CWI Tract 62

A specification system for statistical software

V.J. de Jong

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

1980 Mathematics Subject Classification: 62-04, 68N20, 68Q40.
ISBN 90 6196 370 2
NUGI-code: 811

Box[1969]:

" Unfortunately statistics has got divided up. There is a U-group called Mathematical Statisticians and a Non-U group called Applied Statisticians. The effect of all the U-manship has, not surprisingly, been to produce a U-shaped distribution of talents with these two groups of people either ignoring each other or else eyeing each other distrustfully and getting further and further apart. The result is that instead of having a productive iteration between theory and practice, which history and common sense both show is the key to progress, we have theoreticians with less and less acquaintance with the real world, and we have work being done by (and advice being given by) applied people having less and less acquaintance with theoretical ideas. "

ACKNOWLEDGEMENTS

A large piece of software is something that one can not develop on his own. A major influence in the choice of the design tools was exerted by Paul Klint from The Center of Mathematics and Computer Sience in Amsterdam. He 'forced' me to use the formal specification language ASF. Though it seemed to lead me further away from my goal of implementing a specification environment for statistical software, it saved me from getting lost in the complexity of a large computer program.
Also I am indebted to Larry Menninga from Western Washington University, who spent his sabbatical leave at the University of Groningen. He introduced me in the world of computer science. Many of his solutions to problems are used in the prototype of CONDUCTOR.
It is, of course, hardly possible to know all the details of existing statistical packages. In this respect I greatly benefitted from stimulating comments of Leo van der Weele.
The prototype of CONDUCTOR was constructed with assistance of Theo Stam and Jan-Herman Veldkamp. They have implemented several modules and spent many hours finding bugs in the system.
I also would like to thank the Computer Science Department of the University of Groningen for letting me be their guest on their UNIX system. Without this system a project of the size of CONDUCTOR is hardly possible.
Finally I must thank Tom Wansbeek, Albert Verbeek and numerous other colleagues for reading preliminary drafts of this book.

CONTENTS

PART I

BASIC CONCEPTS AND TOOLS

1. INTRODUCTION

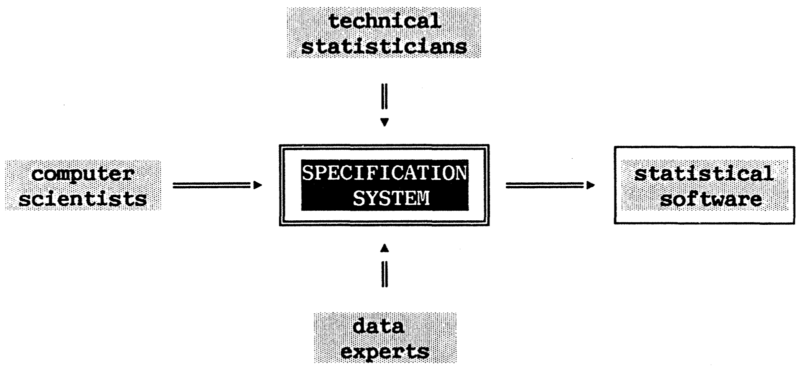
Statistical software developed in the last two decades can only be maintained by professional programmers. It goes without saying that experts in statistics lacking both the required programming skills and the knowledge of the architecture of the software are unable to maintain statistical software. Improvements in statistical techniques, therefore, first have to be explained to programmers. This takes time, costs money and causes errors. In software engineering the problem of *maintainability* is considered to be enormous. Wiener and Sincovec [1984] estimated that maintenance often accounts for more than 80% of the life cycle cost of a software product. A survey by Lientz and Swantson [1980] revealed that approximately 65% of maintenance was perfective (i.e., was the result of changes demanded by the user after the first version of the software product was finished).

An important development that may improve the maintainability of software is the use of *specification* systems. In these systems a *specification language* is available in which the design of the software can be expressed. The specification language is not just some variant of a conventional higher-level programming language. Rather it is a language which is much closer to the problem domain, enabling domain experts to add their knowledge to the system. A program written in a specification language is automatically transformed into an efficient executable program. An instruction in a specification language is equivalent to a large number of instructions in a higher-level programming language. Higher-level programming languages are often referred to as third generation languages; the specification languages evolving from these

languages are also called *fourth generation languages*. An example of a specification system is given in Cheng et al. [1984]. This system allows managers to create software for financial accounting.
In this book we describe a *specification system for statistical software*. Developing and maintaining statistical software is a combined effort of:

- *Technical statisticians*. This group of experts must implement and maintain the statistical techniques in the statistical software,
- *Data experts*. This group of experts must implement and maintain the database in the statistical software,
- *Computer scientists*. This group of experts must implement and maintain the computer science aspects, such as the numerical methods and the user interfaces, of the statistical software.

In a specification system for statistical software each of these groups must be able to contribute their knowledge in a language close to their problem domain. The main design goal of a specification system for statistical software is to orchestrate the individual contributions of each of these groups.



Of course the subdivision of experts is arbitrary. Each group of experts itself again is heterogeneous. For the construction of a specification system for statistical software, however, it is only relevant that the experts in each group are able to express their knowledge in the same language.
A crucial step in the design of a specification system for statistical software is the choice of an appropriate language for each of the groups.

For technical statisticians we created a new language: *the statistical language*. For computer scientists and data experts we chose already existing languages. For computer scientists this is a higher-level programming language and for data experts a database design language. Even though these languages have many drawbacks, in this book no effort is undertaken to improve them. For a discussion on available higher-level programming languages see for example Pratt [1984]; database systems and database design languages are discussed in for example Ullman [1985] and Date [1981].
The statistical language enables a technical statistician to add his knowledge to statistical software in such a way that it can be used by applied statisticians. In the statistical language he can specify for each statistical technique:

- the required input data,
- the resulting statistics,
- the equations describing the calculation of the parameters,
- (pre)tests,
- advice for the user of the technique.

To give concrete form to our ideas in this book a prototype of a specification system for statistical software is developed called CONDUCTOR. To simplify the construction of CONDUCTOR, the current version focusses on econometricians, a subclass of all technical statisticians (CONDUCTOR can of course be tailored for technical statisticians). The statements in the statistical language resemble the mathematical notation used in econometric textbooks, such as Maddala [1977] and Judge et al. [1980]. A technical statistician may use a subset, *matrix notation*, of the mathematical notations used to describe estimation techniques as in these books. In this notation the dimension bounds of matrices are index expressions. This is one of the differences between the statistical language and other matrix languages, like APL (see Pakin [1972]) and SAS/IML [1985].
The main goal of a specification system for statistical software is to produce efficient software. The matrix languages APL and SAS/IML can only be used for what is called rapid prototyping. This is due to the fact that programs written in these languages are interpreted and not compiled. As a result many of the optimization techniques used in compilers for higher-level programming languages can not be applied. If

the programs in matrix languages however could be compiled, the performance of the resulting software would increase dramatically.
A part of the compilation of a program is called type checking. In type checking the type restrictions on particular constructs of the language are verified. A matrix operator, for example, can be restricted to operate on two matrices with equal dimension bounds. The use of symbolic dimension bounds in matrix types complicates the type checking of the statistical language. An important part of this book tries to tackle this problem. Solving this problem removes one of the bottle necks towards the construction of a specification system for statistical software.
The user of a statistical technique initializes the index variables that determine the dimension bounds of the matrices during the execution of the statistical technique. The index variables must be initialized in such a way that no type or dimension bound conflicts occur during the execution of the statistical technique. In software generated by a specification system for statistical software, the user is not aware of the fact that matrices are used in the statistical language. All error messages concerning matrices are therefore meaningless to a user, and should be avoided as much as possible. *Type and dimension bound restrictions* in the statistical language should therefore be checked during the compilation of a statistical program (static type checking).
A matrix with symbolic dimension bounds, as defined in the statistical language, is the equivalent of a dynamic array of reals in a higher-level programming language. Static type and dimension bound checking of dynamic arrays in a higher-level programming language, however, is impossible. A program in these languages may have infinitely many execution trees. That is, there are infinitely many paths leading through a program. To prove that the type and dimension bounds restrictions are not violated in these programs is tedious and in practice often impossible. In higher-level programming languages, and in other matrix languages one, therefore, relies on run-time checks on these restrictions.
The statistical language of our prototype CONDUCTOR has a less complicated control structure than higher-level programming languages:
1. it does not have conditional statements,
2. the only loop control statement is the *for*-statement,
3. a *for*-loop is executed at least once,
4. loop control variables may not be reassigned inside a loop,

5. the input index variables may not be reassigned,
6. index expressions are monotone non-increasing or non-decreasing,
7. dimension ranges are strictly positive.

Due to these properties a statistical program in CONDUCTOR has a *unique symbolic execution tree*: an unconditional symbolic expression, in terms of the input variables, can be calculated for each index variable after each statement in the program. The symbolic expression calculated for the dimension bounds of matrix types are used by the symbolic type and dimension bound checker in CONDUCTOR. If a type or dimension bound restriction can not be verified symbolically, input restrictions are generated on the input index variables in a statistical program. The input variables of a statistical technique are the variables that the user of the statistical technique must initialize in the generated statistical software. This group of variables, and the restrictions on these variables, are assumed to have meaning for an applied statistician who uses the statistical technique. *Symbolic evaluation* was, for example, also used by King [1976] to test higher-level programming language programs with infinitely many execution trees.

The restrictions 1 and 6 on the statistical language are rather strong and seem to demonstrate the impossibility of the creation of a useful statistical language. How these restrictions can be relaxed is one of the things we learned from construction of the prototype. Both restrictions are only imposed in order to make symbolic evaluation of the statistical language possible. We may relax restriction 1 allowing conditional statements, if we impose the restriction that index variables may not be reassigned inside a conditional statement. Under this restriction we still can make an unconditional symbolic evaluation, even though a matrix program as a whole may have infinitely many execution trees. In our prototype CONDUCTOR, however, conditional statements in the statistical language are not implemented. Restriction 6 can be relaxed by introducing an additional data type integer. Of course integers do not have to obey the restrictions on the data type index as long as they are not used in the symbolic evaluation.

Besides the tackling of problems involved in the creation of a statistical language, a major effort discussed in this book is the definition of the interaction between parts of software generated by the different groups of experts. To simplify this definition, in our prototype CONDUCTOR, a *kernel* is introduced. The kernel has a processor that can execute the semantic actions that take place during the evaluation of a statistical program. For the kernel a statistical program is a sequence of these actions, also called kernel instructions. Two examples of kernel instruction are:

- an instruction that requests the user of the statistical technique to initialize an input variable,
- an instruction that calls a numerical function.

These kernel instructions and the numerical counterparts of the functions in the statistical language, must be implemented by computer scientists. CONDUCTOR provides the generated statistical software with *a database interface*. Data in the database is not stored as merely a collection of numbers. Additional information, such as, sample design, instrument and context, collected during the data production process (see David [1985]) can be added to the database. A data expert can write background queries, that are used by the database interface to check the consistency of the retrieved data. When a user of the generated statistical software retrieves a particular series from the database the back-ground query is automatically invoked to do the required consistency checks. Thus detailed knowledge of the data production process is hidden from the user of the data, yet inconsistent data is labeled by the database interface.
CONDUCTOR allows each group of experts to look at a statistical technique from their own level of abstraction. An expert at each of these levels might detect that the execution of a statistical technique must be interrupted because necessary conditions for execution are not satisfied. In computer science such conditions are called *exceptions*, and causing an interrupt is called raising an exception (see Goodenough [1975]). In systems, that must remain in continuous operation, it is important that the execution of a program is not stopped when an exception occurs. Several higher-level programming languages, like ADA, PL/I and PL/C, therefore, provide facilities for exception handlers. When an exception is raised, control of the program is passed to the exception handler. After completion program control is returned to the point where the exception occurred.
Statistical analysis is seen in CONDUCTOR as a continuous process. During statistical analysis, it often happens that a statistical technique can not calculate the required statistics, because necessary conditions are

not satisfied by the analyzed data. Such a situation is seen as the occurrence of an exception. Yet the applied statistician wants to continue the analysis and needs advice. Are alternative statistical techniques available that can tackle the problem? Or do data preprocessing techniques make analysis of his data set possible? The answers, of course, should be given by the experts. The exception handling mechanism in CONDUCTOR allows the experts to provide this information. Exceptions in CONDUCTOR are raised when:

- test results indicate that the applied statistical technique is inadequate for the analyzed data (detected by the software created by the technical statistician),
- computational problems make the calculation of a numerical function impossible (detected by the software created by the computer scientist),
- data is missing or inconsistent (detected by software created by the data expert).

When an exception occurs, the kernel of CONDUCTOR looks for *an exception handler*. Exception handlers describe how the statistical software must react if an exception occurs. Exception handlers can be written by either a technical statistician, a data expert or a computer scientist. For the kernel, exception handlers are independent sequences of kernel instructions. The exception handler mechanism opens the possibility for experts to react on exceptions raised by one of the other groups of experts. An exception handler implemented by a technical statistician, for example, can handle an exception that is raised by a numerical procedure implemented by a computer scientist.

To summarize, the main design goal of a specification system for statistical software is to offer experts in computer science, data collection and technical statistics the possibility to implement and maintain their own restricted contribution to efficient statistical application software. Technical statisticians are trained to think in terms of matrix notation. The widespread use of matrix languages, like APL, among these experts is not surprising. The use of these languages is, however, only suited for rapid prototyping. One can not develop and maintain efficient application software in these languages. In the developed prototype CONDUCTOR, a technical statistician can add his knowledge in a statistical language. CONDUCTOR transforms a program,

written in this language, into efficient statistical software. It combines the software written by the technical statistician with software written by computer scientists and data experts. And, it also generates the appropriate input restrictions.
An applied statistician should not notice the difference between existing statistical software and software produced in CONDUCTOR. He can apply statistical techniques to explore data gathered by data experts. Software constructed in a specification system, however, has two big advantages compared with existing software. The first advantage lies in the *maintainability* of the generated statistical software. New developments in the scientific areas of the different expert groups can be implemented without the need for deliberation with experts in one of the other groups. This will make modern techniques in technical statistics, data production and computer science more rapidly available for applied statisticians. The second advantage is, that software generated in a specification system for statistical software can produce more than just statistics. When during the execution of a statistical technique an exception is raised, the user can be given advice by the appropriate *expert*. Technical statisticians, computer scientists and data experts can implement their messages in exception handlers.
Constructing a specification system for statistical software is a large software project. To make such a project successful it has to be thoroughly specified. To specify the system yet an other language is used: a formal specification language. Such a language enables computer scientists (system developers) to give a formal definition of a software project. Yes this is complicated, this book contains a formal specification of a specification system for statistical software. The formal specification language is used to make a blueprint of CONDUCTOR. Both a UNIX and an MS-DOS version of this prototype exist.
For the specification of our prototype CONDUCTOR we use the formal specification language **ASF** (Bergstra, Heering and Klint [1987]). **ASF** is based on *initial algebraic semantics* for algebraic specifications with conditional equations. Modularization mechanisms in **ASF**, such as parameterization, imports and exports are similar or identical to the ones discussed in Klaeren [1983], Loeckx [1984] and Bergstra et al. [1985]. The formal specification of CONDUCTOR contains definitions of: the data types in the statistical language and in the user language, the

abstract syntax of these languages, the kernel, the translation of a statistical program into a kernel program, the symbolic type and dimension bound checking, and a top level view of the CONDUCTOR environment. A by-product of the CONDUCTOR project is the evaluation of usefulness of **ASF**.

