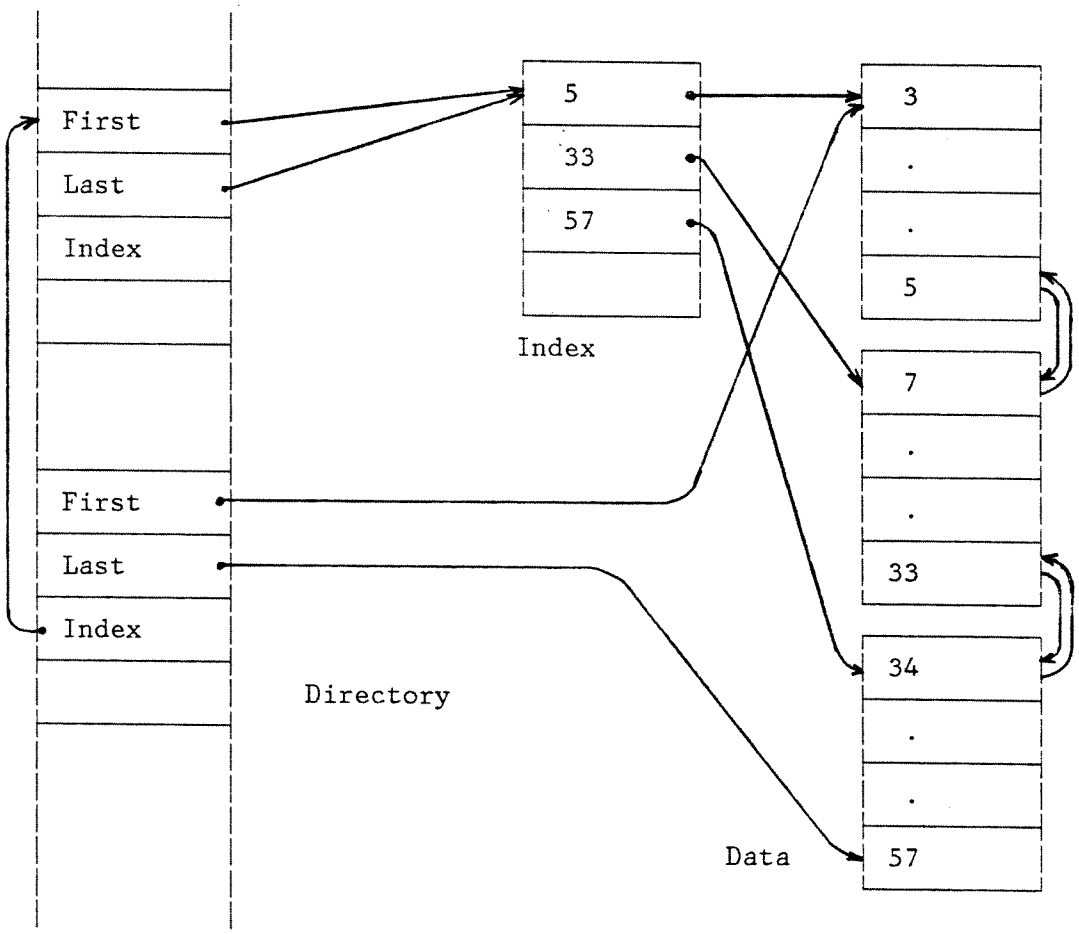

Figure 6-2 Indexed Structure

A tuple access by key proceeds as follows: First the index is searched to find the page on which the tuple may be located. Then this page is searched for the tuple. Page searches are done by binary search as all entries on a page are sorted.

The insert and delete operations of tuples keep the tuple structure as well as the index structure sorted. As the index entries can be considered to be tuples the same insert and delete operations are applicable to both structures. When a tuple is inserted into a page which is already full then this page is split into two pages. Two (logically) adjacent pages are combined when a tuple is deleted and the remainder of the entries on the two pages fits onto one page.

One advantage of the indexed storage structure is the ability to store duplicate values easily. Further all data is accessible sequentially in sorting order. Tables which are not declared with the key property can be stored without an index and the same access and manipulation operations are still applicable. The disadvantage of this index organization is that a tuple in the tuple structure is not associated with a fixed location. Tuples are moved around to maintain the sorting order when tuples are inserted or deleted. This property of the tuple structure must be taken into consideration when operations on tables are implemented. In particular it precludes the use of additional structures which could facilitate the implementation of relational operations between tables. It was assumed, however, that most queries are simple in nature and that no additional access paths to data are required. The resulting data base structure therefore is simple to implement. It would have been also possible to use B-trees [Bayer]. A complication with this organization would have been the handling of duplicate elements. In our case this structure is not necessary as the anticipated update activity is rather small.

6.1.4 Relational Operations

This level in the data base design implements the relational operations. It also constitutes the interface to other components of the model system which manipulate table data. Of importance here are the relation constructor and the for-each clauses of the language. These constructs are transformed into loops which use the access and manipulation operations of the previous level. In particular the join operation is expressed by nested loops.

Several semantic interpretations of the loop construct are possible (see [Aho1]). Here we process the tuples of a table sequentially. The loop mechanism consists of a loop variable (which provides storage for a tuple) and a pointer to the 'next' tuple to be processed. Before the loop body is evaluated the 'current' tuple is retrieved from the table and stored in the loop variable. The organization for handling loops over tables in the data base is shown in Figure 6-3. Here two loops exist over the same table.

There are two particular problems which arise with insert and delete operations within a loop. When a change is made to a table over which the loop ranges then the pointer to the 'next' tuple must be updated as the tuple in the table may have moved within the structure. It may be