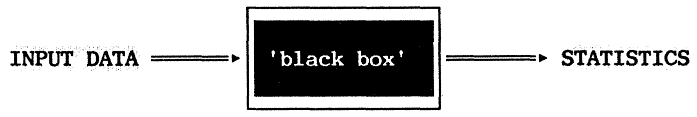
This book is organized as follows. In the remaining chapters of part I a general introduction is given to the basic ideas behind CONDUCTOR and the formalisms used in the definition of CONDUCTOR. Chapter 2 contains a short review of existing software tools in statistics. Information hiding, as applied in CONDUCTOR, is discussed in chapter 3. Chapter 4 shows an example of an implementation of a statistical technique in CONDUCTOR. The formalisms used in the specification of CONDUCTOR, grammars and algebraic specifications, are discussed in chapter 5. In chapter 6 we discuss the general outline of the formal specification of CONDUCTOR. In part II the definition of the statistical language is given. The specification of the concrete and abstract syntax of this language is discussed in chapter 7, the symbolic type checking in chapter 8, and symbolic dimension bound checking in chapter 9. In part III the formal specification of the kernel is given. In chapter 10 the kernel and the kernel instructions are specified. The translation of the statistical programs into kernel instructions is discussed in chapter 11. The exception handler mechanism of CONDUCTOR is described in chapter 12. In part IV both the user language and the data interface are specified. The user language in chapter 13, the data interface in chapter 14. In part V, we conclude, in chapter 15, with a discussion of the prototype, and, in chapter 16, of the use of specification language **ASF**. In chapter 17, we discuss the current status of the CONDUCTOR project and suggest future developments.

2. EXISTING SOFTWARE TOOLS IN STATISTICS

Most empirical work in statistics is done with the use of a few leading statistical packages. These so-called *general statistical packages* contain the commonly used statistical techniques. Well known examples of general packages are SAS [1982], SPSS [1986] and BMDP [1985]. For statisticians with a more specialized field of interest also specialized software exists: *the special purposes packages*. Good examples of special purpose packages are LISREL, a package tailor-made for the estimation of parameters in models with unobservable variables (see Jöreskog and Sörbom [1981]), and TSP [1980], a package for time series analysis. If a statistician wants to use a statistical technique which is not contained in any of the statistical packages, he has to use a *higher-level programming language*. In the statistical community, frequently used higher-level programming languages are FORTRAN, PASCAL, PL/I and C. Also matrix oriented languages, like APL and SAS/IML, are popular among statisticians. In this chapter we will briefly discuss the flexibility and ease of use of both statistical packages and programming languages.

2.1. FLEXIBILITY AND EASE OF USE OF SOFTWARE TOOLS

Statistical packages make statistical techniques available for large groups of users. In order to reach this goal the statistical techniques are implemented as a *'black box'*. The user only has to give the input data and the package returns an impressive amount of statistics.



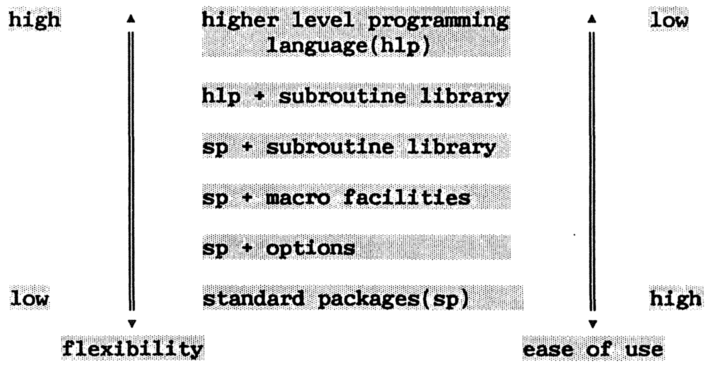
Getting output from a statistical technique in statistical packages, as a result, requires only minimal knowledge of the statistical technique. An example is the use of the instrumental variables technique (INST) in TSP. INST is a statistical technique that can be used if the explanatory variables in a linear model are correlated with the disturbance term. In such a situation ordinary least squares estimates are inconsistent. To solve this problem a group of variables is sought that is both highly correlated with the explanatory variables, and is uncorrelated with the disturbance term. These variables are called instrument variables. INST uses these variables to produce consistent estimates of the coefficients in the linear model. In the TSP program fragment below, a consumption equation is estimated using INST. The first statement in this example gives the name of the program. The second statement in the program loads the data. The third statement gives the equation that is to be estimated. In this example the dependent variable is *consumption* and the explanatory variables are *income* and *consumptionprice*. The variables *importprice*, *export*, *import* and a *constant* are used as instrument variables, as specified in the second part of the statement.

INST program fragment

```
name example "instrumental variables example";
load;
INVR consumption  income consumptionprice
     INST  importprice export import constant;
stop;
end;
```

To get output from INST in TSP only modest knowledge of this statistical technique is needed. The main effort lies in accessing and manipulating the data. Other statistical techniques can be used in a similar way. Of course, for the interpretation of the output, knowledge of INST is highly recommended. Unfortunately this requirement is never enforced by statistical packages. Table 2.1 gives an overview of available techniques for the estimation of equations in standard packages as reported by

Francis [1981], Rodler [1985] and Srba [1985]. The even larger market of statistical software for microcomputers is for example discussed in Woodward, Elliot and Gray [1985] and Van Nes [1987].
The 'black box' approach also has its price from a software engineering point of view. Modifications in the software can only be made by experts with considerable programming skills, knowledge of the statistical technique and knowledge of the architecture of the package in question. And even if one has such rare skills, most commercial software producers do not make available the source programs of their statistical packages. Thus in practice the 'black box' packages are inflexible.
In recent years many features have been added to statistical packages in order to improve the flexibility. Examples of such features are (1) *parameters* allowing the user to choose from different *options* of a statistical technique,(2) *macro facilities* and (3) interfaces with *subroutines* written in higher-level programming languages. Mostly the ease of use of a statistical package (sp) decreases when these facilities are added, while of course the flexibility improves.



If the statistical packages do not contain the desired estimation techniques, a statistician has to use a higher-level programming language. The disadvantage of using a higher-level programming language is obvious: one has to start all over again. Not only the statistical technique has to be implemented, but also user interfaces, report facilities, documentation, etc. This is time consuming. In most cases, after a promising start, the new software engineer ends up in a labyrinth of problems. Only a few will find a reasonable way out. A result is that statisticians write programs that are used by a few friends at most.

Table 2.1 Available estimation techniques in a few leading standard packages for econometric applications.

statistical package	IAS	TSP	SAS/ETS	TROLL
estimation technique				
1. ordinary least squares	x	x	x	x
with AR(1)-correction				
CORC	x	x	-	x
HILU	x	x	-	x
ML-proc	-	x	-	-
iter.	-	-	x	-
2. generalized least squares	-	-	x	-
3. non-linear least squares	x	x	x	x
4. two stage least squares	x	x	x	x
5. instrumental variables	x	x	x	x
6. k-class estimator	x	-	x	x
7. limited information maximum likelihood	x	-	x	x
8. full information maximum likelihood	-	x	-	x
9. limited information instrumental variables	x	-	-	-
10. full information instrumental variables	x	-	-	-
11. three stage least squares	x	x	x	x
12. non-linear 3-stage least squares	-	x	x	-
13. non-linear multivariate regression	-	-	x	-

source: Rodler[1985].

Making the technique available for a large community is simply too much of an effort. One dreams of a kind statistical tool box in which previous efforts can be reused to create software for new developed statistical techniques. A specification system for statistical software is a meant to make that dream come true.

2.2. TWO INSPIRING EXAMPLES

Two software tools developed by large organizations formed an inspiring example for the construction of the statistical language in CONDUCTOR: S, developed by AT&T Bell Laboratory, and IML, an interactive matrix language developed by SAS.

S is a software tool for data analysis and graphics. It emphasizes interactive analysis and graphics, ease of use, flexibility and extendibility. S is developed at AT&T Bell Laboratories and is currently in use under the UNIX operating system. An extensive treatment of S is given in Becker and Chambers [1984a], a short overview of S can be found in Becker and Chambers [1984b]. The design goal of S is stated by Becker and Chambers as: *"to enable and encourage good data analysis, that is to provide users with specific facilities and a general environment that helps them quickly and conveniently to look at many displays, summaries and models for their data and to follow a kind of iterative, exploratory path that most often leads to thorough analysis"*. Particular interesting features of S are:

- in S the language resembles common algebraic notation, using operators and functions,
- S has an interface to user-written functions, which allows functions to be written in a higher level programming language,
- S is centered around "an executive": an interactive parser that parses and evaluates the expressions; "the executive" is an interpreter,
- S focusses on a research environment where statisticians continuously develop new techniques and thus is highly extensible,
- there is a special value NA (not available) which can be used to signify missing data,
- S allows the use of vectors and matrices with fixed dimensions,

- changes in "the executive" of S should not require changes in the code of the user.

IML is a programming language developed by SAS. The basic data elements of IML are matrices. IML can be seen as a successor to the programming language APL in which the rather cryptic special symbols in APL are replaced by a more familiar notation. IML tries to let the user think in terms of matrix notation. One of the big advantages of IML is that it can be used in combination with other software products of SAS such as SAS/GRAPH, which makes IML a powerful tool for matrix oriented scientists. Some interesting features of IML are:

- the matrices in IML are dynamic. The dimension and type of a variable can be changed at any time in a program,
- IML contains a large set of matrix functions and operators,
- no declarations are required in IML, the attributes of a matrix are determined when the matrix is given a value (late binding),
- IML allows data processing,
- IML provides graphic commands.

For a detailed description of SAS/IML see the SAS/IML User's Guide [1985].

Both S and IML are excellent tools for technical statisticians to tackle their problems. Neither tool, however, is intended to be a specification system for statistical software. Programs written in S or IML are used by their creators, and are not meant to be used by others. Both S and IML are suitable for what is called rapid prototyping, and do not produce efficient application software. A program in the statistical language of CONDUCTOR is compiled in order to make efficient execution possible, whereas a program in S or IML is interpreted. The difference between the statistical language in CONDUCTOR and both S and IML is, among other things, reflected in the fact that the matrices in CONDUCTOR's statistical language may have symbolic dimension bounds. Whereas in S and IML the exact dimensions at any time during the interpretation of a program are given.

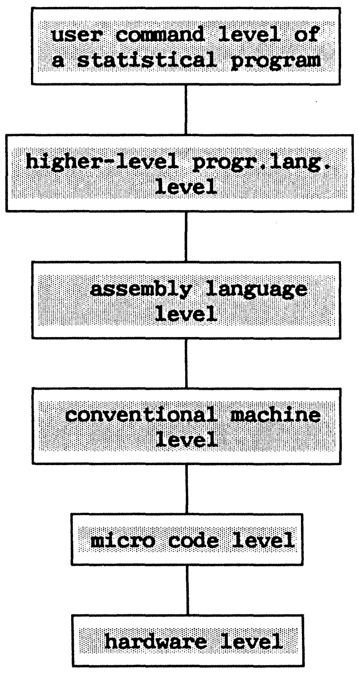
3. INFORMATION HIDING CONCEPTS

In a specification system for statistical software, where different experts cooperate, knowledge implemented by one expert must be completely transparent for experts in other scientific disciplines. This concept, called information hiding, is well-known in computer science. In this chapter two basic concepts of information hiding are discussed: *the multi-layered approach and modularization*. Both in the multi-layered approach and modularization, a problem is tackled at different levels of abstraction. The difference between the two concepts of information hiding is whether or not a separate language is defined at each level of abstraction. In the multi-layered approach a separate language is created for each level, whereas in modularization the solutions of problems at different levels of abstraction are expressed in the same language.

3.1. THE MULTI-LAYERED APPROACH

Consider running a statistical program on a computer. A user of a statistical program gives an instruction, using the command language of the statistical program, to calculate certain statistics. This instruction is equivalent to a large number of micro-code instructions, that are executed by the hardware of the computer. In modern computers many intermediate levels exist between the hardware level of the computer and the statistical program level. The statistical program level is like the top of an iceberg. Underneath the surface are a lot of other levels. Instructions in the statistical language are interpreted by a program written in a higher-level programming language. Statements in the higher-level programming languages are compiled into statements in lower-level languages until finally the hardware level is reached where the instructions are executed by the electric circuits of the computer. For a discussion on the multilevel architecture of computers see Tanenbaum [1976]. Another beautiful example of the multi-layered approach is in the construction of distributed database systems (see Ceri and Pelagatti [1985]).

The multi-layered approach makes it possible to make modifications at one level without influencing the other levels. Changes can be made at, for example, the assembly level without influencing the other levels. Of course, certain changes require that the interfaces between the levels also must be modified. The advantage of the multi-layered approach is that problems can be solved at the appropriate level of abstraction, in a language close to the domain language of the expert who must make the changes.


user command level of a statistical program
higher-level progr.lang. level
assembly language level
conventional machine level
micro code level
hardware level

3.2. MODULARIZATION

At one level of abstraction in the multi-layered approach, a problem can be so complex that it has to be divided into subproblems that can be tackled separately. This form of information hiding is called modularization. In contrast to the multi-layered approach, all subproblems are solved using the same language.
In the academic world it is considered to be 'self-evident' that large and complex pieces of software are constructed using the principles of modular design. Using these design principles in practice, however, appears to be difficult. Parnas et al. [1985] have specified the following goals of module decomposition:

- each module's structure should be simple enough to be understood fully,
- it should be possible to change the implementation of one module without the knowledge of the implementation of other modules and without affecting the behaviour of other modules,
- only very unlikely changes should require changes in the interface of widely used modules,
- it should be possible to make a major software change as a set of independent changes to individual modules,
- a software engineer should be able to understand the responsibilities of a module without understanding the details of the internal design,
- a reader with a well-defined concern should easily be able to identify the relevant modules without studying irrelevant modules,
- the number of branches at each non-terminal module should be small enough that the designer can give convincing arguments that the submodules have no overlapping responsibilities.

For large projects the number of modules is enormous and the modular design principles are difficult to check. Therefore much effort is undertaken to improve modularization techniques and tools. Important in this respect are the development of new programming languages and formal specification languages.
Again a warning for the reader. Do not confuse formal specification languages with the statistical language. A formal specification language is meant for software engineers who develop any kind of software, not

just statistical software. Using this language a software engineer can formalize the requirements and properties of his software. In this book it is used to formalize the requirements of a specification system for statistical software. The statistical language is part of this system that has to be formalized.

3.2.1. **Programming languages.**

New programming languages facilitate the construction of modular software. Important languages in this respect are **ADA** and **MODULA-2**. The ADA programming language was developed at the initiative of tne U.S. Department of Defense (USDoD) between 1979 and 1983. In April 1979 a language design team, headed by Jean Ichbiah of CII Honeywell-Bull won a four-way competition for the best language design. This design was thoroughly tested and revised between April 1979 and July 1982. Nevertheless many computer scientists hold critical views with respect to ADA. For a further introduction to ADA see Wiener [1983] and USDoD [1983]. MODULA-2 was introduced by Niklaus Wirth, the founder of PASCAL, in 1980 (Wirth [1983]). The MODULA-2 programming language overcomes many of the deficiencies of PASCAL. It combines PASCAL's simplicity with much of ADA's power. ADA and MODULA-2 provide facilities for reducing two major difficulties in large scale software design:

- poor interface between separate software components,
- interference between components because shared data (global data) is incorrectly modified by some program unit.

As a result the ***interfaces between separate modules*** in ADA and MODULA-2 are precisely defined.

Though ADA and MODULA-2 are excellent programming languages for large scale software development, they are not the type of language in which non-computer scientists, like statisticians, easily maintain and develop software. The languages simply contain too many features irrelevant to non-computer scientists, while other necessary features, like dynamic arrays of reals, are missing. Another problem is that, even though ADA and MODULA-2 are superior to earlier programming languages, FORTRAN still has the historic advantage that many procedures are already available in procedure libraries like IMSL and NAG.

3.2.2. Formal specification.

The new programming languages offer the possibility to implement modularized projects. An even bigger problem is *how* to modularize a large project. Many modularization techniques exist these days that offer a discipline to modularize problems. An example is the Jackson-design method (see Jackson [1975,1983]). Not surprisingly, no algorithm has ever been found to modularize large scale problems. The process of modularization remains and probably will always remain dependent on the creativity of the problem analyst. Therefore, it is still very important that a system analyst thoroughly defines his problem before he actually starts implementing it. This specification process, however, is also not without problems. The pitfalls in specifying a software project were listed by Meyer [1985] as "the seven sins":

- *noise*: elements in the specification do not add information,
- *silence*: aspects of the problem are not treated in the specification,
- *overspecification*: aspects of the specification do not deal with the problem but with a possible solution of the problem,
- *contradiction*: elements of the specification contradict with other elements in the specification,
- *ambiguity*: elements of the specification can be interpreted in more than one way,
- *forward reference*: elements in the specification refer to problems solved later in the specification,
- *wishful thinking*: there are elements in the specification for which no realistic solution exists.

Some of these sins can be averted if the analyst makes use of a formal specification language. The mathematical notation in such a language is better suited to give a precise description of a problem than natural languages. The language used in this book, for example, helps to remove noise, contradiction, ambiguity and forward reference in the specification of CONDUCTOR. Silence, overspecification and wishful thinking in this specification, however, remain the responsibility of the author of this book.

An important aspect of a formal specification language is, that it has facilities to describe the modularization of a problem. This offers a system analyst the possibility to describe how his problem is divided in subproblems, before he starts implementing such modules. The formal specification langauge **ASF** has such facilities.

3.3. APPLICATIONS OF INFORMATION HIDING

Modularization and the multi-layered approach are applied almost everywhere in computer science. In this section we discuss a few of the applications that influenced the design of CONDUCTOR. These applications are: procedure libraries, the UNIX environment, and very high-level programming producing systems.

3.3.1. Procedure libraries.

A beautiful and simple application of modularization is the standard function in higher level programming languages. These functions can be linked into a program without having to be coded line by line by the programmer who wants to use these functions. For some scientific areas also special libraries are created containing non-standard functions and procedures. Good examples of such libraries are the collection of over 500 mathematical and statistical routines in the IMSL library and the NAG library. Another good example is given by the 400 functions associated with the UNIX "programmers workbench"(see Ivie [1977]).

3.3.2. The UNIX environment.

The UNIX environment contains a variety of facilities that apply information hiding. Besides function libraries, a good example is the UNIX pipe. The UNIX pipe makes whole programs building blocks of larger computational structures. This has led to the development of a literature of specialized programs. These programs structured as simple filters can be applied in many applications.
Other examples of information hiding in UNIX are the shell and the generic facilities. The shell hides the implementation details of UNIX on a particular computer from the UNIX user. Examples of generic facilities are the screen management software (cursors and termcap) and program generators (lex and yacc). The program generator yacc can be used to create a parser for a programming language. The implementation details of the parsing algorithm in yacc are hidden from the user of yacc. A discussion of the information hiding principles applied in UNIX can be

found in Kernighan [1984]. A detailed discussion of UNIX is given in Kernighan and Pike [1984].

3.3.3. Very high-level program-producing systems.

In computer science, a system that in interaction with an expert can produce software for solving the expert's problems, is called a very high-level program-producing system (VHLPPS). Such a system does not accept a variant of a conventional high-level programming language, rather it accepts a language closer to the problem domain of the expert. The system may have a great deal of information built into it, either about the domain or about how to create programs for this domain. Different types of VHLPPS are discussed in Horowitz and Muson [1984]. Examples of VHLPPS are DRACO a system developed by Neighbours and Freeman [1980, 1984], and MODEL developed by Prywes et al. [1977, 1979]).
An approach taken by several researchers is to use a formal specification language as the domain language, and try to transform a program in this language into an efficient program. This approach has only been used for small prototypes and the question is if this approach will work for large-scale software projects. Examples of this approach are given in Arsac [1979] and Balzer [1981].

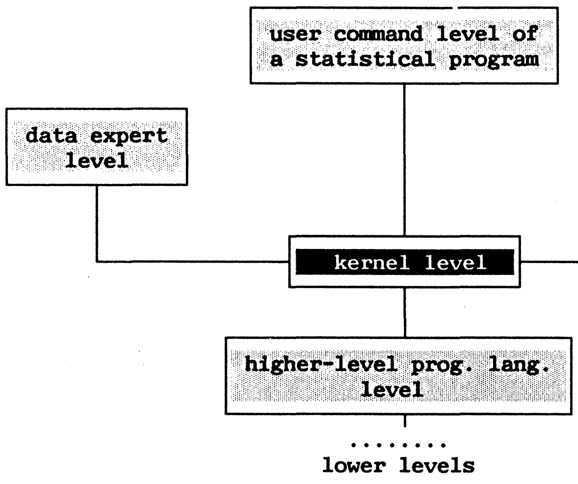
3.4. INFORMATION HIDING IN CONDUCTOR

CONDUCTOR is a VHLPPS: a system for the development of statistical software. CONDUCTOR has built-in knowledge to generate efficient statistical software from the statistical programs written by the statistical expert. Given a statistical program CONDUCTOR generates:
- a user interface,
- an interface with a database,
- input restrictions,
- links with numerical procedures.

Both modularization and the multi-layered approach are applied in the design of CONDUCTOR. The emphasis on the multi-layered approach is reflected by the fact that the two upper levels of abstraction in conventional statistical software, the user command level and the higher-level programming language level, are subdivided into five levels:

- *a statistical level*: at this level statistical techniques are implemented,
- *a data expert level*: at this level data experts can specify how data sets are collected,
- *a user command level*: the command level of the resulting software. It resembles the conventional user command level in statistical software packages. Using a command language a user can apply statistical techniques to data stored in a database,
- *a kernel level*: statements at the statistical level are translated into kernel instructions. The kernel level is a shell that allows computer scientists, technical statisticians and data experts to make changes on their own level without knowledge of the other levels,
- *The higher-level programming level*: At this level, computer science experts can add their knowledge to CONDUCTOR, and, of course, CONDUCTOR itself is implemented.

Procedures and functions in libraries are the building blocks of CONDUCTOR: the conductor kernel calls these procedures and functions. Examples are procedures to calculate the inverse of a matrix, and procedures to calculate the eigenvalues and eigenvectors of a matrix. CONDUCTOR allows technical statisticians to combine these procedures in infinitely many ways, freeing them of the burden of the tedious parameter substitution problems involved in the use of these procedures in higher-level programming languages. From the UNIX operating system CONDUCTOR 'steals' the idea of the shell (the kernel of CONDUCTOR), which hides tedious details involved in the implementation of UNIX on a particular computer from the main design of UNIX.
CONDUCTOR consists of the kernel plus the four interfaces with the surrounding levels, to wit a compiler for the statistical language, a user language interpreter, a database interface, and facilities to connect numerical procedures written in a higher-level programming language. The statistical and data expert level are built on top of a higher-level programming level. This is the general outline of a specification system in which statistical and data experts can create, debug and modify their (restricted) contribution to the software.


user command level of a statistical program
data expert level
statistical expert level
kernel level
higher-level prog. lang. level
....
lower levels

Creating a specification system for statistical software is a large software project. A necessary condition for such a project to be successful is that all requirements are described, and that the project is modularized. Therefore a complete formal specification of CONDUCTOR is given in the specification language **ASF**. ADA or MODULA-2 are very good higher-level programming languages to implement CONDUCTOR. One could benefit from the elegant way in which modularization is possible in these languages. CONDUCTOR, however, is implemented in C and runs under both the UNIX and the MS-DOS operating systems. The choice for C was mainly based on practical reasons. ADA and MODULA-2 compilers were not available at the computers used by the author. Yet, C in combination with the UNIX operating system forms an excellent software development environment, and was available on the VAX computer used by the author.

4. THE IMPLEMENTATION OF A STATISTICAL TECHNIQUE

In this chapter it is demonstrated how a statistical technique can be implemented in our prototype specification system for statistical software. As an example we use a statistical technique known as the bootstrap method. After a brief introduction of this statistical technique, it is shown how a technical statistician can implement this technique, and how it can be used by an applied statistician in the resulting software. Furthermore the role of the kernel and the contributions of both the data expert and the computer scientist in this example are discussed.

4.1. THE BOOTSTRAP METHOD

The bootstrap method is a statistical method that can be used to estimate the statistical error of estimated parameters. In this example the method is used to estimate the variance of regression coefficients in a linear model. In the linear model it is assumed that the variation in the dependent variable **y** can be explained by the variation in the independent variables $X_1,..,X_k$.
Suppose (y,X) is a realization of the random matrix $(\underline{y},\underline{X})$, whose rows are identical independently distributed with unknown distribution **F**; **X** is a matrix of order **n x k** and contains the observations of the independent variables, and **y** is an **(n x 1)**-vector of the observations on the dependent variable. Define

$$\underline{b} = (\underline{X}'\underline{X})^{-1}\underline{X}'\underline{y}$$

so that $\underline{X}\ \underline{b}$ is the orthogonal projection of $\underline{y}$ on the space spanned by $\underline{X}$ (a general formulation of the well-known regression coefficient in the linear model). The distribution of $\underline{b}$ is unknown, but asymptotic theory usually yields useful approximations. The bootstrap offers a valid alternative replacing mathematical analysis in the field of asymptotic theory by 'massive calculations'. Its basic idea is simple: estimate F by the empirical distribution $\hat{F}$ of the observations (y,X) and perform a Monte Carlo study drawing samples from $\hat{F}$.

$\hat{F}$ = mass (1/n) on each observed data point of y.

More precisely the bootstrap method proceeds as follows:
- draw with replacement a sample from the n observations of ($\underline{y}$,$\underline{X}$)
- calculate estimator **b** for the sample,
- repeat the first two steps mentioned above and calculate the average and variance of the sample estimates.

The calculated average and variance are respectively the bootstrap estimates of the expected value of the regression coefficient and the variance of this parameter. For a more detailed discussion of the bootstrap method see for example Efron and Gong [1983].

4.2. IMPLEMENTATION OF THE BOOTSTRAP METHOD

A statistical technique is implemented in CONDUCTOR in the statistical language. A program in this language defines:
- the *name* of the technique,
- the *input variables*, the variables that must be initialized by the user, and the *output variables*, the variables that contain the results of the statistical technique,
- the *equations* that specify how the required statistics are calculated,
- the *tests* of either the basic assumptions of the implemented technique, or the significance of the estimated parameters; when the test results are negative an exception is raised

- the *exception handlers* for exceptions that may occur during the execution of the statistical technique.

For the bootstrap technique we first have to decide which identifier must be used in the resulting software to call the technique. An obvious choice is the identifier '*bootstrap*'. This identifier is given in the name section of a statistical program.

name section statistical program

```
NAME bootstrap
```

Next it has to determined what are the input and output variables of the bootstrap technique. Clearly the user has to give the observations for the dependent and independent variables, **Y** and **X**. Furthermore he must indicate the number of observations **n** and the number of independent variables **k** in his application[1]. In the example below, also the size of each random sample **s** and the number of times a sample is drawn **d** must be determined by the user. The output variables are the estimated expected value of the regression coefficients **b** and the corresponding variance (**var_b**).

Messages, that are used to prompt for input, may accompany the declaration of input variables. Similar, a message in an output declaration may explain the calculated result. The input and output variables are declared in the input/output section of a statistical program.

input/output section statistical program

```
INTERFACE

 INPUT

   INDEX   k          MESSAGE: "number of independents";
   INDEX   n          MESSAGE: "number of observations";
   INDEX   s          MESSAGE: "size of bootstrap sample";
   INDEX   d          MESSAGE: "how many samples must be drawn";
```

[1] Of course CONDUCTOR can be made smarter by adding a function that can determine the dimensions of a matrix, thus removing the burden from the user to enter this data. In the current version of CONDUCTOR, however, this is not implemented.

```
  MATRIX [1 to n, 1 to k]          X
          MESSAGE: "The independent variables are:";
  VECTOR [1 to n]                  Y
          MESSAGE: "The dependent variable is: " ;
OUTPUT
  VECTOR [1 to k]                  b
          MESSAGE: "The estimated expected value is: " ;
  VECTOR [1 to k]                  var_b
          MESSAGE: "The variance of this estimate is"
```

The equations in the implementation section of a statistical program specify how the resulting statistics **b** and **var_b** are calculated. The implementation section consists of a part in which internal variables are declared and a part in which the equations are given.
Here a remark on the choice of the functions in the specification of the bootstrap technique is on its place. In statistical computing it is well known (see Kennedy and Gentle [1980]) that using the inversion procedure is not the most efficient and stable approach to estimate the regression coefficients. Other procedures, like Cholesky and Householder decomposition, in many cases have better numerical properties. For the design of CONDUCTOR, however, this is irrelevant. The choice of a particular set of functions only influences which experts are able to use the statistical language. To improve the readability of the examples we have chosen the notation used in econometric textbooks. Adding Householder and Cholesky decompositions function to CONDUCTOR would require technical statisticians with more knowledge of the numerical problems in statistical computing. Surely this would improve the efficiency of the generated software, but it would reduce the group of potential users.

implementation section statistical program

```
VARIABLES

     INDEX       i,j,sample ;
     MATRIX      [1 to s, 1 to m]       X_temp;
     VECTOR      [1 to s]               y_temp;
     VECTOR      [1 to k]               b_temp;

EQUATIONS

beta[i] = 0   i = 1,..,k ;

FOR  i := 1 TO d DO
   {
    /* draw sample */
    FOR  j := 1 to s DO
         {
          sample := draw_random_index(1,n);

          X_temp[j,1 to m] := X[sample,1 to m];
          y_temp[j]        := y[sample];
         };

    /* calculate the OLS estimator */

    b-temp := inv(X_temp'*X_temp)*(X_temp'*y_temp);

    /* update b and b_var */

    beta  := ((i-1)/i)*beta + (1/i)*b_temp;

    b_var := b_var - (i-1)/i * (b_var-beta)^2
   }
```

In test sections the technical statistician can specify tests that check the significance of the resulting statistics. A test, for example, may check if all estimated regression coefficient are significant. In the test section below this test is 'the rule of thumb'. This rule states that an estimated regression coefficient is significant if it is at least twice the size of the estimated standard deviation of the coefficient.

test section statistical program

```
TEST
        RAISE  insignificant_coefficients

        WHEN   abs(beta) <= 2*sqrt(b_var)
```

Note that the functions **abs** and **sqrt** and the operator <= operate element-wise on the vectors. If all the inequalities hold the result of the <= operator is the boolean value **true**. Though the matrix notation suggests that this is a combined test of the significance of the estimated coefficients, it consists in fact of independent tests of the individual coefficients in the vector beta.
After the execution of the implementation section of the bootstrap technique the conditional expression in the test is evaluated. If the result is **true**, an exception is raised and the execution of the technique is interrupted. During the execution of the bootstrap technique also exceptions can be raised in software created by other experts. Exceptions are regarded as signals that further execution of the technique is impossible or meaningless.
In the statistical program the technical statistician can write exception handlers. An exception handler determines how the software reacts in case an exception occurs. An exception handler may consist of a warning for the user of the statistical technique, as shown in the following exception handler section of a statistical program.

```
exception handler section statistical program

WHEN
  near_singular:

  MESSAGE:
    "THE INDEPENDENT VARIABLES IN SOME BOOTSTRAP SAMPLES ARE
     STRONGLY CORRELATED ( MULTICOLLINEARITY). DUE TO THIS THE
     NUMERICAL RESULTS OF THE TECHNIQUE MAY BE INACCURATE."
```

If an exception *near_singular* is raised, during the execution of the bootstrap technique, the message in the exception handler is displayed. Because no explicit **STOP** statement is added to the exception handler, the execution of the statistical technique continues after the exception handler is executed. The example shows that the exception handling mechanism makes it possible for a technical statistician to react on exceptions raised by the software created by other experts. In the example the check on near-singularity is assumed to be implemented by a computer scientist in a higher-level programming language. When a near-singular matrix is inverted the numerical results may be inaccurate. The inversion function in the bootstrap program inverts the matrix (X'*X).