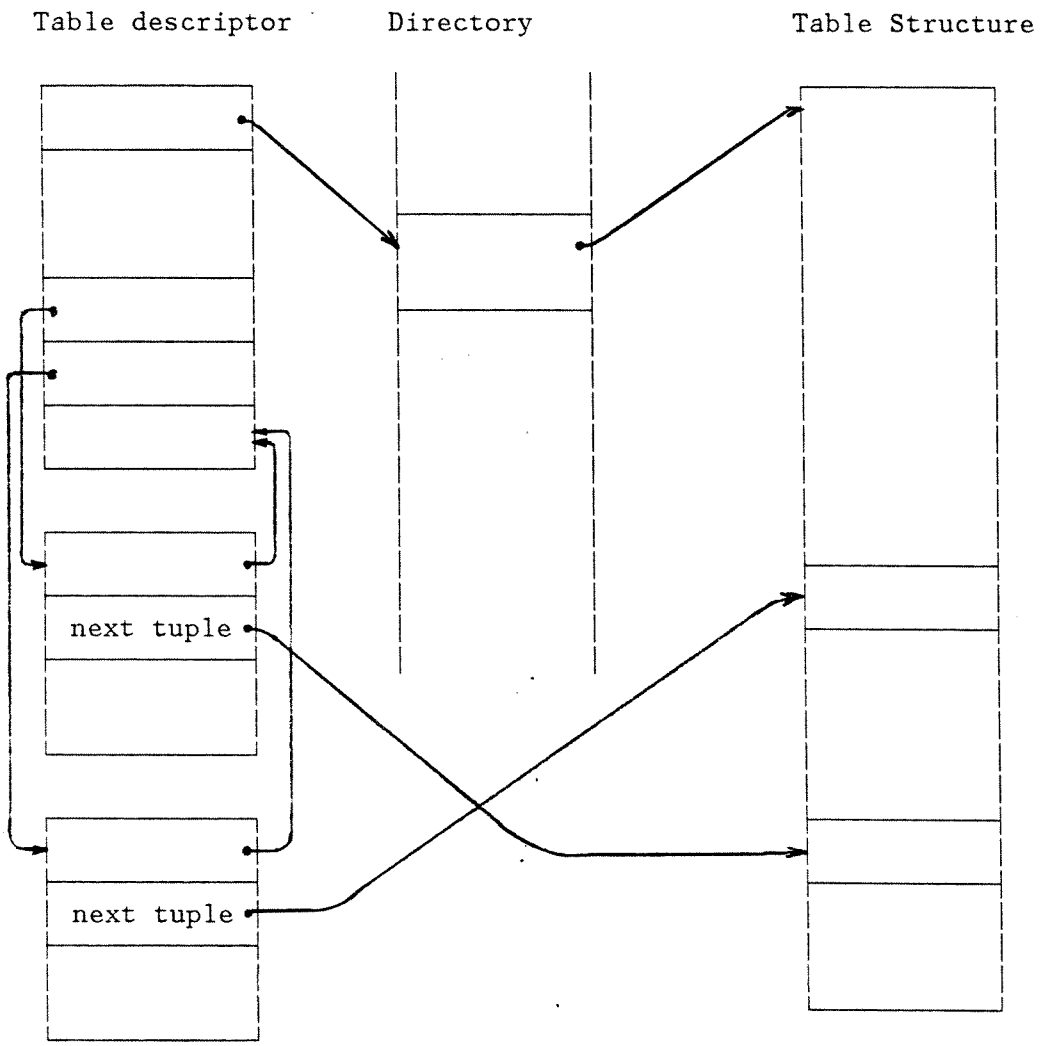

Figure 6-3 Loop Organization

necessary to update several pointers if several loops exist for the table (see Figure 6-3). The second problem is of a semantic nature. In the given implementation tuples are accessed sequentially. The result of a loop can depend on where a tuple is inserted into the table. If the tuple is inserted after the current tuple of the loop then this tuple will be within the range of the loop still to be processed. If this problem is to be avoided then an auxiliary table must be created into which the new tuples are inserted. This table is then combined with the original table.

6.1.5 Data Values

The system is designed so that each component of a tuple occupies one word of memory. Integers and reals can be stored directly. Scalar values and strings, however, must be encoded into integers. Most of the data base operations can be carried out without having to resort to the original representation of these values. Transfer structures are provided which map the external representation into the internal representation and vice versa.

Two tables are created for each scalar type which accomplish the mapping. We refer to them as input and output tables. By restricting the scalar names to a certain length they can be used as a key in the input table to obtain the encoding. When e.g. the scalar values need to printed, then the encoding is used as a key in the output table to obtain the representation of the scalar value.

For strings a more elaborate storage scheme would be necessary as strings might be arbitrarily long. In this implementation, however, they are restricted to a fixed upper bound length, and the same storage mechanism as for scalars can be used. A string type is treated to be universal within a data module and thus two tables per data module are sufficient to handle the string mapping. There is, however, a difference between strings and scalars for comparison operations. Strings themselves are values, and therefore their actual representation must be used for comparison. Scalars are synonyms for their encodings, and thus the encoded value of a scalar is sufficient for these operations.

As string and scalar representations are stored separately from the 'actual' data a problem arises with the deletion of these entries. When a tuple is deleted which contains values belonging to a domain of scalar or string values then the representation of these values could be also deleted if there exist no other tuples which refer to these values. Finding these tuples is an expensive operation as all data structures must be searched which have links to the representation structure. The deletion of any 'unused' entries in the mapping structures is therefore deferred to the time when the data base is re-organized. (Data base re-organization is done externally).

6.1.6 Array Attributes

Array attributes require some special considerations. The array items are stored in a 'linear' storage structure. A linear storage structure is characterized by two properties: Items are not ordered, and each item in the structure has a fixed position. An item can be therefore accessed by a pointer. A pointer consists of the page number on which the item is stored and an offset on the page where the item starts. New items are always added at the end of a linear structure. An item is deleted by setting the appropriate delete bit in the organisational words of the page on which the item appears. This information is checked when a linear

structure must be processed sequentially. The storage occupied by deleted items is reclaimed when the data base is re-organized.

A tuple of a table which contains array attributes is organized as follows. Each field which represents an array attribute contains a pointer to the array of values stored in the respective linear structure. Many data manipulations can be carried out without concern for array attributes.

6.1.7 Text Objects and Compilation Objects

Text objects contain the external formulation of modules. A variety of access criteria to parts of text might be useful e.g. for supporting incremental development or phrase-oriented editing. In the initial design of the modeling system, however, text objects have no internal structure. Text objects are always treated as a whole.

Text is stored as a 'blocked' storage structure. Pages are only artificial subdivisions so that the basic mechanisms of the data base system can be used to store the information. For editing purposes a text object is actually transformed into a file on which the modifications are carried out. This file is then transformed back into a text object, replacing the original text object in the data base.

Compilation objects are produced when an external formulation is translated. The translation of a module usually creates two objects: a symbol table and a code table. Compilation objects are also stored as blocked structures. For access or modification a compilation object is always moved as a whole into main memory (thereby removing the artificial page boundaries).

6.1.8 System Objects

Directories are system objects. Depending on their need for sequential access they are stored as index structures or as linear structures. Two directories are described briefly: the system directory and the module directory.

The system directory of objects is a linear structure. An entry in the system directory contains besides page information other object related information. The entry for a data object for example contains a pointer to the directory entry for its index, the number of tuples stored in the tuple structure, and information on where the fields are in the tuple which form the key.

The module directory is organized as an index structure. The module name is used as key to access the module directory. An entry contains pointers to the objects which constitute the module, e.g. pointers to compilation objects and text objects. Associated with the module directory is a table which records the cross-references between modules.

Other system objects are the objects which are created for the purpose of interfacing with a solution system. These objects may represent text or data. They are stored in the data base as blocked storage structures. They are kept in the data base until a solution is obtained from the solution system and the solution is converted into the representation used by the modeling system.

6.1.9 Further Problems

An implementation of the data base system must deal with two additional problems: data base recovery and data base re-organization. Data base recovery is of practical importance as a crash of the system may destroy the data base. At least some rudimentary mechanisms must be provided to keep a data base usable. Data base re-organization on the other hand is not necessary for the operation of the modeling system. However, it may be used to improve the performance of the data base.

When a fatal error occurs in running the modeling system then information contained e.g. in a directory entry may have been updated in memory but not yet written out to the data base thus making the information in the data base inconsistent. A crash can also be induced by the user when a run of the model system is aborted. There are two ways of handling this situation depending on the severety of the crash. The first one is to go back to a previous version of the data base and to re-apply the modifications made so far. This requires that the sequence of modifications is kept. The second one is to try to crash 'gracefully' and to keep the data base information consistent.

The second method is only applicable in circumstances where it can be assured that the crash did not occur during the modification of the data base or a data base related structure. We may then assume that the part of the data base which is kept in memory is still consistent with the data base. After control has been regained from the operating system modified pages in the page pool and directory entries in the directory pool are written back to secondary memory. In addition certain environment information, e.g. the symbol table of the current module being processed (which is resident in memory and which also could have been altered) is stored back into the data base.

Data base re-organization can be handled by a utility program outside the modeling system. Certain operations do not delete information in the data base but simply make the information inaccessible. This space can be reclaimed by re-organizing the data structures concerned. Based on statistics gathered on access activities of certain data objects the utility could be also used to define additional levels of indexes, or used to transform a linear structure into an indexed structure.

6.2 The Translator

The translator transforms the various formulations into symbol table information and/or code for the interpreter. The translation process consists of four phases: lexical analysis, syntactic analysis, semantic analysis, and code generation. In addition an error handler is provided which reports the errors which might have been encountered in any of the phases.

The translator is called in several contexts: during command processing, in browsing mode, and in the translation of formulation modules. Depending on the context the translator accepts input from the terminal, from a file, or from a text object in the data base. When input is interactive then the translation is terminated whenever an error is encountered. The user is returned to a state in which he can start over with the translation. When input is not interactive then the whole file or text object is always processed.

There are two features special to this translator which distinguish it e.g. from a translator for a programming language. One is the need to handle entities which are stored in the data base. The other is the need to generate code which combines data manipulation with symbol manipulation. With this in mind we describe in the following the symbol table structure, the storage administration, and the handling of data base objects referring to the run-time environment if necessary. Also the translation of some language constructs is described, in particular the translation of equations.

6.2.1 Implementation Method

The lexical analyser and the syntax analyser are generated with the aid of the XBSW parser generator system [Burger2]. This system accepts an LALR(1) grammar [Aho2] definition of the language as input and produces from it lexical scanner tables and parse tables. These tables are then used in the scanning and parsing algorithms of the analysers. One significant advantage of using a parser generator is the increased reliablity. A mechanically-generated parser is more likely to be correct than one produced by hand.

The grammar rules are associated with 'semantic actions'. This allows a syntax-oriented translation scheme where the code generation and symbol table manipulaton can be directly related with the syntactic structure of the language. Semantic actions are calls to routines which perform the required operations. These routines are supplied by the implementor.

The translator employs a semantic stack which is used for storing properties about the constructs being translated. This information together with the information stored in the symbol table is used for semantic analysis, e.g. for checking type compatibility between operands and operators. Other information is also stored on the stack which is needed for code generation.

Code is generated for a stack machine in the spirit of the Pascal-P compiler [Nori]. Data manipulation operations are translated into postfix sequences. These sequences can be evaluated by the interpreter with the help of a run-time stack. An instruction consists of an operator and of up to two operands. Expressions which involve symbolic operations are translated into tree structures. Here an instruction consists of an operator and up to two links. An expression is evaluated by walking through the tree using the stack if necessary to keep track of the branches. Operators themselves are interpreted according to their context.

6.2.2 Symbol Table

The symbol table contains the names and structure descriptions of named entities (e.g. variables) found in a formulation module. Symbol tables are created for all modules. A symbol table may consist of three structures: the name table, the scope table, and the string table. The name table contains information about an entity such as name, type, kind, structure, etc. The scope table is used to partition the name space of a symbol table. With the help of this table the translator manages the visibility of names in any given context. The string table is used for storing string constants encountered in the text of a module.

While the information described so far serves mainly for semantic analysis additional information for code generation is required. In particular addresses must be assigned to entities for which storage will be allocated at run-time. The storage administration is described in the next section.

An entry in the symbol table can be also directly associated with an object in the data base. A pointer to the system directory entry of the data object is then stored in the symbol table. Table constants are handled in this way. Of interest here is the symbol table of a data module: the symbol table acts also as directory for the data objects stored in the data module.

6.2.3 Storage Administration

The modeling language is essentially a block-structured language. A dynamic storage allocation mechanism is used for providing storage for blocks. Whenever a block is entered a so-called stack frame is created which contains space for the entities of the block and some organizational information. This stack frame is de-allocated when the block is terminated. Relation constructors, routines, equations, problem definitions, and modules themselves are blocks in this sense.

Code for accessing entities is generated according to the stack frame concept. Frames are accessed by base addresses. Base addresses of frames are managed at run-time by a display (see Section 6.3.1). The entities within a frame are addressed relative to the beginning of the frame.

These addresses are stored in the symbol table together with other relevant information of the entities.

There are three kinds of entities for which space is provided in a frame: parameters, variables, and descriptor blocks. Descriptor blocks are entities which are associated with data base objects. Space for variables and parameters is allocated in the usual way of Pascal-like languages. Reference parameters need space for holding an address. Variables (not associated with data base objects) and value type parameters need space according to their type. A tuple for example requires several storage units.

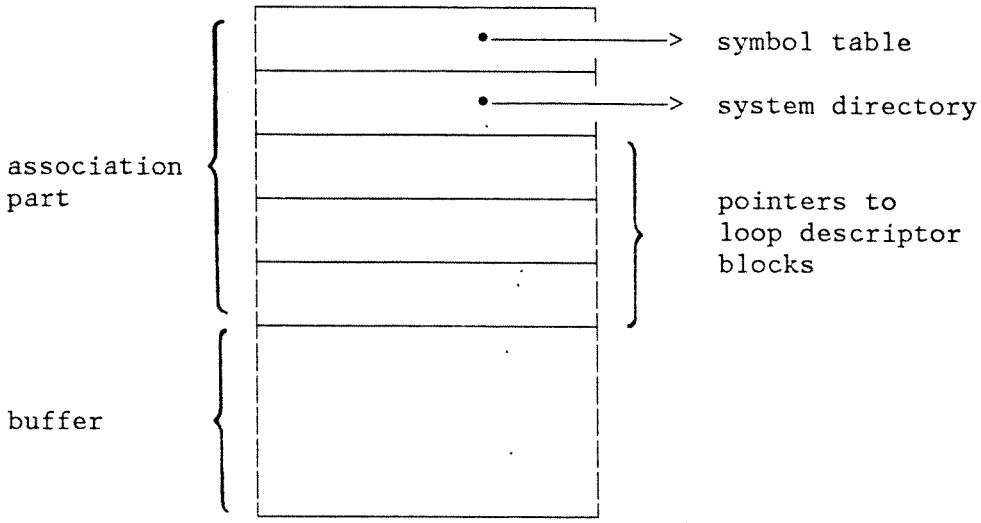

Figure 6-4 Variable Descriptor Block

A descriptor block consists of an association part and a buffer. The buffer is used to hold an element of the associated data base object. Buffers are treated like local variables with respect to code generation. There are two kinds of descriptor blocks: variable descriptor blocks and loop descriptor blocks. The association part of a variable descriptor block is initialized at run-time either when the associated data object is created, or in the case of an execution module when the data object from the data base is bound to a variable of the execution module. The association part contains pointers to the symbol table, the system directory, and pointers to loops which may be in existence for the data object (see Figure 6-4).

The association part of a loop descriptor block is initialized when a loop is started. It contains a pointer to a variable descriptor block of the data object over which the loop ranges, a pointer to the next tuple which is to be accessed, and the termination condition of the loop (see Figure 6-5). The buffer always holds the 'current' tuple.

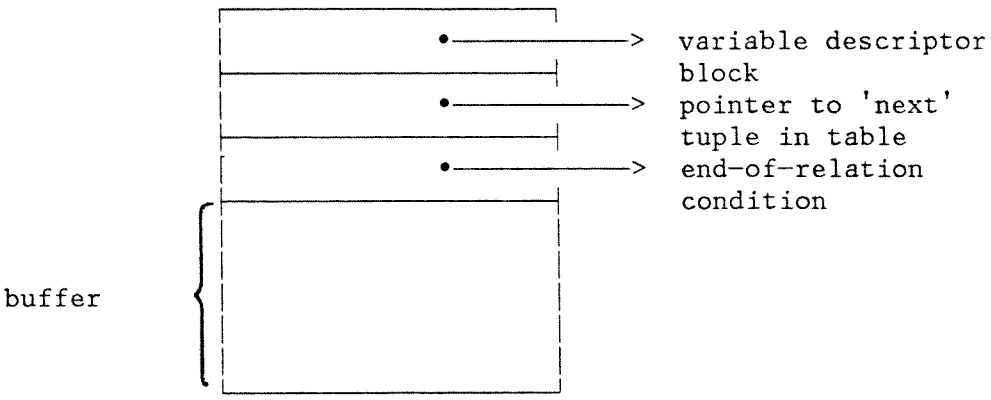

Figure 6-5 Loop Descriptor Block

6.2.4 Equations

Equations are transformed into 'special' procedures which when called generate part of the input to a solution system. The generation of the solution system input takes place when the solve-statement of a problem section (see Section 4.1.3) is executed. The generation combines data manipulation operations with symbol manipulation operations: model 'variables' are transformed into names, whereas 'parameters' reference the data base to obtain their values. For an LP solution system this step represents the matrix generation. The equations generate row and column names as well as the matrix elements.

Equations are translated into expression trees. The tree structure is chosen for several reasons. First the structure of an expression must be preserved as it is not possible to generate code for arithmetic operations. The operands involved may be parameters or variables (in the modeling sense), and this information is only supplied when the equation is called. Second tree structures can be easily re-arranged if e.g. an expression must be re-ordered. Finally the same tree building mechanisms would be applicable for the translation of non-linear equations. Figure 6-6 shows the tree built for a simple equation.

A variable in the expression tree is represented by its symbol table address. If the variable is a model variable at run-time then the symbol table address is used in the name generation for the solution system. If the variable is a model parameter at run-time then the descriptor block of the variable in the stack frame is accessed to obtain the value. If the operands of an arithmetic operation are values then the operation is carried out with these values, otherwise some part of the solution system input is generated. Mapping tables are established for name generation when a problem section is executed. Name generation as well as the generation of LP input is described further in Section 6.3.2.