The notion of near-singularity in this case is interpreted by the technical statistician as multicollinearity: the independent variables are strongly correlated. The exception handler of the technical statistician communicates this meaning to the user of the statistical technique.
In the example no **STOP** statement was added to the exception handler. By adding such a statement, the technical statistician determines that the execution of the statistical technique must be stopped when an exception occurs, because the exception is considered to be fatal. This decision is clearly an expert opinion; it may be different for different statistical techniques.

4.3. INPUT RESTRICTIONS

The applied statistician who uses the generated statistical software must not be confronted with cryptic errors concerning the matrix notation used by the technical statistician. CONDUCTOR, therefore, performs *extensive type and dimension bound checks* to reduce the number of cryptic error messages that may occur during the execution of a statistical technique. Not all the type and dimension bound restrictions can be checked during the compilation of a statistical program. The input restrictions, however, that remain to be checked during the execution of the statistical technique are *restrictions on the input index variables*. These variables, and restrictions on these variables, are assumed to have meaning for the user of the statistical technique. In our example the following restrictions on the index variables of the bootstrap technique are generated[2]:

[2] In the current version of CONDUCTOR all input restrictions are written as

```
          expression >= 0,
or        expression  > 0,
or        expression  = 0
```

```
n - 1 >= 0
m - 1 >= 0
s - 1 >= 0
n - s >= 0
d - 1 >= 0
```

The input restrictions are checked after the user has initialized all input index variables of the statistical technique.

4.4. USING THE BOOTSTRAP TECHNIQUE

The use of the bootstrap technique is demonstrated in the following user session. In this example the user first retrieves data from a database. He indicates which sample is to be retrieved from the database and which variables he wants to retrieve. The symbol >u: is the prompt of the user language interpreter in CONDUCTOR. After retrieving the data from the database, the user calls the bootstrap technique, and the user is asked to initialize the input variables.

```
                         user session
>u:  set sample  1945 to 1985
>u:  retrieve income, indirect_tax, credit, consumption
>u:  bootstrap
number of independents
>u:  3
number of observations
>u:  41
size of bootstrap sample
>u:  20
how many samples must be drawn
>u:  100
The independent variables are
>u:  income, indirect_tax , credit
The dependent variable is
>u:  consumption
```

After the results of the bootstrap method are calculated, the requested output is displayed. If exceptions occur during the execution of the statistical technique, warnings and suggestions from experts may accompany this output. For example, it may have been detected that the matrix inversion was near-singular. The interrupt, handled by the exception handler defined by the technical statistician in the bootstrap technique, displays the specified message in this handler.

4.5. THE DATA

In the user session, in the previous section, series were used for consumption, income, indirect tax and credit. In statistical research these series are often produced by data experts. In the Netherlands a large part of the data in statistical research is produced by the Central Bureau of Statistics. During the data production process additional information on the series is gathered. In CONDUCTOR it is assumed that this additional information is stored in a database. Unfortunately such a database is still a dream. CONDUCTOR only contains the interface with this dream, and hopefully is an extra motivation to make this dream come true.
When the series is retrieved from the database, the additional information is checked for consistency. The consistency checks must, of course, be written by the data expert. In our example a consistency check may reveal that the series *consumption* is based on different types of measurement in the sample period. When an inconsistent series of observations is used by a statistical technique an exception is raised. A data expert can specify an exception handler to warn the user. Note that this exception handler is not a part of a statistical program. Such an exception handler is called an external handler in CONDUCTOR.

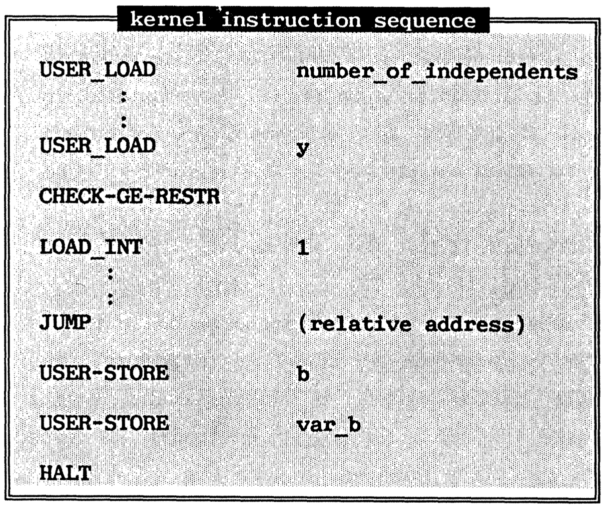
4.6. THE KERNEL

A statistical program is executed when it is called in a user session. The executing technique may invoke exception handlers written by either the data expert, the technical statistician or the computer scientist. During the execution of a statistical technique a limited number of semantic actions are executed. These actions form the instruction set of a virtual machine called the kernel. The kernel has instructions to

- request the user to initialize the input variables,
- evaluate expressions on a stack,

- change the sequential pattern of execution,
- control the exception handling,
- display messages,
- check input restrictions,
- terminate execution.

A compiler in CONDUCTOR translates a statistical program into a sequence of kernel instructions. The bootstrap technique specified in section 4.2, for example, is translated into the following instruction sequences. Of course, only a summary of all generated instructions is shown.



```
kernel instruction sequence

USER_LOAD            number_of_independents
     :
     :
USER_LOAD            y
CHECK-GE-RESTR
LOAD_INT             1
    :
    :
JUMP                 (relative address)
USER-STORE           b
USER-STORE           var_b
HALT
```

The instruction sequence generated for the exception handlers in the statistical technique, are stored separate from the main instruction sequence of the statistical technique.

4.7. THE CONTRIBUTION OF COMPUTER SCIENTISTS

The numerical procedures in the bootstrap technique, such as matrix inversion and the generation of a random number, must be implemented by computer scientists in a higher-level programming language. Furthermore, these experts, of course, have to create CONDUCTOR.

5. THE FORMALISMS USED IN THE DESCRIPTION OF CONDUCTOR

In this chapter we introduce the formalisms used in the definition of CONDUCTOR. The definitions of the statistical, the user and the kernel language, together form the definition of CONDUCTOR. The definition of programming languages has received a lot of attention in the last two decades (see McGettrick [1980]). From the informal description of the first programming language many, more formal, definitions have evolved. A complete language definition nowadays consists of:

- a set of building blocks (words), this set is called lexicon, or sometimes alphabet,
- a possibly infinite set of sequences of words, called sentences,
- a predicate on sentences indicating whether or not a sentence is an element of the language,
- the specification of the semantics of the sentences.

The first three parts of this definition are referred to as the *syntax* of the language. In section 1 of this chapter a short review of syntax definitions is given. For a discussions on this topic see also Aho and Ullman [1977] and Aho, Sethi and Ullman [1986].
To define the *semantics* of the languages in CONDUCTOR we use the formal specification language **ASF**. **ASF** is discussed in section 2 of this chapter.

5.1. SYNTAX DEFINITIONS

Among the many notations used in describing the syntax of a programming language are *context-free grammars* and *syntax trees*.

5.1.1. Context-free grammars.

A context-free grammar **G** is a 4-tuple **(N,T,S,P)**, where **N** is a finite set of non-terminals, T is a finite set of terminals, **S** is a start symbol, and an element of **N**, and **P** is a finite set of production rules. The terminals are the symbols from which sentences are formed in the language. The set of non-terminals **N** are the grammatical categories of the language. The start symbol **S** is a special non-terminal indicating a correct sentence in the language. The production rules **P** specify the ways in which sentences can be constructed from **S**. A string of terminals is a correct sentence in the grammar **G**, if and only if it can be derived from S using the production rules.

Example 5.1.: (from Aho and Ullman [1977]).

Consider the grammar **G** for simple arithmetic expressions. The non-terminals are **expression** and **operator**, with **expression** as the start symbol.

```
N = { expression, operator }
S = expression
```

The set of terminals is:

```
T = { id, +,  -,  *,  /,  ^,  (,  ) }
```

The set of productions **P** contains the productions:

```
expression  --> expression  operator expression
expression  --> ( expression )
expression  --> id
operator    --> + | - | * | / | ^
```

Examples of correct sentences in this language are:

```
id * id - id
id
id * (id - (id + id))
```

{end example 5.1.}

The process of checking whether a sentence can be generated by the grammar is called *the parse* of a sentence. In order to be able to write an efficient parser for a grammar, the grammar must satisfy certain restrictions. For most programming languages either LL(1) or LALR(1) grammars are used to describe their syntax. These grammars are subclasses of the general class of context-free grammars, but they are powerful enough to describe most syntactic constructs in these languages. Both grammar classes guarantee that no back-tracking is required in the parse. LL(1) parsers are top-down, deterministic parsers with one symbol lookahead, and were first described by Foster [1968], and received a theoretical treatment in Knuth [1971]. LL(1) parsers are for example used in compilers for programming languages like PASCAL and MODULA-2. LALR(1) parsers are bottom-up deterministic parsers with one symbol lookahead. The LALR(1) parsers are a subclass of the more general LR parsers introduced by Knuth [1965]. Examples of languages using LALR(1) grammars are ADA and C.
For the LALR(1) grammars there exist algorithms which can automatically construct a parser. LR(1) parser-construction algorithms are due to DeRemer [1969, 1971]. An efficient parser-constructor algorithm is also given in Park, Choe and Chang [1985]. An example of a parser generator is *yacc* written by Johnson [1975]. In CONDUCTOR languages are defined using grammars that satisfy the LALR(1) restrictions. The parser in the prototype of CONDUCTOR is developed with the use of *yacc*.

5.1.2. Syntax trees.

A useful representation of the result of a parse are syntax trees. The tree representation of the parse of a sentence in the grammar is called the *concrete syntax tree or parse tree*. Each node in this syntax tree represents a non-terminal of the grammar, the leafs represent terminals.

Example 5.2.:

The parse of the last sentence of example 5.1. can be represented by the following concrete syntax tree (the non-terminals are abbreviated):

```
              expr
       /       |      \
     expr     op       expr
      |        |     /  |   \
     id        *    (  expr   )
                       / | \
                 expr   op    expr
                  |      |   /  |  \
                 id      -  ( expr )
                             / | \
                        expr  op  expr
                          |    |    |
                         id    +   id
```

{ end example 5.2.}

A condensed version of the concrete syntax tree is called the *abstract syntax tree*. In the abstract syntax tree superfluous information in the concrete syntax tree is removed. Superficial distinctions in form, unimportant for the translation, do not appear in the abstract syntax tree. McKeeman [1974] showed that the transformation of a concrete syntax tree into a abstract syntax tree can be described by the use of a transduction grammar.

Example 5.3.:
Using the transduction grammar with the following "tree constructing rules"

```
expr  --> expr op expr    ==>        op
                                    /  \
                                expr    expr

          | ( expr )      ==>       expr

          | id            ==>       id

op    --> +               ==>       +

          | -             ==>       -

          | *             ==>       *
```

the concrete syntax tree of example 5.2. reduces to the abstract syntax tree

```
       *
     /   \
   id     -
        /   \
      id     +
           /   \
          id    id
```

{end example 5.3}

5.2. ALGEBRAIC SPECIFICATION

For the description of CONDUCTOR we use the formal specification language **ASF**, defined in Bergstra, Heering and Klint [1987]. This description includes a description of the semantics of the various languages in CONDUCTOR. **ASF** is based on algebraic specification techniques as described in Klaeren [1983], Wirsing [1983], Gaudel [1984] and Loeckx [1984]. **ASF** extends the algebraic specification formalism based on signatures and sets of equations in several ways. It supports (1) prefix and infix operators, (2) multiple output values of functions, and (3) module expressions. In **ASF** it is possible to give an algebraic specification, with conditional equations, of the languages in CONDUCTOR. For a review on algebraic specifications see Meseguer and Goguen [1982] and Klaeren [1984].
The basic concepts in algebraic specification are sorts, carrier sets and signatures. One can think of a sort as an abstract data type. The elements of a carrier set represent distinct instances of a sort. Consider, for example, the sort *Booleans*. If we take the intuitive meaning of this sort it could be represented by the carrier set (0,1) or the carrier set (*true, false*). Of course infinitely many other carrier set (representations) of *Booleans* can be chosen. A signature describes a set of functions. For each function it is exactly specified which sorts are expected as input and what sort is returned as output. A signature is defined more thoroughly by Meseguer and Goguen [1982]. A signature can be defined as

Let S be a set of sorts. If **a** and **b** are both sorts in S. Then a family of functions is defined as

{ f: a -> b | a,b in S }

These are typically all the function with as input of sort **a** and output of sort **b**. If we allow more than one sort as input we get what is called an S-sorted signature Σ

$$\Sigma_{w,s} = \{f\colon w \rightarrow s \mid w \text{ in } S^*,\ s \text{ in } S\}$$

A signature defines some structure of interest. For example the signature *Boolean* specifies booleans using a set of sort *(bool)* and a set of functions *(true,false, and)*.

Boolean = { (bool) , (true,false,and) }

with

```
true:                  -> bool
false:                 -> bool
and:     bool # bool   -> bool
```

By assigning a carrier set to each of the sorts we get a what is called a *Σ-algebra*. Many Σ-algebras may exist for the same signature, therefore *an initial Σ-algebra* is defined as:

A Σ-algebra A is initial in a class of Σ-algebras Φ that describe the same structure if and only if there is only one Σ-homomorphism for each Σ-algebra C in Φ from A to C (see Meseguer and Goguen [1982]).