Also of interest is the handling of implicit variables. They are defined in the solve statement of a problem section. The solution system interface provides a symbol table of 'standard' names and their structures which are special to the solution system which is going to be used for

Equation

```
each i in I::
sigma (j in J:: x[i,j].k*a[i,j].n) <= b[i].m
```

Tree Structure

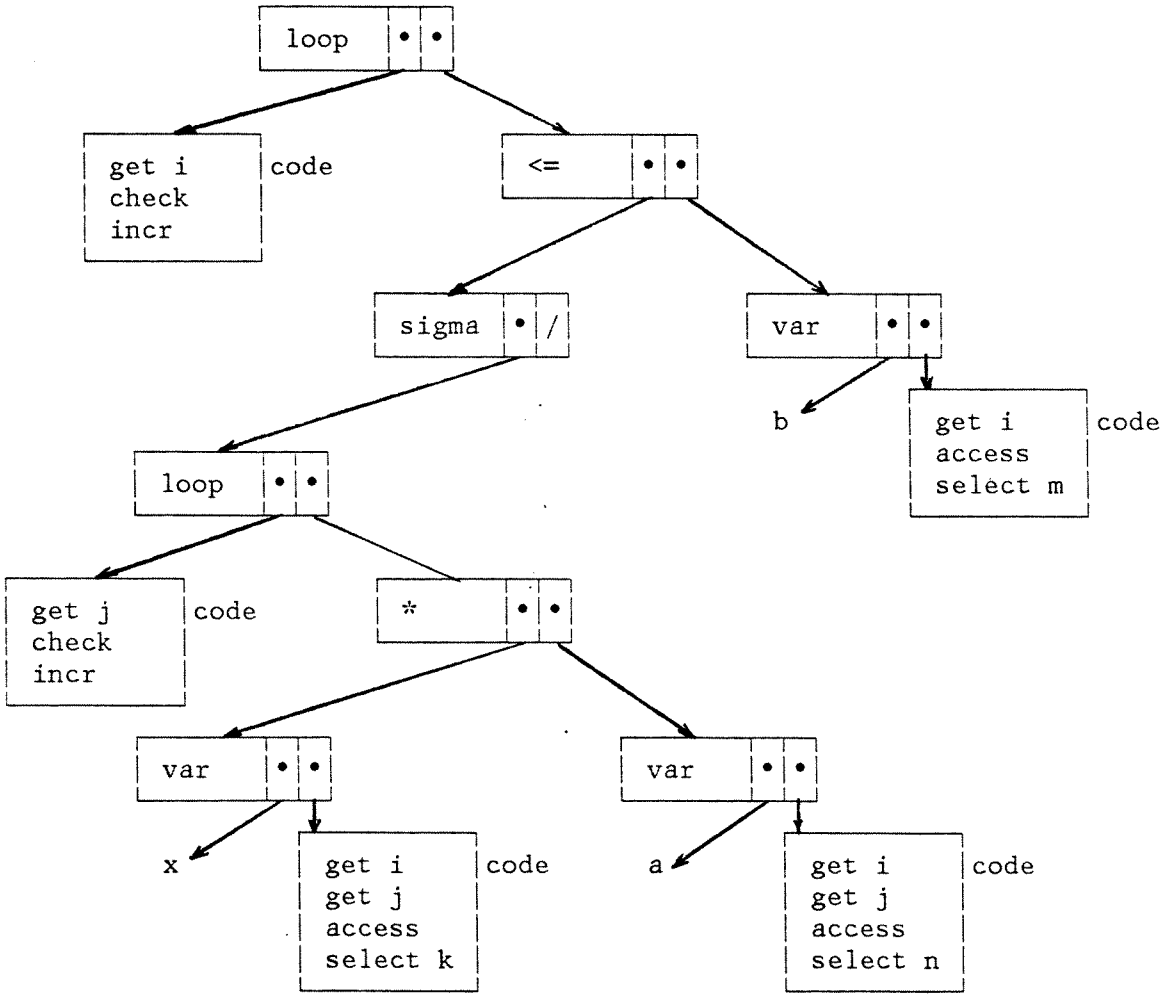

Figure 6-6 Tree structure for a simple equation

solving the model. The information of this table is added to the symbol table when the solve statement is processed.

6.2.5 Other Features

In this section we handle interface specifications and use- and include clauses.

The include clause adds the type and variable declarations of a declaration module to the symbol table of the current module. The name of a type or variable already declared in the module may appear also in the declaration module if name and structure are exactly the same.

The use clause adds the names of routines and their parameter declarations of the referenced routine module to the symbol table of the current module. Either all routines or only those selected by the import list are added. Type declarations for routine parameters must be defined in the current module. These types are checked for compatibility with the types as defined in the routine module.

Interface specifications of model modules and of execution modules are syntactically similar but they are handled differently. The interface specification which appears in the problem definition of a model module is a formal parameter definition. It is handled in a similar fashion as the formal parameter definition of a routine. At translation time, however, it is not known which variables of the model module are local variables or which ones are formal parameters. For translation purposes all variables are considered to be formal parameters. At run-time two stack frames are allocated, one for their role as parameters, and one for their role as variables. The variables of the variable stack frame are taken by default as the actual parameters of the model module. This parameter assignment is overwritten by the actual parameters supplied by the execute statement according to the definition of the problem which is being executed.

The interface specification of an execution module on the other hand is a binding and intialization mechanism. All variables of the execution module are local variables. The run command initializes the respective variables of the execution module with values, or binds them to the data structures of the data module with which the execution module is executed.

6.3 The Interpreter

The interpreter executes the code generated by the translator. The interpreter is called by the 'run' command. It is also called in browsing mode to handle the data manipulations. This section deals only with two aspects of the interpreter which are of interest: the run-time storage organization and the LP system input generation.

6.3.1 Run-Time Storage Organization

The interpreter can be thought of as a simulator of a high-level machine specifically tailored to the needs of the modeling language. This machine provides three storage areas and several registers. Two of the storage areas contain code segments and symbol tables respectively. Access to code segments and symbol tables (except to the symbol table of a data module) is read-only. The third storage area is organized as run-time stack. The registers contain pointers to the code (program counter), top of stack, and current active stack frames (the display). In the initial design of the code generator no relocatable code is produced for the interpreter. Additional mapping registers are therefore provided which are used to address the appropriate code segment or symbol table.

Before the interpreter starts, the code segments and symbol tables of the modules needed for the execution are obtained from the data base and loaded into the storage areas. In browsing mode this step is done when the 'query' command is issued. Here the symbol table of the data module is loaded. The 'run' command loads the execution module, possibly a model module and several routine modules. In addition the symbol table of the respective data module is loaded so that the variables of the execution module can be bound to the data objects in the data base.

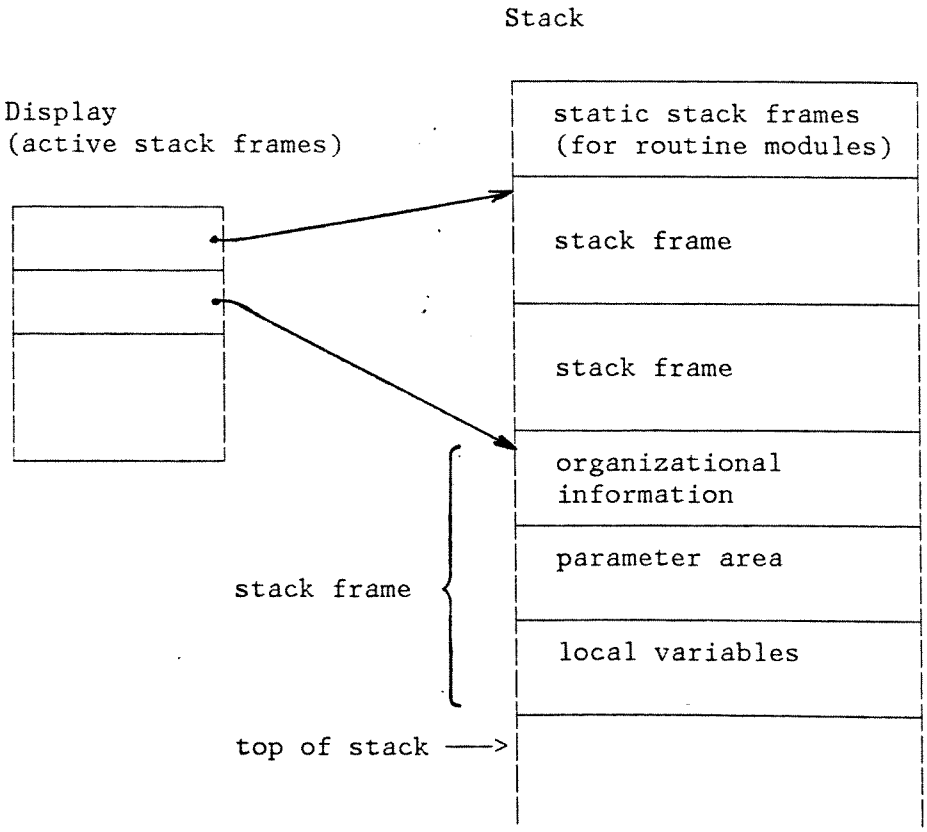

Figure 6-7 Run-Time Stack

Next the run-time stack is initialized. If routine modules with global variables are present then stack frames for these modules are allocated. The stack frame for the execution module is allocated and the interface variables are initialized. The information for variables associated with data base objects is obtained from the symbol table of the data module. This step checks also the type compatibility between the variables declared in the data module and in the execution module. From then on stack frames are allocated and de-allocated by the interpreter in a dynamic manner (updating the display accordingly). Figure 6-7 shows the run-time stack structure.

In browsing mode the run-time stack initialization is similar. An execution-module like stack frame is built up when a data manipulation operation is translated. This stack frame is allocated and initialized with the data objects in the data base. The symbol table of the data module is modified when new types and variables are declared, or when a new variable is created through a data manipulation operation.

6.3.2 LP System Input Generation

LP system input is generated when the solve-statement of a problem section is executed. Each equation called generates some parts of the LP input. These parts are then assembled to form the standard LP input of the system in use.

Row names and column names are obtained as follows. The symbol table address of an equation together with the index values of the outermost loops of the equation represents a row name. The symbol table address of a variable together with the symbol table address of the field name and the index values represents a column name. These representations are mapped into actual names which are used in the input formulation of the LP system. Two tables are established for this purpose when the problem section of a model module is executed. (These mapping tables are used again when the results of the LP solution are stored into the respective data objects of the data base).

The following example demonstrates what input is generated for the LP system by a simple equation.

```
constraint is
each i in I::
   sigma (j in J:: amount[i,j].x) <= capacity[i].k;
```

We make the following assumptions. The model variable is amount[i,j].x. The set I contains 2 elements and the set J contains 3 elements. The 'capacity' table in the data base contains the values 5.0 and 10.0. The equation call then generates 2 row names, 2 right hand sides, and 6 matrix elements. The row names and column names are mapped as described above. The equation represented by (constraint,1) for example is mapped into the row name R00001. The model variable amount[1,1].x may be represented by

(amount,x,1,1). It is mapped into the column name C00001. The equation above then generates the following three parts of the LP input:

```
ROW ID
       R00001  L
       R00002  L
MATRIX
       C00001  R00001    1.0
       C00001  R00002    1.0
       C00002  R00001    1.0
       C00002  R00002    1.0
       C00003  R00001    1.0
       C00003  R00002    1.0
RHSIDE
       R00001             5.0
       R00002            10.0
```

Letters in the row identification section indicate the kind of equation. The letters are L, G, R, or E which denote less-than, greater-than, range, or equality equations respectively.

Additional parts of LP input will be generated by other kinds of equations, e.g. a bound equation generates a bound section.

6.4 Implementation Stage

Parts of the modeling system have been implemented on CDC Cyber and DEC-10 machines. The user interface for command processing and for browsing mode is operational. Data declaration and data manipulation constructs can be processed and executed in browsing mode. The lower levels of the data base management system as well as a restricted relation constructor have been implemented. Formulation modules can be parsed and stored in the data base. Work is continuing on the translator for equations, and on the solution system interface.

CHAPTER 7

Conclusion

7.1 Highlights of the Thesis

We developed a system for mathematical programming which approaches the problem of model formulation and model data management in a new and different way by bringing ideas from programming language design and data base design to the area of operations research. We identified the problems in model development and introduced the module mechanism to break the modeling task into subtasks that can be handled separately. In particular model formulation and model data management are treated independently.

We separated further the model formulation into two parts: equation formulation and problem formulation. Algebraic notation is used to express the model equations. This notation is more natural to a model builder than for example the notation required for matrix generation in programming LP models in commercial systems. A set of equations may be used to represent similar and related models. The problem definition selects the relevant equations for a particular model. This mechanism allows us to test and examine different aspects of the real-world problem which is being modelled.

We introduced data handling facilities based on the relational data model to deal with the important problem of model data management. The relation constructor was defined for expressing simple queries in a non-procedural form. The data definition and data manipulation language was integrated with the model formulation language to share a common notation for data declarations and data operations.

We provided an interactive environment for the modeling system to support the design process of a model and to make it possible to explore alternative solutions of a problem. To this end a command language was defined with which different modes of man-machine interaction can be initiated. The browsing mode is of significance here for exploring alternative solutions. In this mode the user can apply the data definition and data manipulation constructs interactively. This can be used to change or create data for a different solution; it may be also used to learn more about the data at hand. Results can be obtained in a short period of time thus making the modeling system a valuable tool for decision making.

Finally we considered the design and implementation issues of a modeling system. The three major parts of the system we dealt with were the data base, the translator, and the interpreter. We described the data

base structures which are necessary to store model data and model formulations in the data base, and which can support the relational data operations of the modeling language. We assumed that the data definition, data manipulation, and model formulation constructs are translated for a stack machine. We paid some special attention to the translation of equations and the handling of data base objects. We further described the run-time organization of the interpreter and the generation of input for a mathematical programming solution system.

7.2 Future Research

We developed a modeling system that can handle problems which fall into the class of well-structured problems expressible by mathematical programming. Further research is required to extend the modeling system for models which deal with less structured problems, and which require for their solution the active participation of the user of the model system. Models of this kind may be solved by simulations or by a combination of optimization and simulation [Wolters]. Research is also needed to develop methods for analyzing the model formulations. Special properties of a model could be exploited to select the best solution method.

More research is warranted in the area of man-machine communication. A suggestion mechanism which guides the user through model design and model solution could be added to the user interface. This facility would be valuable to the experienced user as well as to the novice. It will be important when the problems to be handled by the modeling system become more complex. Ideas on a suggestion mechanism in the context of a program design system were developed in [Moriconi].

The data analysis capabilities of the modeling system is another area of interest. These capabilities could be greatly enhanced by providing interface facilities for statistical analysis systems. Extending the system in this direction would require a re-evaluation of the current design of the interfaces to application packages in general. The interface design could be generalized to include also interfaces to external data base systems. Preliminary work on the integration between data base systems and applications has been reported in [Mitchell].

The most ideal implementation of the modeling system would consist of some loosely coupled self-contained programs. Current operating systems have difficulties to support such a concept. The notable exception is the UNIX operating system [Ritchie] which provides interface handling for program modules. More complex programs can be built up by connecting input/output of the existing programs. A research effort must be undertaken to identify the problems of sharing and composing software parts for the development of large software systems. In particular operating systems must be developed which can support the integration of software.