In other words an initial Σ-algebra is the 'smallest' representation of a structure. Note that there may be more than one initial Σ-algebra in the same class. Two initial algebras in the same class Φ are abstractly the same but differ in the representation given to the elements.
From the functions in a signature Σ-terms can be formed, similar to

sentences in context-free grammars. This set of terms can be used as a carrier set. All distinct terms are a possible representation of the distinct data-items in a signature. For example, in our signature Boolean we may form the terms

true, false, and(true,false), and(and(true,false),false),...

It is clear that many of the above terms represent the same data item. *Equations* are introduced to specify which terms are equal. To restrict the Σ-algebra *Booleans* to only two data items the functions in this signature must satisfy the equations

```
and(true,false)  = false
and(false,true)  = false
and(false,false) = false
and(true,true)   = true
```

To get meaningful restrictions the sets of equations should obey the following restrictions:

- applying the equational logic to deduce new equations should always yield equations that are satisfied by any algebra satisfying the equations (*soundness*),
- every equation, satisfied by all algebras satisfying the given equations, can be deduced using the equational logic (*completeness*).

Using one signature to describe a large software product would yield an enormous amount of sorts, function and equations. In **ASF**, therefore, a signature can be modularized. In a *module expressions* make one can import sorts and functions specified in another module by importing that module. Each module may contain an export clause indicating which sorts and functions can be imported by other modules. Our Booleans can be expressed in the following module

```
module Booleans
begin
  export
    begin
      sorts BOOL
      functions
         true:                  -> BOOL
         false:                 -> BOOL
         and:    BOOL # BOOL    -> BOOL
    end
```

```
 functions
        not:    BOOL            -> BOOL

 equations

      and(true,false)  = false
      and(false,true)  = false
      and(false,false) = false
      and(true,true)   = true

      not(true)        = false
      not(false)       = true

end Booleans
```

The module *Booleans* is imported in the module *Integers*. This module can use the functions *true*, *false* and *and* specified in module *Booleans*. For example, the constant functions *true* and *false* are used in the specification of a function *equal* in module Integers.

```
module Integers
begin
     export
     begin
       sort INT
       functions
         null:                  -> INT
         increment:  INT        -> INT
         equal:      INT # INT -> BOOL
     end

     imports Booleans

     variables

      i,i1,i2         :-> INT

     equations

      equal(null,null)                        = true
      equal(null,increment(i))                = false
      equal(increment(i),null)                = false
      equal(increment(i1),increment(i2)) = equal(i1,i2)

end Integers
```

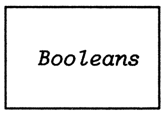
To make modules more generally applicable, *parameterization* is available in **ASF**. Each formal parameter is a submodule and contains one or more sorts or functions, which at a later stage have to be bound to the actual parameter. Consider the formal specification of a sequence

```
module Sequences
begin
   parameter Items
     begin
       sorts ITEM
     end Items

   exports
    begin
       sorts SEQ
       functions
           add-item: ITEM # SEQ -> SEQ
    end

end Sequences
```

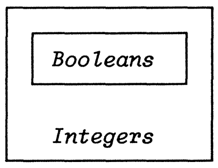
A sequence of integers can be specified by binding the parameter *Items* to the module *Integers*.

```
module Integer-sequences
begin
   imports Sequences
      { renamed by
           [ SEQ   -> INT-SEQ ]
        Items bound by
           [ ITEM  -> INT ]
             to Integers }
end Integer-sequences
```

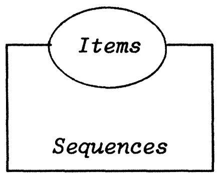
The module *Integer-sequences* defines the sort *INT-SEQ*. All functions specified for the sort *SEQ* are also defined for the sort *INT-SEQ*. Note that not all parameters have to be bound when a module with parameters is imported. Such unbound parameters are called inherited parameters.
The overall structure of specifications is illustrated by *structure diagrams*. Each module is represented by a rectangular box. For example the module *Booleans* is represented by



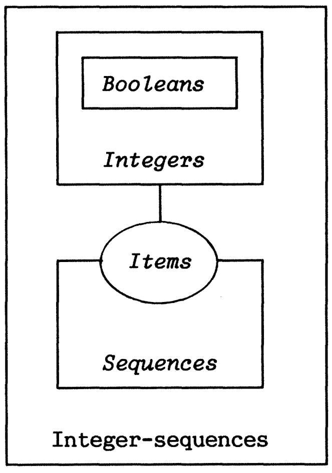
Modules imported by a module M are represented by structure diagrams inside the box representing M. For example the module *Integers* imports the module *Booleans*.



Parameters of the module are represented by ellipses carrying the name of the parameter. The module Sequences with parameters Items, for example, is represented by:

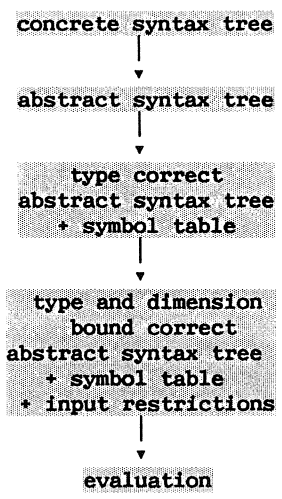


The binding of formal parameters is represented by joining the formal parameter and the module to which it is bound. This leads to the following representation of the module Integer-sequences:



5.3 THE DEFINITION OF THE LANGUAGES IN CONDUCTOR

An important part of the definition of CONDUCTOR consists of the definition of the statistical language, the kernel language and the user language. The first part of the definition of the statistical language defines *the concrete syntax*. The next part is the definition of *the abstract syntax*. In the abstract syntax tree the `syntactic sugar` of the concrete syntax is removed. The third part defines the type restrictions in the statistical language. A type correct statistical program is represented as *a type correct abstract syntax tree* plus a symbol table containing the type information of all variables in the abstract syntax tree. The fourth part is the definition of the dimension bound restrictions. *A type and dimension bound correct abstract syntax tree* includes a set of *input restrictions*. In the final part of the definition of the statistical language, the evaluation of the abstract syntax tree is defined.


concrete syntax tree
abstract syntax tree
type correct abstract syntax tree + symbol table
type and dimension bound correct abstract syntax tree + symbol table + input restrictions
evaluation

A type and dimension bound correct statistical program is represented in the kernel as a kernel program. The semantic actions of the statistical program in this representation are a sequence of kernel instructions. The set of all possible kernel instructions is called the kernel language. Of the kernel language only the abstract syntax and a description of the

evaluation are specified. The definition of the user language consists of a concrete syntax, an abstract syntax, and the evaluation process of abstract syntax trees. One of the semantic actions in the user language is the evaluation of a statistical program.
The concrete syntax of the user and the statistical language is described by a grammar. This grammar satisfies the LALR(1) restriction and can be found in appendix A. The remaining parts of the language definitions are described in the formal specification language **ASF**. The definition of the statistical language is given in part II of this book. The kernel language and the evaluation of statistical kernel programs can be found in part III. The data interface and the user language are given in part IV.

6. THE FORMAL SPECIFICATION OF CONDUCTOR

The definition of CONDUCTOR is a combined definition of the statistical, the user and the kernel language. The definition consists of the grammar describing the concrete syntax of these languages, as given in appendix A, and the formal specification, as given in appendix C through L. In this chapter we discuss the modules that describe CONDUCTOR at a high level of abstraction. These modules can be found in appendix L.

6.1. A TOP LEVEL VIEW OF CONDUCTOR

CONDUCTOR maintains a global state. This state is determined by:
- the statistical techniques implemented by the technical statisticians,
- the external handlers implemented by either the computer scientists or the data experts,
- the state of a user session.

All implemented statistical techniques are stored in the statistical technique table. This table is represented in the formal specification by the sort *STAT-TECH-TABLE*. All external handlers are stored in the external handler table, represented by the sort *EXT-HANDL-TABLE*. The state of a user session is specified by the sort *USER-STATE*. Every combination of the sorts *USER-STATE*, *STAT-TECH-TABLE* and *EXT-HANDL-TABLE* is a state of the CONDUCTOR. This state is represented by the sort *CDT-STATE*. The function *state* in module *Conductor-states* specifies the state of CONDUCTOR *state: USER-STATE # STAT-TECH-TABLE # EXT-HANDL-TABLE -> CDT-STATE*

The details of the sorts *USER-STATE*, *STAT-TECH-TABLE* and *EXT-HANDL-TABLE* are defined in the imported modules of module *Conductor-states*.
The state of CONDUCTOR can be modified in session. Module *Conductor-sessions* specifies that a session is either:

- a session of a technical statistician; in such a session a technical statistician implements a statistical technique (represented by the sort *STAT-PRO*),

  stat-session: STAT-PRO -> SESSION

- a session in which either a data expert or a computer scientist implements an external handler (represented by the sort *HANDLER*),

  hand-session: HANDLER -> SESSION

- a session in which the implemented statistical software is used (such a session consists of a user program represented by the sort *USER-PRO*).

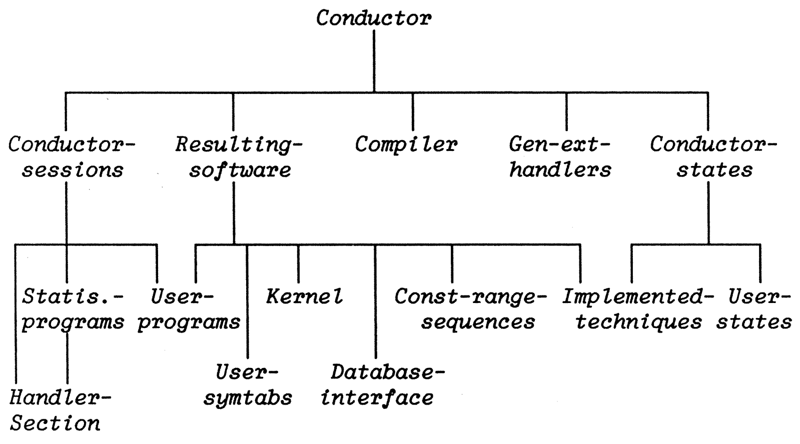
  user-session: USER-PRO -> SESSION

A session may modify the state of CONDUCTOR, as specified in the module *Conductor* in the function *execute*

execute: SESSION # CDT-STATE -> CDT-STATE

The module Conductor specifies the most abstract notion of CONDUCTOR. Details are defined in the modules imported by the module Conductor. Of course, one easily looses track in the modules. The complete specification consists of 114 modules with a total length of 4500 lines. It contains, for example, the formal specification of the abstract syntax of the statistical language, the static symbolic type checking of this language, and the kernel. To get better insight in the relation between the modules, a tree representation is presented, that gives a top-down overview of the import relations between the modules of CONDUCTOR. The tree, given below, shows the import relations between the modules, that specify an abstract notion of CONDUCTOR.

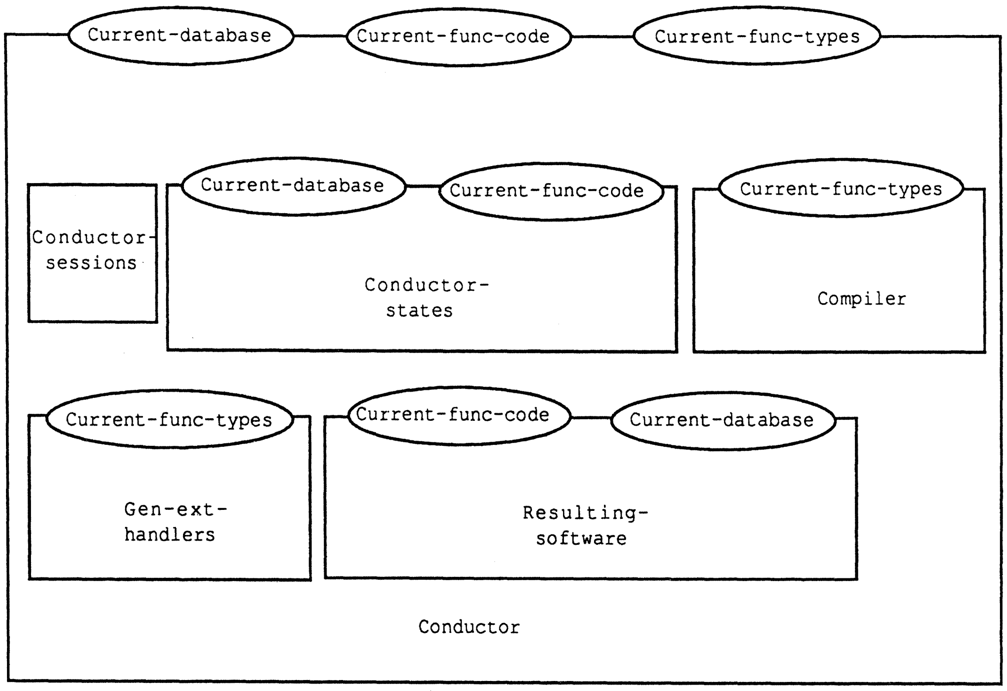
Tree 1. The import structure of CONDUCTOR modules.



The complete tree representation of the system can be found in appendix M. In the trees the information on inherited parameters, and the import of basic modules such as the modules *Booleans* and *Sequences* is omitted. The graphical representation of the modules generated by the ASF specification checker is used as illustration throughout this book. The graphical representation of the module *Conductor* on page 52 illustrates that CONDUCTOR has three parameters: *Current-func-types*, *Current-func-code* and *Current-database*. These parameters are discussed in sections 3 and 4 of this chapter.

6.2. THE CONTRIBUTION OF THE TECHNICAL STATISTICIAN

A technical statistician can implement statistical techniques in CONDUCTOR. He does this by writing a program in the statistical language. CONDUCTOR compiles this statistical program and stores it in the



statistical technique table. A successful session of a technical statistician modifies the state of CONDUCTOR as defined by the following equation for the function *execute* in module *Conductor*:

```
[416]   execute(stat-session(sp),state(ust,stt,eht))
            = state(ust,
                    store-stat-tech(sp,stt),
                    eht)
```

This equation specifies that, given that CONDUCTOR is in a state determined by user state *ust*, statistical technique table *sst* and external handler table *eht*, the execution of a session of a technical statistician results in a state where the statistical program is compiled and stored in the statistical technique table. The tag *[416]* is the number of the equation in appendices.
More details of the compilation and storage process are specified in the imported module *Compiler*. In this module the function *store-stat-tech* is

specified

store-stat-tech: STAT-PRO # STAT-TECH-TABLE -> STAT-TECH-TABLE

with equation

[403]   store-stat-tech(sp,stt) = insert(name(sp),compile(sp),stt)

implying that a statistical program, *sp*, is compiled and stored in the statistical technique table *stt*. The module *Compiler* also contains the specification of the functions *compile* and *name*. These functions are discussed in chapter 11. Of course, all details involved in the compilation and storage of a statistical program, including the function *insert*, are specified in the imported modules of module *Compiler*.