Syntax Summary and Index

```
 1 TypeDeclaration = SimpleTypedecl | StrictTypedecl |
 2                   StructTypedecl
 3 SimpleTypedecl = TYPE ident "=" basicType
 4 StrictTypedecl = DOMAIN ident "=" strictType
 5 StructTypedecl = TYPE ident "=" structType
 6 basicType = standardType ["[" range "]"]
 7 standardType = INTEGER | REAL | BOOLEAN | STRING
 8 range = rangeConst ".." rangeConst | POSITIVE | NEGATIVE
 9 rangeConst = ["+"|"-"] (number | INF)
10 strictType = (basicType | enumeration | externaldefined)
11              [constraint]
12 enumeration = "(" ident {"," ident} ")"
13 externaldefined = ENUMERATED | SUBRANGE
14 constraint = ORDERED | UNORDERED | RESTRICTED
15 structType = tupleType | tableType | setType
16 tupledef = "[" attributeSequence "]"
17 tupleType = TUPLE tupledef
18 attributeSequence = attribute {";" attribute}
19 attribute = ident [extension] |
20             ident {"," ident} [extension] ":" simpleType
21 simpleType = ident | basicType
22 extension = "<" ident ">"
23 tableType = TABLE columndef [WITH (keydef | DUPLICATES)]
24 columndef = ident | tupledef
25 keydef = KEY ident {"," ident}
26 setType = SET simpleType
27 VariableDeclaration = VAR ident {"," ident} ":" typedef
28 typedef = ident | basicType | structType | derivedType
29 derivedType = TYPE ident ["." ident] | TUPLE ident
30 primary = designator | number | sconst | tconst |
31           StructConst | FunctionCall | RelationConstructor
32 factor = primary | "(" expression ")" | N-operator factor
33 term = factor {M-operator factor}
34 simpleExpr = ["+"|"-"] term {A-operator term}
35 expression = simpleExpr [R-operator simpleExpr]
36 N-operator = NOT | TAKE
37 M-operator = "*" | "/" | DIV | MOD | AND
38 A-operator = "+" | "-" | OR | UNION | INTERSECT | MINUS
39 R-operator = "=" | "<>" | "<" | ">" | "<=" | ">=" | IN |
40              CONTAINS | PARTOF
41 designator = ident [accessList] [fieldSelector] [property]
42 accessList = "[" expression {"," expression} "]"
```

```
43 fieldSelector = "." ident ["[" expression "]"]
44 property = "'" ident ["[" expression "]"]
45 sconst = string | TRUE | FALSE | UNKNOWN
46 slconst = ident | ["+"|"-"] number | sconst
47 tconst = typeop slconst
48 StructConst = [typeop] (SET setstruct | TUPLE tuplestruct |
49              TableConst | EMPTY)
50 setstruct = "[" setElement {"," setElement} "]" | "[" "]"
51 setElement = expression [".." expression]
52 tuplestruct = "[" tupleElement {"," tupleElement} "]"
53 tupleElement = [expression]
54 typeop = "$" typeref ":"
55 typeref = ident | basicType | derivedType
56 RelationConstructor = "{" resultPart rc-op inputPart
57                       [selectionPart] "}"
58 rc-op = "|" | "||"
59 resultPart = telement {"," telement} [WITH keydef]
60 telement = tident | ident "=" tident
61 tident = ident ["." ident]
62 inputPart = itable [BY bySelector] |
63             itable "," itable "(" tident J-operator tident ")"
64 itable = [ident IN] ident
65 J-operator = "=" | "<>" | "<" | ">" | "<=" | ">="
66 bySelector = ident ["[" expression "]"]
67 selectionPart = ":" expression
68 FunctionCall = ident "(" [ActualParameters] ")"
69 ActualParameters = expression {"," expression}
70 statement = [assignment | insertion | deletion |
71              ProcedureCall | IfStatement |
72              LoopStatement | ExitStatement |
73              ForStatement | WithStatement |
74              Modification | InputOutput |
75              ExecuteStatement]
76 statementlist = statement {";" statement}
77 assignment = reference ":=" expression
78 reference = ident ["." ident ["[" expression "]"] ]
79 insertion = INSERT expression INTO ident
80 deletion = DELETE [ALL] expression [FROM ident]
81 ProcedureCall = ident ["(" ActualParameters ")"]
82 IfStatement = IF expression THEN statementlist
83               [ELSE statementlist] END
84 LoopStatement = LOOP statementlist END
85 ExitStatement = EXIT WHEN expression
86 ForStatement = FOR ForSelection DO statementlist END
87 ForSelection = selection [":" expression]
88 selection = [ident IN] ident [BY bySelector] |
89             ident IN basicType
90 WithStatement = WITH ident DO statementlist END
91 Modification = MODIFY ident [accessList] ( WITH | DO )
92                statementlist END
93 ModelModule = MM-header M-declarations M-equations
94               problemsection END
```

```
 95 MM-header = MODEL MODULE ident ";"
 96 M-declarations = {D-declaration} {R-declaration}
 97 M-equations = EQUATIONS M-equation {M-equation}
 98 M-equation = equationHeader IS equation ";"
 99 equationHeader = ident ["(" Q-formal {";" Q-formal}")"]
100 Q-formal = ident {"," ident} ":" typeref
101 equation = [EACH eachpart] Q-leftpart Q-rightpart
102 eachpart = Q-selection {"," Q-selection} ":"
103            [expression] ":"
104 Q-selection = [ident IN] ident | ident IN basicType
105 summation = SIGMA "(" eachpart Q-expression ")"
106 Q-primary = designator | number
107 Q-factor = Q-primary | summation | "(" Q-expression ")"
108 Q-term = Q-factor {QM-op Q-factor}
109 Q-expression = ["+"|"-"] Q-term {QA-op Q-term}
110 QM-op = "*" | "/" | MOD | DIV
111 QA-op = "+" | "-"
112 sQ-primary = ["+"|"-"] Q-primary
113 Q-leftpart = Q-expression
114 Q-rightpart = Q-op sQ-primary | OBJECTIVE |
115              RANGE sQ-primary "," sQ-primary
116 Q-op = EQ | LEQ | GEQ | = | <= | >=
117 problemsection = problem {problem}
118 problem = problemHeader {TV-declaration} [interfaceSpec]
119          [statementlist] SolveStatement [statementlist]
120          END
121 problemHeader = PROBLEM ident ";"
122 interfaceSpec = INTERFACE ["*"] ident {"," ["*"] ident} ";"
123 SolveStatement = SOLVE SS-interface actualmodel END
124 actualmodel = variables objective constraints [bounds]
125 variables = VARIABLES ":" v-ident {"," v-ident } ";"
126 v-ident = ident ["." ident]
127 objective = OBJECTIVE ":" option equationcall
128 option = MINIMIZE | MAXIMIZE
129 constraints = CONSTRAINTS ":" equationcall {equationcall}
130 bounds = BOUNDS ":" equationcall {equationcall}
131 equationcall = ident ["(" expression {"," expression}")"] ";"
132 SS-interface = "(" ident ":" SS-parameter
133               {"," SS-parameter} ")"
134 SS-parameter = ident ["=" expression]
135 DeclarationModule = DM-header {D-declaration} END
136 DM-header = DECLARATION MODULE ident ";"
137 TV-declaration = TypeDeclaration ";" |
138                  VariableDeclaration ";"
139 D-declaration = TV-declaration |
140                 INCLUDE ident {"," ident} ";"
141 RoutineModule = RM-header {D-declaration} {R-declaration}
142                 END
143 RM-header = ROUTINE MODULE ident ";"
144 R-declaration = routineheader {TV-declaration}
145                 BEGIN statementlist END ";" |
146                 use-declaration ";"
```

```
147 routineheader = PROCEDURE ident [formalPart] ";" |
148                 FUNCTION ident [formalPart] ":" typeref ";"
149 formalPart = "(" parameterDeclaration
150              {";" parameterDeclaration} ")"
151 parameterDeclaration = [mode] ident {"," ident} ":" typeref
152 mode = VALUE | CONST
153 use-declaration = USE ident [IMPORT ident {"," ident}]
154 ExecutionModule = EM-header {D-declaration} [interfaceSpec]
155                   {R-declaration} statementlist END
156 EM-header = EXECUTION MODULE ident ";"
157 ExecuteStatement = EXECUTE ident OF ident sharedVariables
158 sharedVariables = "(" ident {"," ident} ")"
159 InputOutput = WRITE "(" expression {"," expression} ")" |
160               WRITEF "(" expression {"," expression } ")" |
161               READ "(" [sentinel ","] reference [sentinel]
162               {"," reference [sentinel]} ")" |
163               READF "(" expression [sentinel] "," reference
164               [sentinel] {reference [sentinel]} ")"
165 sentinel = ":" expression
166 DataDeclModule = DDM-header D-declaration {D-declaration}
167                  [INITIALIZE statementlist] END
168 DDM-header = DATA DECLARATION MODULE ident ";"
169 TranslationUnit = module ";" { module ";"}
170 module = ModelModule | DeclarationModule | RoutineModule |
171          ExecutionModule | DataDeclModule | CommandModule
172 CommandModule = CM-header commands END
173 CM-header = COMMAND MODULE ident ";"
```

```
A-operator              34   -38
accessList              41   -42    91
actualmodel            123  -124
ActualParameters        68   -69    81
assignment              70   -77
attribute               18    18   -19
attributeSequence       16   -18
basicType                3    -6    10    21    28    55    89
                       104
bounds                 124  -130
bySelector              62   -66    88
CM-header              172  -173
columndef               23   -24
CommandModule          171  -172
constraint              11   -14
constraints            124  -129
D-declaration           96   135  -139   141   154   166   166
DataDeclModule        -166   171
DDM-header             166  -168
DeclarationModule      -135   170
deletion                70   -80
derivedType             28   -29    55
designator              30   -41   106
DM-header              135  -136
```

APPENDIX B

Transportation Model Example

This appendix contains an example session for the formulation and solution of a transportation problem. Though the model chosen is rather simple we still can demonstrate with it the important features of the modeling system.