6.3. THE CONTRIBUTION OF THE DATA EXPERT

A data expert has to construct the database that is connected to CONDUCTOR. This is represented, in the formal specification by the parameter *Current-database* of the module *Conductor*. This parameter is inherited from module *Database-interface*. Recall that an inherited parameter is a parameter of an imported module that is not bound in the importing module. The parameter *Current-database* in module *Database-interface* reads:

```
parameters Current-database
   begin
     functions
        current-db:                        -> DATA-BASE
        data-query: DATA-BASE # ID #
                    CONST-RANGE-SEQ        -> USER-DATA
        retrv-data: DATA-BASE # ID #
                    CONST-RANGE-SEQ        -> SCALAR-SEQ
        bg-query:   DATA-BASE # ID #
                    CONST-RANGE-SEQ        -> ID-SEQ
end Current-database
```

This parameter contains the functions *current-db*, *data-query*, *retrv-data* and *bg-query*. The parameters specify, respectively, the database, how

data is retrieved from the data base, and how a background query checks the consistency of the data. The sort *ID* in these functions specifies the name of the series that is to be retrieved, and the sort *CONST-RANGE-SEQ* describes the sample. A background query may result in a sequence of exception identifiers: the sort *ID-SEQ*. A specification of all sorts in the definition of parameter *Current-database* can be found in the modules imported by the module *Database-interface*.
A background query may reveal that a retrieved series is inconsistent. If such an inconsistency is detected an exception identifier is added to the retrieved series. If this series is used in a statistical technique, the accompanying exception is raised and the execution of the statistical technique is interrupted.
The data expert can add external handlers to CONDUCTOR in an external handler session. An external handler session of a data expert modifies the state of CONDUCTOR as defined by the following equation for the function *execute* in module *Conductor*

```
[417]   execute(hand-session(hnd),state(ust,stt,eht))
            = state(ust,
                    stt,
                    store-ext-handler(hnd,eht))
```

Given that CONDUCTOR is in a state determined by user state *ust*, statistical technique table *sst* and external handler table *eht*, the execution of a handler session results in a state where the external handler is compiled and stored in the external handler table. The definitions of the function *store-ext-handler* is given in the module *Gen-ext-handlers*, a module that is imported by the module *Conductor*

6.4. THE CONTRIBUTION OF THE COMPUTER SCIENTIST

The contribution of the computer scientist is, to specify CONDUCTOR, and to make efficient implementations in a higher level programming language of all modules in the specification. This book can be seen as the description of the contribution of the computer scientists in CONDUCTOR as far as the formal specification of the modules concerns. Which functions are available in the statistical language is left unspecified. The type restrictions checks on these functions, as well as the numerical

procedures for the actual calculations, have to be constructed by the computer scientists. For example, functions in the statistical language, such as, matrix inversion, matrix multiplication and Kronecker product are not specified. The fact that functions in the statistical language are left unspecified is represented in the formal specification by unbound parameters *Current-func-types* and *Current-func-code*. These parameters are discussed in respectively chapter 8 and chapter 10.
To handle such exceptions, the computer scientist may, like the data expert, add external exception handlers to CONDUCTOR in an external handler session.

6.5. USING THE IMPLEMENTED STATISTICAL SOFTWARE

A user can apply the statistical techniques implemented by the technical statisticians to the data collected by the data experts in an environment created by computer scientists. A user session modifies the state of CONDUCTOR, as defined by the following equation for the function *execute* in module *Conductor*.

```
[415]   execute(user-session(up),state(ust,stt,eht))
            = state(exec-user-pro(ust,available(sst,eht),up),
                    stt,
                    eht)
```

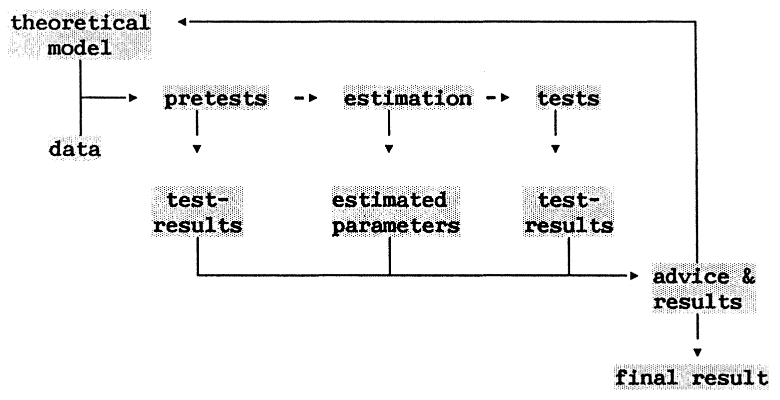
This equation specifies that, given that CONDUCTOR is in a state determined by user state *ust*, statistical technique table *stt* and external handler table *eht*, a user session results in a state where the user program is interpreted and the user state is modified. The definitions of the function *exec-user-pro* is given in the module *Resulting-software*, a module imported by the module *Conductor*.

PART II

FORMAL SPECIFICATION OF THE STATISTICAL LANGUAGE

7. THE SYNTAX OF THE STATISTICAL LANGUAGE

In statistical analysis a theoretical model is postulated that must explain the observed data. If the basic assumptions of this model are not rejected by (pre)tests, the unknown parameters of the model can be estimated. Particular test results and parameter estimations may lead to modification of the postulated model, and thus to a better description of the observed data.



Statistical software made it possible to calculate test results and parameter estimations very fast. Besides positive effects, this also introduced new perils. Trying many variants of the same theoretical model can easily lead to data peeping and chance capitalization (see Lovell [1983] and Meyer [1975]). The contribution of a technical statistician to statistical analysis, therefore, should not only be restricted the creation of tests and estimation techniques for specific theoretical models. A technical statistician should also advice the users of his techniques how to modify a postulated model given the calculated results. The statistical language enables the technical statistician to add his knowledge to statistical software. In this chapter both the concrete and the abstract syntax of the statistical language are discussed. The complete set of production rules of the concrete syntax of the statistical language is given in appendix A. The formal specification of the abstract syntax in appendix F.

7.1. ABSTRACT SETS OF STATISTICAL TECHNIQUES

For the definition of the statistical language one only needs to know what operators, functions and data types are required to describe the statistical techniques a technical statistician want to implement. Knowledge of the statistical technique is not required. In this section we define statistical techniques in terms of these building blocks. This will lead to a few straightforward restrictions on the statistical language.

<u>Definitions</u>:

1. TARGET is the set of n statistical techniques that the statistician intends to express in the statistical language:

   TARGET $= \{TEC_1, \ldots, TEC_n\}$

2. FUNCTIONS is a set of m functions used in the equations of the statistical techniques in the set TARGET.

   FUNCTIONS $= \{F_1, \ldots, F_m\}$

3. OPERATORS is the set of p operators used in the equations of the statistical techniques in the set TARGET.

   OPERATORS $= \{OP_1, \ldots, OP_p\}$

4. TYPES is the set of k data types used in the equations of the statistical techniques in the set TARGET.

   TYPES $= \{T_1, \ldots, T_k\}$

5. SST is the set of all statistical techniques that can be expressed in terms of the sets FUNCTIONS, OPERATOR and TYPES.

$$\mathbf{SST = \{ \; TEC \; | \; g(TYPES,FUNCTIONS,OPERATORS) \rightarrow TEC \; \}}$$

Note that the difference between operators and functions is not fundamental. Operators can be regarded as functions with the appropriate number of parameters. Nevertheless operators are used frequently in statistical textbooks and can not be omitted if one wants to keep close resemblance to statistical notational conventions.

Example 7.1:
Assume we live in 'a simple statistical world' and have the following estimation techniques at our disposal: ordinary least squares (OLS) and two stage least squares (2-SLS). These techniques give an estimation of regression coefficients in a linear (simultaneous) model:

$$\mathbf{y} = \mathbf{Y}_1 \boldsymbol{\beta} + \mathbf{X}_1 \boldsymbol{\tau} + \mathbf{u}$$

Where **y** is a (n x 1)-vector of observations of the endogenous variable that must be explained in the equation, $\mathbf{Y}_1$ is a (n x m)-matrix of the observations of the other endogenous variables in the equation, $\mathbf{X}_1$ is a (n x p)-matrix of the observations of the p exogenous variables in the equation, **u** is a (n x 1)-matrix of random disturbances, $\boldsymbol{\beta}$ and τ are, respectively (m x 1), (p x 1) vectors of model parameters.
To estimate the model parameters the OLS-estimator can be used, however this estimator results in inconsistent estimates of $\boldsymbol{\beta}$ and τ. The OLS-estimator combines the matrix **X1** and **Y1** in the matrix **X**

$$\mathbf{X} = (\; \mathbf{Y}_1 \; | \; \mathbf{X}_1)$$

and estimates $\boldsymbol{\delta} = (\; \boldsymbol{\beta} \; | \; \tau \;)'$ as

$$\mathbf{d}_{\mathbf{olsq}} = (\mathbf{X'X})^{-1}\mathbf{X'Y}$$

whereas the 2-SLS estimator estimates the coefficients in two rounds can be written as (see Maddala [1977])

$$d_{2\text{-}sls} = \begin{bmatrix} (Y1'Y1 - V1'V1) & Y1'X1 \\ X1'Y1 & X1'X1 \end{bmatrix}^{-1} \begin{bmatrix} (Y1-V1)'y \\ X1'y \end{bmatrix}$$

where **V1** is the (n x m)- matrix from the least-squares regression of **Y1** on **X**. The two estimation techniques are the target set of the statistical language.

TARGET = {OLS,2-SLS}

The equations that describe the two techniques in TARGET make use of the following functions, operators and data types:

FUNCTIONS	= **{matrix inversion}**
OPERATORS	= **{matrix subtraction (-),** **matrix multiplication (*),** **matrix transpose ('),** **matrix assignment (=) }**
TYPES	= **{scalar,matrix,submatrix,partitioned matrix}**

Using the sets FUNCTIONS, OPERATORS and TYPES we can also describe a statistical technique called generalized least squares. This technique therefore belongs to the set SST.

SST = **{OLS, 2-SLS, GLS,....}**

{end example 7.1.}

Given the sets FUNCTIONS, OPERATORS and TYPES we can derive the following necessary conditions for the grammar of the statistical language. Recall that the grammar describes the syntax of the language, as discussed in chapter 5. A grammar **G** consisted of a set of non-terminals **N**, a set of terminal symbols **T**, a set of production rules **P**, and a starting symbol **S**. S is one of the non-terminal symbols.

Condition 1: The identifiers of elements of the sets FUNCTIONS, TYPES and OPERATOR are in **T**.

This condition implies that all identifiers (function identifiers, operator symbols and data type identifiers) in the description of the statistical technique must be terminals of the statistical language.

Condition 2: The equations describing the statistical techniques in TARGET are correct sentences in the grammar.

In other words the production rules in the grammar must enable the technical statistician to form the required expressions.

7.2. THE GENERAL STRUCTURE OF A STATISTICAL PROGRAM

In the statistical language the statistician determines:

- *the user interface* of the statistical technique; the user interface consists of the input variables and the resulting statistics of the technique,
- *the equations* that describe the calculation of the statistic
- *the (pre)tests* that check the basic assumptions of the statistical technique and the significance of the calculated parameters,
- *the exception handlers* that describe how the software reacts if an exception occurs (assumptions are violated or problems occur on other levels of CONDUCTOR).
- *a unique name* for the statistical technique.

Each of these parts can be specified in separate sections of a statistical program. A statistical program consists of one or more sections as stated in the following production rules.

```
statistical_program       --> statistical_program section |
                              section

section                   --> name_section |
                              input_output_section |
                              implementation_section |
                              test_section |
                              exception_handler_section
```

The corresponding abstract syntax is specified in the module *Statistical-programs*.

```
module Statistical-programs
begin
   exports
        begin
          sorts STAT-PRO, SECTION
          functions
            abs-prog: STAT-PRO # SECTION -> STAT-PRO
            abs-prog: SECTION            -> STAT-PRO
            abs-sect: ID                 -> SECTION
            abs-sect: IO-SEC             -> SECTION
            abs-sect: IMPL-SEC           -> SECTION
            abs-sect: TEST-SEC           -> SECTION
            abs-sect: HANDL-SEC          -> SECTION
        end

   imports Decl-abstr-syntax, Impl-abstr-syntax
           Test-abstr-syntax, Handler-abstr-syntax,
           Identifiers

end Statistical-programs
```

Note that this module imports several other ones, which define the abstract syntax of separate language constructs. The reader is referred to appendix F for the definition of these modules; they will not further be discussed in this chapter.

7.3. DECLARATION OF VARIABLES

Variables in the statistical language are divided in three categories: input variables, output variables and the internal variables. The input variables are the variables that must be initialized by the user of the statistical technique; the output variables will contain the results of the statistical technique that are returned to the user. The input and output variables are declared in a input/output section of a statistical program. A message string may accompany the declaration of input/output variables. For input variables this message string is used to prompt the user of the statistical technique to enter his data. For output variables this message is clarifies the calculated results. Internal variables can be declared in an implementation, a test and an exception handler section. These variables remain invisible for the user of the statistical technique.
Each declaration is a list of identifiers followed by the type of the declared variables and a message.

```
declaration      --> var_type id_list  message
```

In the current version of CONDUCTOR the data types booleans, scalars, indices and matrices are available. Submatrices, matrix elements and partitioned matrices can also defined. These matrix references, however, are regarded as functions to access particular parts of matrices.

```
var_type         -->  'BOOL' | 'INDEX' | 'SCALAR' |
                       vec_type | mat_type
```

The syntax of vector and matrix types includes the declaration of the dimension bounds.

```
vec_type         -->  'VECTOR' range

mat_type         -->  'MATRIX' '[' ranges ']'

ranges           -->  ranges ',' range |
                      range

range            -->  index_expr  'TO' index_expr
```

The upper and lower bound of a dimension range are expressed in terms of index expressions. These index expressions are somewhat restricted compared to integer expressions in higher-level programming languages in order to make symbolic type and dimension bound checking possible. Introducing a data type integers next to the data type indices relaxes these restriction considerable (See chapters 8 and 9.) The abstract syntax of declarations is specified in module *Decl-abstr-syntax* in appendix F.

Example 7.2: Declarations in the statistical language.

Examples of correct declarations in the statistical language are:

```
legal declarations in a statistical program

SCALAR   R2, Y, D;
BOOL     flag1;
INDEX    N;
MATRIX   [ 1 to N, 1 to N]        X
         MESSAGE: "please enter the X matrix"
```

{ end example 7.2.}

7.4. THE IMPLEMENTATION SECTION

The implementation section of a statistical program contains the equations that describe how the parameters in the theoretical model are calculated. An implementation section consists of internal declarations of variables, and statements.

```
implementation_section     --> internal_declarations
                               'EQUATIONS' statements
```

In CONDUCTOR five types of statements are distinguished.

```
statement                --> assignment |
                             index_assignment |
                             message |
                             for_statement |
                             compound_statement
```

Conditional statements are not included in CONDUCTOR because they would make symbolic evaluation of a statistical program extremely difficult. The assignment statement, index assignment statement and compound statement are identical to similar statements defined in higher level programming languages. The message statement is a rudimentary print statement.
The for-statement in the statistical language is adapted to statistical notational convention. In statistical textbooks the following notation is found frequently

$$X_{i,j} = 0 \qquad i = 1,..,m \qquad j = 1,..,n$$

This is reflected in the following syntax rules

```
   for_statement        --> compound_statement offsets
```

```
offsets              --> offsets ',' offset |
                         offset

offset               --> index_var '='
                         index_expr ',...,' index_expr
```

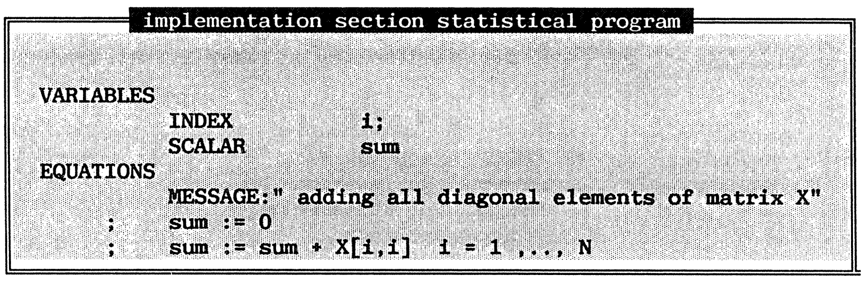
Besides the 'statistical' syntax, also the 'usual' syntax of for-statements in higher-level programming languages is available in the statistical language

```
for_statement        --> 'FOR'index_var ':=' index_expr
                         'TO' index_expr compound_statement
```

Note that, though two different forms of for-statements exist in the concrete syntax, there is only one representation of a for-statement in the abstract syntax.