First we specify the transportation model in mathematical notation. We introduce the symbols used, and then define the model equations. The transportation model is stated as follows. Goods are shipped to markets. The constraints are the capacities of the plants where the goods are manufactured, and the demands of the markets (equation 1 and 2). The cost of shipping is to be minimized (equation 4). All shipments must be positive (equation 3). This simple transportation model is then extended with two additional constraints: A minimum of goods must be shipped from a plant (equation 1'). Further the transportation capacity between a plant and a market may be limited (equation 3').

The transportation model is then handled with the modeling system. We describe a scenario which consists of three major parts: the definition of the transportation model in the language of the modeling system, the creation of a data module, and the solution of the transportation problem with the given data. The session was created for the most part with the available front-end of the modeling system. Annotations were added in the form of comments.

The transportation model is defined by three modules. The first module is a declaration module which corresponds to the symbol definition of the mathematical formulation of the model. The second module is the model module which contains the model equations and two problem sections. The first problem section defines the simple transportation problem, the second problem section defines the extended transportation problem. The third module is an execution module which defines the operational aspects of the model solution. It contains two execute statements for each of the problems of the transportation module. The data interface of the execution module uses tables which are different from those which are used for the problem execution. This allows us to demonstrate some of the data manipulation statements.

The data module is created interactively. The user defines data structures for information on plants, markets, and shipping. This is followed by the entering of data.

Solutions for the transportation problems are obtained by running the execution module with the data module just created. The values for the model variables of the optimal solutions are stored in the shipping information table. The optimal solution for the simple transportation problem and the extended transportation problem are then displayed side by side.

The Specification of a Transportation Model

Symbol	Definition
i	plant i in the set of plants I
j	market j in the set of markets J
k_i	capacity of plant i
r_j	requirements of market j
m_i	minimum shipment from plant i
x_{ij}	amount of goods shipped from i to j
c_{ij}	cost of a unit shipped from i to j
l_{ij}	limit in shipment from i to j

Equations

(1) $$\sum_{j \varepsilon J} x_{ij} \leq k_i \qquad i \varepsilon I$$

(2) $$\sum_{i \varepsilon I} x_{ij} \geq r_j \qquad j \varepsilon J$$

(3) $$x_{ij} \geq 0 \qquad i \varepsilon I, j \varepsilon J$$

(4) Minimize

$$\sum_{i \varepsilon I} \sum_{j \varepsilon J} c_{ij} x_{ij}$$

Extensions

$$(1') \quad m_i \leq \sum_{j \in J} x_{ij} \leq k_i \qquad i \in I$$

$$(3') \quad 0 \leq x_{ij} \leq l_{ij} \qquad i \in I, j \in J$$

Sample Session

```
Modeling System (Vers. 1)
type ? for help

-> ?
available commands:
   BROWSE
   CATALOG
   CREATE
   PROCESS
   QUIT
   REMOVE
   RUN
   SHOW
   OPTION
   ?
-> process "example"
                    -- the transportation model modules and the
                    -- execution module are read from a file

processing file example

    1   declaration module tpdecl;
    2
    3     var capacity:     table [plant; k:real]
    4                            with key plant;
    5     var requirement:  table [market; r:real]
    6                            with key market;
    7     var cost:         table [plant; market; c:real]
    8                            with key plant, market;
    9     var amount:       table [plant; market; x:real]
   10                            with key plant, market;
   11     var mshipment:    table [plant; m:real]
   12                            with key plant;
   13     var tplimit:      table [plant; market; t:real]
   14                            with key plant, market;
   15
   16   end;
   17
   18   model module transportation;
```

```
19
20      --declarations
21      include tpdecl;
22      var ii: set plant;
23      var jj: set market;
24
25      equations
26        -- a plant cannot ship more than it produces
27        cpconstraint is
28        each i in ii::
29          sigma (j in jj:: amount[i,j].x) <= capacity[i].k;
30
31        -- a market must at least receive what it requires
32        mkrequirement is
33        each j in jj::
34          sigma (i in ii:: amount[i,j].x) >= requirement[j].r;
35
36        -- shipments must be positive
37        nonnegbound is
38        each i in ii, j in jj:: amount[i,j].x >= 0;
39
40        -- cost function
41        totcost is
42          sigma (i in ii,j in jj:: amount[i,j].x*cost[i,j].c)
43          objective;
44
45        -- minimal shipment requirement
46        msconstraint is
47        each i in ii::
48          sigma(j in jj:: amount[i,j].x) >= mshipment[i].m;
49
50        -- limitation on transportation
51        shipmentbound is
52        each i in ii, j in jj:: amount[i,j].x <= tplimit[i,j].t;
53
54      problem p1;
55        var rs: boolean;
56
57        -- problem interface
58        interface  rs, capacity, requirement, cost, amount;
59
60        -- index computations
61        ii:= {plant | capacity};
62        jj:= {market | requirement};
63
64        solve (PDQLP: status)
65          variables:   amount.x;
66          objective:   minimize totcost;
67          constraints: mkrequirement;
68                       cpconstraint;
69          bounds:      nonnegbound;
70        end;
```

```
71
72        rs:= status'name="optimal";
73      end;
74
75      problem p2;
76        var rs: boolean;
77
78        -- problem interface
79        interface  rs, capacity, requirement, mshipment,
80                   tplimit, cost, amount;
81
82        -- index computations
83        ii:= {plant | capacity};
84        jj:= {market | requirement};
85
86        solve (PDQLP: status)
87          variables:   amount.x;
88          objective:   minimize totcost;
89          constraints: mkrequirement;
90                       cpconstraint;
91                       msconstraint;
92          bounds:      nonnegbound;
93                       shipmentbound;
94        end;
95
96        rs:= status'name="optimal";
97      end;
98
99    end;
100
101   execution module example;
102     -- declarations
103     include tpdecl;
104     var plantdsr: table [plant; k,m:real] with key plant;
105     var shipmdsr: table [plant; market; c,t,x1,x2:real]
106                         with key plant,market;
107     var rs: boolean;
108
109     interface plantdsr,requirement,shipmdsr;
110
111     -- get values for first problem
112     capacity := {plant,k | plantdsr};
113     cost := {plant,market,c | shipmdsr};
114
115     execute p1 of transportation
116       (rs,capacity,requirement,cost,amount);
117
118     if rs then write(" Optimal solution found");
119       for s in shipmdsr do
120         modify s with x1:=amount[plant,market].x end;
121       end;
122     else write(" No solution found");
```

```
  123     end;
  124
  125     -- get values for second problem
  126     mshipment := {plant,m | plantdsr};
  127     tplimit := {plant,market,t | shipmdsr};
  128
  129     execute p2 of transportation
  130       (rs,capacity,requirement,mshipment,tplimit,
  131        cost,amount);
  132
  133     if rs then write(" Optimal solution found");
  134       for s in shipmdsr do
  135         modify s with x2:=amount[plant,market].x end;
  136       end;
  137     else write(" No solution found");
  138     end;
  139
  140   end;

        no errors found

-> catalog
                     -- the user checks which modules are in the
                     -- data base
   d.tpdecl
   e.example
   m.transportation
-> browse
module name: tpdata
module not found, create new module (y or n)? y
                     -- the user creates interactively a data
                     -- module, defines the data structures, and
                     -- enters the data
processing module tpdata
type ? for help

=> ?
available commands:
   DESCRIBE
   ENTER
   LIST
   RETURN
   SHOW
   OPTION
   ?
=> var plantdata: table[plant; k,m:real]
*                 with key plant;
=> var marketdata: table[market; r:real]
*                 with key market;
=> var shipmentdata: table[plant;market;
*                    c,t,x1,x2:real]
```

```
*                    with key plant,market;
=> list variables;
   plantdata  marketdata  shipmentdata
=> enter plantdata;
#  P1   50   20
#  P2  100   95
#.
=> enter marketdata;
#  M1   50
#  M2   70
#  M3   20
#.
=> enter shipmentdata;
#  P1  M1   5   30   0   0
#  P1  M2   5   40   0   0
#  P1  M3  10    5   0   0
#  P2  M1  15   40   0   0
#  P2  M2  30   50   0   0
#  P2  M3   5   20   0   0
#.
=> return
leaving browsing mode
                     -- the user executes next the example with
                     -- the data module just created
-> run example with tpdata
*    plantdsr = plantdata;
*    requirement = marketdata;
*    shipmdsr = shipmentdata;
*  end

wait ...
    Optimal solution found
    Optimal solution found

-> browse tpdata
processing module tpdata
type ? for help

=> {plant,market,x1,x2 | shipmentdata};

   P1        M1             0.00     10.00
   P1        M2            50.00     35.00
   P1        M3             0.00      0.00
   P2        M1            50.00     40.00
   P2        M2            20.00     35.00
   P2        M3            20.00     20.00

=> return
leaving browsing mode
-> quit
end of session
```

REFERENCES

[ADA]
"Reference Manual for the ADA Programming Language", United States Department of Defense, July 1980

[Aho1]
A.V. Aho, J.D. Ullman, "Universality of Data Retrieval Languages", Conference on Principles of Programming Languages, San Antonio, January 1979

[Aho2]
A.V. Aho, J.D. Ullman, "Principles of Compiler Design", Addison-Wesley Publishing Company, Reading, Massachusetts, 1977

[Aigner]
D.J. Aigner, "An Interpretative Input Routine for Linear Programming", CACM 10, 1, January 1967

[Alter]
S.L. Alter, "A Study of Computer Decision Making in Organizations", MIT, Dissertation, June 1975

[Bayer]
R. Bayer, E.M. McCreight, "Organization and Maintenance of Large Ordered Indices", Acta Informatica 1, 3, 1972

[Brodie]
M.L. Brodie, "Axiomatic Definitions for Data Model Semantic Integrity", University of Toronto, Dissertation, CSRG-91, March 1978

[Burger1]
W.F. Burger, M. Chandy, "A Language and Data Base for Modeling", ORSA/TIMS, November 1978

[Burger2]
W.F. Burger, "XBSW 3.3 - A Parser Generator", Software Systems International, Washington, D.C., December 1979

[Burroughs]
"GAMMA User's Manual", Burroughs Corporation, Detroit, Michigan, 1976

[Chamberlin]
D. Chamberlin et al., "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control", IBM Journal of Research and Development 20, 6, November 1976

[Codd1]
E.F. Codd, "A Relational Model of Data for Large Shared Data Banks", Communications of the ACM 13, 6, June 1970

[Codd2]
E.F. Codd, "A Data Base Sublanguage Founded on the Relational Calculus", ACM SIGFIDET Workshop on Data Description, Access and Control, 1971

[Dahl]
O.J. Dahl et al., "Simula 67 Common Base Language", Norwegian Computing Center, Oslo, 1968

[Date]
C.J. Date "An Introduction to Database Systems", Addison-Wesley Publishing Company, 2nd Edition, Reading, Massachusetts, 1977

[DTBG]
DTBG of CODASYL Programming Language Committee Report, ACM, New York, 1971

[Fromm]
G. Fromm, W.W.L. Hamilton, D.E. Hamilton, "Federally Supported Mathematical Models: Survey and Analysis", US Government Printing Office, Washington, D.C., 1974

[Haverly]
"OMNI Problem Descriptor System", Haverly Systems Inc., Denville, New Jersey, March 1980

[Huits]
M.H. Huits, "Requirements for Languages in Database Systems", in Data Base Description, North-Holland Publishing Company, Amsterdam, 1975

[Katz]
S. Katz, L.J. Risman, M. Rodeh, "A system for constructing linear programming models", IBM Systems Journal, 19, 4, 1980

[Kernighan]
B.W. Kernighan, D.M. Ritchie, "The C Programming Language", Prentice-Hall, Englewood Cliffs, New Jersey, 1978

[Ketron]
"DATAFORM", Ketron, Inc., Arlington, Virginia, 1978

[Klein]
M. Klein, "FINSIM - A Decision Support System for Financial Planning and Engineering", IFIPS 1977

[Labstat]
"LABSTAT User's Guide", U.S. Department of Labor, Bureau of Labor Statistics, 1978

[Meeraus1]
A. Meeraus, "GAMS - General Algebraic Modeling System", World Bank, Development Research Center, August 1977

[Meeraus2]
J. Bisshop, A. Meeraus, "Toward Successful Modeling Applications in a Strategic Planning Environment", World Bank, Development Research Center, Working Paper, March 1980

[Meyer]
D. Meyer, "Entwurf und Implementation von Pascal-Erweiterungen fuer die Bearbeitung relationaler Datenbanken", Universitaet Hamburg, Institut fuer Informatik, Bericht 52, June 1978

[Mitchell]
F.S. Carl-Mitchell, R.F. Berry, A.G. Dale, "April: Accessing and Processing Interface Language for Data Base Applications", University of Texas at Austin, ICSCA, RDS-2, February 1979

[Moriconi]
M.S. Moriconi, "A System for Incrementally Designing and Verifying Programs", University of Texas at Austin, ICSCA, CMP-9, December 1977

[Nori]
K.V. Nori, U. Ammann, K. Jensen, H.H. Naegeli, "The Pascal-P Compiler: Implementation Notes", ETH Zurich, Institut fuer Informatik, Bericht 10, 1974

[Prenner]
C.J. Prenner, L.A. Rowe, "Programming Languages for Relational Database Systems", National Computer Conference, 1978

[Ritchie]
D.M. Ritchie, K.L. Thompson, "The UNIX time sharing system", CACM 17, 7, July 1974

[Schmidt1]
J.W. Schmidt, "Type Concepts for Data Base Definitions: An Investigation Based on Extensions to Pascal", Universitaet Hamburg, Institut fuer Informatik, Bericht 34, May 1977

[Schmidt2]
J.W. Schmidt, "Some High Level Language Constructs for Data of Type Relation", ACM Transactions on Database Systems 2, 3, September 1977

[Schmidt3]
J.W. Schmidt, H. Fischer, M. Jarke, D. Meyer, W. Ullmer, "A Structured Framework for the Implementation of Relations", Bericht 46, University of Hamburg, February 1978

[Smith]
J.M. Smith, D.C.P. Smith, "Database Abstraction: Aggregation and Generalization", ACM Transactions on Database Systems 2, 2, June 1978

[Suvorov]
B.P. Suvorov, A.B. Florinsky, "Main Features of the Automated Information System 'ELLIPSE'", Data Models and Database Systems, Proceedings of the Joint US-USSR Seminar, Moscow, November 1977

[Turnherr]
B. Turnherr, C.A. Zehnder, "Global Data Base Aspects, Consequences for the Relational Model and a Conceptual Schema Language", ETH Zurich, Institut fuer Informatik, Bericht 30, 1979

[Ullman]
J.D. Ullman, "Principles of Database Systems", Computer Science Press, Potomac, Maryland, 1980

[Welch]
J.S. Welch, "Answers Delayed are Answers Denied", SIGMAP Bulletin 28, December 1979

[White]
T.R. White, W.A. Mitchell, "AMBUSH - An Advanced Model Builder for Linear Programming", National Petroleum Refiners Conference, Houston, November 1971

[Wirth1]
N. Wirth, "The Programming Language PASCAL", Acta Informatica 1, 1, 1971

[Wirth2]
N. Wirth, "Modula-2", ETH Zurich, Institut fuer Informatik, Bericht 36, March 1980

[Wolters]
J.A.M. Wolters, "Computer Based Planning and Modeling Systems", IFIPS 1977

[Zannetos]
Z.S. Zannetos, "Intelligent Information Systems: A Decade Later", Sloane School of Management, MIT, Working Paper 1028-78, November 1978