Example 7.3: an implementation section.

In example 7.1. a matrix X is declared. In the following implementation section all diagonal elements of matrix X are added.



```
implementation section statistical program

VARIABLES
          INDEX         i;
          SCALAR        sum
EQUATIONS
          MESSAGE:" adding all diagonal elements of matrix X"
     ;    sum := 0
     ;    sum := sum + X[i,i]  i = 1 ,.., N
```

{ end example 7.3. }

7.5. THE TEST AND EXCEPTION HANDLER SECTION

Exceptions are a well known phenomenon in computer science, and can occur at every level of the hardware and/or software. Exceptions are unexpected situations which make further execution of a program undesirable or sometimes impossible. Typical examples of exceptions are underflow, overflow and division by zero. These situations are called exceptions

because they were not intended to appear during the normal course of execution of the program.
In the test sections of a statistical program a technical statistician can specify when an exception must be raised. For instance, an exception can be raised if one of the assumptions underlying a statistical technique is rejected by a test: the software created by the technical statistician will signal that there is something wrong with the analyzed data set and interrupts the calculation of the statistical technique.
The syntax of a test in the statistical language is:

```
test_section            -->   'TEST'  internal_declarations
                              'EQUATIONS' statements raises

raises                  -->   raises raise |
                              raise

raise                   -->   'RAISE' exception_flag
                              'WHEN' condition

exception_flag          -->   IDENTIFIER_NAME
```

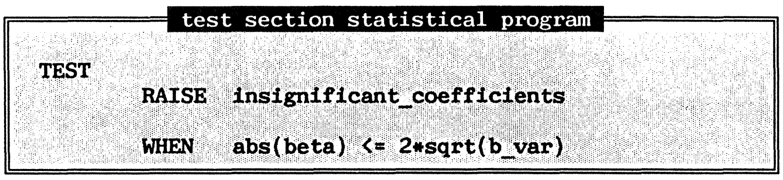
Exception handlers describe how the program must react if an exception is raised. The syntax of an exception handler in the statistical language is:

```
handler                  --> 'WHEN' exception_flag ':'
                             internal_declarations
                             'EQUATIONS' statements
                             stop_or_continue

stop_or_continue         --> 'STOP' |
                             /* default action continue */
```

Exception handlers in CONDUCTOR can be created by the technical statistician in a statistical program. The exception handler is only executed if an exception occurs during the execution of the statistical technique. In the exception handler the technical statistician can decide if the calculation of the statistical technique must be aborted or if it may continue, when an exception occurs. A further discussion of exception handling can be found in chapter 12. The abstract syntax of test and handler sections is specified in modules *Test-abstr-syntax* and *Handler-abstr-syntax* in appendix F.

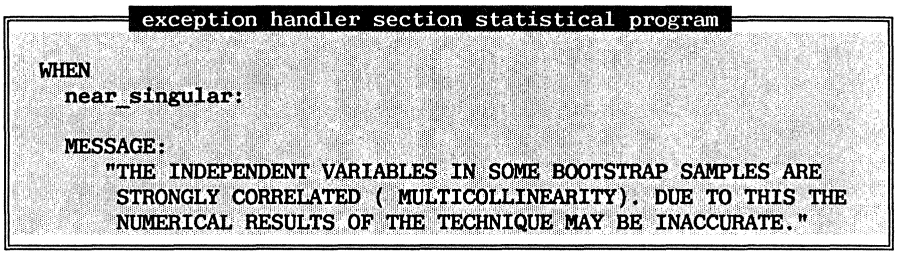
Example 7.4. Tests and exception handlers.

In chapter 4 we described the implementation of a bootstrap method to estimate the regression coefficients in a linear model. In this example we used the 'rule of thumb' to check the significance of the calculated parameters.



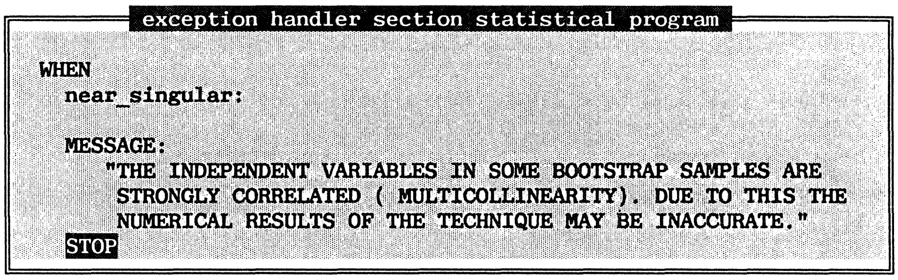
```
test section statistical program

TEST
        RAISE  insignificant_coefficients

        WHEN   abs(beta) <= 2*sqrt(b_var)
```

In that chapter also an exception handler was given for near-singularity.



```
exception handler section statistical program

WHEN
  near_singular:

  MESSAGE:
     "THE INDEPENDENT VARIABLES IN SOME BOOTSTRAP SAMPLES ARE
      STRONGLY CORRELATED ( MULTICOLLINEARITY). DUE TO THIS THE
      NUMERICAL RESULTS OF THE TECHNIQUE MAY BE INACCURATE."
```

This exception handler warns the user, and continues the calculation after the message is displayed, whereas the following handler



```
exception handler section statistical program

WHEN
  near_singular:

  MESSAGE:
     "THE INDEPENDENT VARIABLES IN SOME BOOTSTRAP SAMPLES ARE
      STRONGLY CORRELATED ( MULTICOLLINEARITY). DUE TO THIS THE
      NUMERICAL RESULTS OF THE TECHNIQUE MAY BE INACCURATE."
  STOP
```

displays the same message and then aborts the interrupted statistical technique if near-singularity occurs, because a stop instruction is added to the handler.
{end example 7.4.}

8. SYMBOLIC TYPE CHECKING OF STATISTICAL PROGRAMS

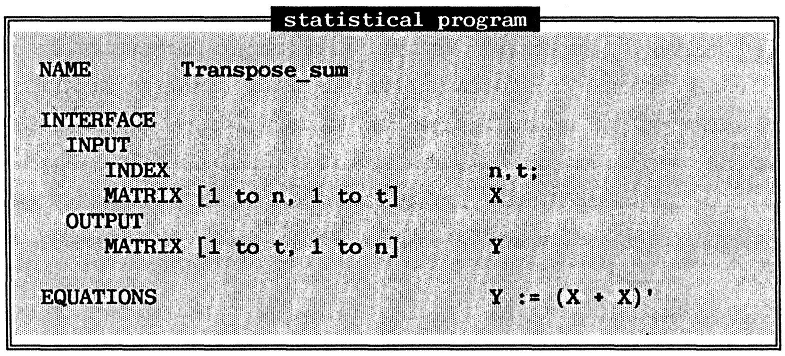
The syntax rules discussed in the previous chapter do not include any restrictions on the type of particular constructs in the language. A matrix function, for example, can be restricted to have only a square matrix as its argument. A type checker verifies that a construct in a language has the type expected by its context. In the early higher-level programming languages type checking was not considered to be a serious problem (see Sheridan [1959]). One of the first languages that allowed the type of language constructs to be evaluated systematically was Algol68. In type checking a distinction is made between *static and dynamic type checking*. If type checking can be done before the program is executed, it is called *static*, whereas it is called *dynamic*, if it is done during the execution of a program. Programming languages that involve matrices, like APL (see Iverson [1962]), rely on dynamic type checking.

Type checking refers to a large class of problems. In the statistical language we particularly are interested in type checking of assignment statements that include matrix variables. Type checking of other constructs in the statistical language is done in exactly the same way as in higher-level programming languages. We concentrate on the restrictions on the dimension bounds of matrix types in assignment statements. The related problem of dimension bound checking in matrix element references is discussed in chapter 9.

Allowing symbolic dimension bounds in the statistical language makes checking the dimension bound restrictions complicated. The dimension bounds of matrices in the statistical language are given in terms of index expressions. A variable representing the number of observations, for example, will often specify the dimension bounds of matrices in a statistical technique. The actual dimension bounds of matrices are determined by the values of the input variables as given by the user of the statistical technique. This user, however, has no knowledge of the implementation details of the statistical technique, such as, constraints imposed by the dimensions of matrices used in the implementation of the statistical technique. He, therefore, should never be confronted with error messages concerning violations of such constraints. To ensure that no errors occur, type and dimension restrictions concerning (matrix) operators and (matrix) functions in the statistical language are checked extensively. Example 8.1 shows some type restrictions that are checked by the type checker.

Example 8.1: Type checking of a statistical program.

Consider the following statistical program.



```
                      statistical program
NAME        Transpose_sum

INTERFACE
  INPUT
     INDEX                          n,t;
     MATRIX [1 to n, 1 to t]        X
  OUTPUT
     MATRIX [1 to t, 1 to n]        Y

EQUATIONS                           Y := (X + X)'
```

The type checker checks if in case of the transpose (')operator that:

- both argument and result are two-dimensional matrices
- the first dimension of the result is equal to the second dimension of the argument
- the second dimension of the result is equal to the first dimension of the argument

And in case of plus (+) and the assign (:=) operator that:

- both arguments are of the same symbolic type

{end example 8.1}

The type checking of the statistical language is based on symbolic evaluation. The symbolic evaluation of a matrix type requires the symbolic evaluation and comparison of the dimension ranges and involved index expressions. The symbolic comparison of index expressions is based on a simple mechanism, that multiplies out an index expression, and combines identical terms and factors. This mechanism is, for example, also applied in the symbolic manipulation languages REDUCE [1973] and MACSYMA [1977].
In this chapter, first the symbolic comparison of index expressions is discussed in section 8.1. In section 8.2 the algebraic specification of this comparison is given and in section 8.3 the symbolic comparison of dimension ranges and matrix types. A symbol table containing the type information of variables declared in a statistical program is specified in section 8.4, and in this section it is shown how declarations in the statistical language are stored in this table. The type information in the symbol table is used in the symbolic type checking of assignment statements in section 8.5. The specification of symbolic type checking of function calls is discussed in section 8.6. A statistical program that satisfies all symbolic type restrictions is called a *type correct statistical program*. The specification of such a program is discussed in section 8.7.

8.1. THE INDEX EXPRESSIONS

The index expressions in the statistical language have a restricted syntax compared to integer expressions in higher-level programming language. The index variables, however, are only intended to be used in the declaration of dimension bounds of matrices, and in matrix element and submatrix references. An index expression may consist of a combination of index variables, index constants and the operators plus, minus and multiplication as stated in the following syntax rules.

```
index_expr          --> '(' index_expr ')' ¦
                    --> index_expr '+' index_expr ¦
                    --> index_expr '-' index_expr ¦
                    --> index_expr '*' index_expr ¦
                    --> INDEX_VARIABLE ¦
                    --> INDEX_VALUE
```

Note that the syntax rules are ambiguous. This ambiguity is easily solved by giving precedence rules for the operators in the parser-generator *yacc* (see Johnson [1975]). The multiply operator (*) has the highest priority of the operators, and is right associative, the plus (+) and minus (-) operators are left associative.
During a parse of an index expression its abstract syntax tree is constructed. The abstract syntax tree of the index expression n*t-1 is, for instance:

```
        -
       / \
      *   1
     / \
    n   t
```

During type checking the need arises to determine whether two index expressions are equivalent[1], i.e. will have the same value when evaluated for any possible combination of values for the variables occurring in them. Unfortunately, equivalent index expressions may have different abstract syntax trees. For example, the equivalent expressions

```
a + (b + c)                (a + b) + c
```

correspond to

```
      +                           +
    /   \                        / \
   a     +                      +   c
        / \                    / \
       b   c                  a   b
```

Another example is given by the expressions

```
a*(b+c)        and        a*b+a*c
```

which are clearly equivalent (due to distributivity) but have non-equivalent abstract syntax trees

[1]. We ignore the problems due to underflow and overflow of integers.

```
       *                    +
      / \                  /   \
     a   +                *     *
        / \              / \   / \
       b   c            a   b a   c
```

To make symbolic comparison of the abstract syntax trees possible they are *"normalized"*. That is, all symbolic equivalent index expressions are represented by the same abstract syntax tree. In CONDUCTOR index expression are normalized using the algorithm 8.1. In the description of this algorithm we use the following definitions:

factor: an index expression consisting of either an index variable or an index constant.

term: a factor or an index expression in which factors are multiplied.

<u>Algorithm 8.1</u>:

This algorithm multiplies out an index expression, and combines similar terms and factors. The algorithm proceed as follows:

1. It distributes multiplication over additions and subtractions.

   ```
   (ie1 + ie2)* ie3  => ie1 * ie3 + ie2 * ie3
   (ie1 - ie2)* ie3  => ie1 * ie3 - ie2 * ie3
   ```

   where ie1, ie2 and ie3 are index expressions.

2. It eliminates the influence of parentheses within + or - operators and within the * operator[2]. (+,-: left associative, *: right associative).

[2] Applying this rule ie2 and ie3 change position in order to keep the rules consistent with the implementation of the algorithm where the trees are rebuilt in the following way

```
    +                  +
   / \        =>      / \
ie1   -              -    ie2
     / \            / \
  ie2  ie3       ie1  ie3
```

```
ie1 + (ie2 + ie3) => ie1 + ie3 + ie2
ie1 - (ie2 - ie3) => ie1 + ie3 + ie2
ie1 + (ie2 - ie3) => ie1 - ie3 + ie2
ie1 - (ie2 + ie3) => ie1 - ie3 - ie2

ie1 * ie2 * ie3   => ie1 * (ie2 * ie3)
```

where ie1, ie2 and ie3 are index expressions.

3. It sorts factors and multiplies constant factors within terms,

   factor1 * factor2 => factor2 * factor1

   when factor2 $<_f$ factor1, and where $<_f$ is a lexicographical order on the factors (constants $<_f$ variables) and

   c1 * c2 * term   => c3 * term   (c3 = c1 * c2)

   where c1, c2 and c3 are constant factors.

4. It sorts terms and adds/subtracts identical terms.

   term1 + term2 => term2 + term1

   when term2 $<_t$ term1, where $<_t$ is a lexicographical order on the terms (constants $<_t$ variables) and

   c1 * term + c2 * term => c3 * term   (c3 = c1 * c2)

   where c1, c2 and c3 are constant factors.

5. It adjusts for negative signs in constant factors in terms.

```
-(-c) * term  =>   c * term
+(-c) * term  => - c * term
```

Example 8.2: Normalization of the abstract syntax tree of an index expression.

Consider the following index expression

```
(a+3)*(a-6)
```

The abstract syntax tree of this index expression is

```
          *
       /     \
      +       -
    /   \   /   \
   a     3 a     6
```

Applying the algorithm gives

Step 1: distribution of * over +/- :

```
(a+3)*(a-6) => (a*(a-6)) + (3*(a-6))
            => ((a-6)*a) + ((a-6)*3))
            => (a*a-6*a) + (a*3-6*3)
```

resulting in the tree

```
                +
          /           \
       -                  -
   /       \          /       \
  *         *        *         *
 / \       / \      / \       / \
a   a     6   a    a   3     6   3
```

Step 2: remove influence of parentheses:

```
(a*a-6*a) + (a*3-6*3) => ((a*a-6*a)-6*3)+a*3
                      => a*a - 6*a - 6*3 + a*3
```

resulting in the tree

```
                +
              /   \
             -      *
        /      \   / \
       -        *  a  3
    /    \     / \
   *      *   6   3
  / \    / \
 a   a  6   a
```

Step 3: sort factors and multiply constant factors within terms:

a*a - 6*a - 6*3 + a*3 => a*a - 6*a - 18 + 3*a

resulting in the tree

```
                 +
               /   \
              -     *
           /    \  / \
          -     18 3  a
       /    \
      *      *
     / \    / \
    a   a  6   a
```

Step 4: Sort terms and add/subtract identical terms:

```
a*a - 6*a - 18 + 3*a  =>  a*a - 6*a + 3*a - 18
                      =>  a*a + (-3)*a - 18
```

resulting in the tree

```
              -
           /    \
          +     18
       /    \
      *      *
     / \    / \
    a   a  -3  a
```

Step 5: Adjust for negative signs:

a*a +(-3)*a - 18 => a*a-3*a-18

resulting in the tree:

```
              -
           /    \
          -     18
       /    \
      *      *
     / \    / \
    a   a  3   a
```

Equivalent index expressions are always represented in CONDUCTOR by the same abstract syntax tree. An example of an equivalent index expression is a*a- 3*(a-6).
{end example 8.2.}

8.2. ALGEBRAIC SPECIFICATION OF INDEX EXPRESSIONS

The abstract syntax trees of index expressions are specified in module *Ind-expr-abstr-syntax*.

```
module Ind-expr-abstr-syntax
begin
   exports
     begin
       sorts IND-EXPR
       functions
          abs-plus:      IND-EXPR # IND-EXPR       -> IND-EXPR
          abs-minus:     IND-EXPR # IND-EXPR       -> IND-EXPR
          abs-mul:       IND-EXPR # IND-EXPR       -> IND-EXPR
          abs-const:     INDEX                     -> IND-EXPR
          abs-ind:       ID                        -> IND-EXPR
          null:                                    -> IND-EXPR
          eq-result:     IND-EXPR # IND-EXPR -> BOOL
     end
   imports Indices, Identifiers, Booleans

end  Ind-expr-abstr-syntax
```

Equivalence classes of index expressions are defined by equations in module *Ind-expr-abstr-syntax*. Each step in the normalization algorithm requires that the functions in the module *Ind-expr-abstr-syntax* satisfy certain equations.

Step 1. *Distributivity* of the multiply operator over the add and
minus operator.

```
[59]      eq-result(abs-mul(e1,abs-plus(e2,e3)),
                   abs-plus(abs-mul(e1,e2),abs-mul(e1,e3)))
                 = true
```

and

```
[60]      eq-result(abs-mul(e1,abs-minus(e2,e3)),
                   abs-minus(abs-mul(e1,e2),abs-mul(e1,e3)))
                 = true
```

Step 2. *Associativity* of the plus, minus and multiply operator.

```
[61]      eq-result(abs-plus(e1,abs-plus(e2,e3)),
                   abs-plus(abs-plus(e1,e2),e3))
                 = true
```

and

```
[62]      eq-result(abs-minus(e1,abs-minus(e2,e3)),
                    abs-minus(abs-minus(e1,e2),e3))
                = true
```

and

```
[63]      eq-result(abs-mul(e1,abs-mul(e2,e3)),
                    abs-mul(abs-mul(e1,e2),e3))
                = true
```

The other steps in the algorithm require that:

- the *multiplication* of indices can be applied to the constant factors in one term,
- the multiply operator is *associative* and *commutative*,
- the plus and minus operator are *associative* and *commutative*, and the *arithmetic rules* of indices must be applicable to *add/subtract* terms,
- subtracting a negative term is equal to adding the negation of that term.

A complete specification of these requirements is given in module *Ind-expr-abstr-syntax* in appendix E. The equations in this module define the equivalence relation on index expressions.

8.3. SYMBOLIC EQUIVALENCE OF DIMENSION RANGES AND DATA TYPES

A dimension range consists of a pair of index expressions. This pair describes the lower and upper bound of the range. The symbolic equivalence of dimension ranges is specified by straightforward application of the symbolic equivalence relation on index expressions

```
     range:    INDEX-EXPR # INDEX-EXPR  -> RANGE
     eq-range: RANGE # RANGE            -> BOOL
```

with equation

```
[82]  eq-range(range(e1,e2),range(e3,e4))
            = eq-result(abs-minus(e2,e1),abs-minus(e4,e3))
```

In words, two ranges are symbolic equivalent if the difference between the upper bound and lower bound of the two ranges are symbolic equivalent.

A matrix in the statistical language may have more than one dimension. A sequence of dimension ranges is specified in the module *Range-sequences*.

This module is imported in the module *Tech-types*, where the type descriptions of the variables in the statistical language are specified.

```
module Technique-types
begin
   exports
     begin
       sorts TECH-TYPE
       functions
          tech-type:    SIMPLE-TYPE   -> TECH-TYPE
          matrix-type: RANGE-SEQ      -> TECH-TYPE

          eq-type:      TECH-TYPE # TECH-TYPE   -> BOOL
     end

   imports Range-sequences, Booleans

end Technique-types
```

The type of a variable in the statistical language, is either a simple type (boolean, index or scalar) or a matrix type. Symbolic equivalence of type descriptions is specified in the function *eq-type*. Two matrix type descriptions are symbolic equivalent when the dimension ranges (*rs1* and *rs2*) are symbolic equivalent.

```
[90]   eq-type(matrix-type(rs1),matrix-type(rs2))
                = eq-ranges(rs1,rs2)
```

A complete specification of the function *eq-type* is found in appendix E.

<u>Example 8.4</u>: Comparison of matrix-types.

Let the INPUT section of a statistical program contain the following variable declarations:

```
declarations stat. program

INDEX                  k,n,m,p;
VECTOR [n+p TO n+k]    X;
VECTOR [m+p TO m+k]    Y;
```

To simplify the example we assume that in the formal specification the index variables **k**, **n**, **m** and **p** are represented by the same identifier. Note that in general the identifiers in the specification are different from the identifiers in the program.

```
k,n,m,p         :-> ID
```

Using the functions in the formal specification the type description of the variable X reads

```
t1 = matrix-type(
        add-item(range(abs-plus(abs-ind(n),abs-ind(p)),
                       abs-plus(abs-ind(n),abs-ind(k))),
                 null-range))
```

and the type description of variable Y reads

```
t2= matrix-type(
        add-item(range(abs-plus(abs-ind(m),abs-ind(p),
                       abs-plus(abs-ind(m),abs-ind(k)),
                 null-range))
```

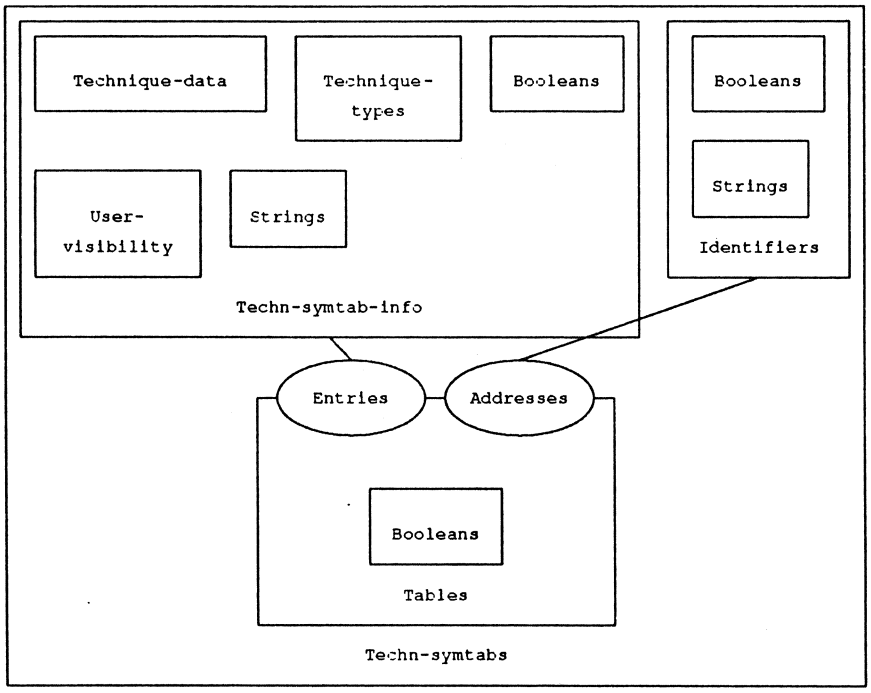
The two type descriptions t1 and t2 are equivalent as the following derivation shows:

```
eq-type(t1,t2) =

eq-ranges(add-item(range(abs-plus(abs-ind(n),abs-ind(p),
                         abs-plus(abs-ind(n),abs-ind(k)),
                   null-range),
          add-item(range(abs-plus(abs-ind(m),abs-ind(p),
                         abs-plus(abs-ind(m),abs-ind(k)),
                   null-range))
                 =
and(
   eq-range(range(abs-plus(abs-ind(n),abs-ind(p),
                  abs-plus(abs-ind(n),abs-ind(k)),
            range(abs-plus(abs-ind(m),abs-ind(p),
                  abs-plus(abs-ind(m),abs-ind(k))),
   eq-ranges(null-range,null-range))
                 =
eq-range(range(abs-plus(abs-ind(n),abs-ind(p),
               abs-plus(abs-ind(n),abs-ind(k)),
         range(abs-plus(abs-ind(m),abs-ind(p),
               abs-plus(abs-ind(m),abs-ind(k)))
                 =
eq-result(abs-minus(abs-plus(abs-ind(n),abs-ind(p)),
                    abs-plus(abs-ind(n),abs-ind(k)))),
          abs-minus(abs-plus(abs-ind(m),abs-ind(p)),
                    abs-plus(abs-ind(m),abs-ind(k))))
                 =
eq-result(abs-minus(abs-ind(p),abs-ind(k)),
          abs-minus(abs-ind(p),abs-ind(k)))
                 =
true
```

{ end example 8.4. }

8.4. MANAGING TYPE INFORMATION DURING TYPE CHECKING

In order to manage the type information obtained during type checking we introduce a symbol table that relates variables in a statistical program with their type. This table is called the technique symbol table. For each variable the data type description (the sort *TECH-TYPE*), the visibility (the sort *USER-VIEW*), the value (the sort *TECH-DATA*) and a message for the user (the sort *STRING*) is stored. The information for each variable in the symbol table can be located by the variable identifier (the sort *ID*).



The variables in the declaration sections of a statistical program are stored in the technique symbol table. The module *Store-declarations* imports the abstract syntax of declarations in the statistical language from the module Decl-abstr-syntax, and imports the technique symbol tables from the module *Tech-symtabs*. The function *store-id* specifies that the information in a variable declaration, the identifier (*id*), the type description (*tt*), the user view (*uv*) and the message (*mess*), is inserted in the technique symbol table (*ts*). A variable is not assigned a value when it is declared. This is specified by the constant function *uninitialized*.

```
[118] store-id(ts,id,uv,mess)
          = insert(id,
                   ts-info(tt,uv,uninitialized,mess),
                   ts)
```

8.5. TYPE CHECKING OF ASSIGNMENT STATEMENTS

Symbolic type checks are performed for all assignment statements in a statistical program. The other statements do not, or only indirectly, impose type restriction. The function *type-check-stmnt*,

```
     type-check-stmnt:  TECH-SYMTAB # STATEMENT -> BOOL
```

with equation

```
[156]  type-check-stmnt(abs-assgn(var,expr),tst)
            =  eq-type(expr-type(expr,tst),var-type(var,tst))
```

specifies the type restrictions on statements. The type of the variable (*var*) in the left-hand-side of the assignment statement is determined by the function *var-type(var,tst)*. This function retrieves the type of the variable from the technique symbol table *tst*. The type of the expression on the right-hand-side of an assignment statement is specified in the function *expr-type* in module *Stat-check-expressions*

```
     expr-type: EXPR # TECHN-SYMTAB -> TECH-TYPE
```

If, for example, an expression only consists of a scalar (*sca*) its resulting type is scalar type, as expressed in the equation

```
[150]  expr-type(abs-expr(t-data(sca)),tst) = tech-type(scalar-type)
```

where *tst* is a technique symbol table. The specification of the type of other expressions can be found in module *Stat-check-expressions* in appendix H.

8.6. TYPE CHECKING OF FUNCTION CALLS

Type checking of functions calls in the statistical language is organized around the notion of a type list. The type list contains the types of the function arguments. The types in the type list are used to check the type restrictions in the function table. The latter captures the type information for each function in the statistical language. Type restrictions on functions describe the required types of the result and the argument(s) of a function. The symbolic type checker in CONDUCTOR checks two sorts of type restrictions:

(1) *skeleton restrictions*
(2) *dimension restrictions*

To check the first class of restrictions the function *eq-skelet* is specified in the module *Tech-types*.

```
eq-skelet: TECH-TYPE # TECH-TYPE      -> BOOL
```

In this function the equivalence of matrix-types is less restrictive than in the function *eq-type*. The function *eq-skelet* only checks if the number of dimensions of the matrix-types, with range-sequences *r1* and *r2*, are equal,

```
[94]  eq-skelet(matrix-type(r1),matrix-type(r2))
             = eq(n-of-dims(r1),n-of-dims(r2))
```

whereas the function *eq-type* checks if the dimension ranges are equivalent.
A skeleton restriction consists of two indices, indicating which arguments in the type list must satisfy the skeleton restrictions, or an index and a type description where the indicated element in the type list must be equal to the given type.

```
restr: INDEX # INDEX  -> SKELET-RESTR
restr: INDEX # TYPE   -> SKELET-RESTR
```

A check of a skeleton restriction is specified in function *check*

```
check: SKELET-RESTR # TYPE-LIST -> BOOL
```

with equations

```
[124]   check(restr(ind1,ind2),tl)
                = eq-skelet(argument(tl,ind1),argument(tl,ind2))

[125]   check(restr(ind,t),tl)
                = eq-skelet(argument(tl,ind1),t)
```

A dimension restriction imposes an equality restriction or a constant restriction on particular dimension ranges of arguments. A dimension range of an argument is indicated in the type list by a constant range. The first element of the constant range specifies the argument number, the second index the dimension number. An equality restriction on two dimension ranges consists of two constant ranges; a constant restriction on a dimension range consist of a constant range and a dimension range.

```
dim-restr: CONST-RANGE # CONST-RANGE  -> DIM-RESTR
dim-restr: CONST-RANGE # RANGE        -> DIM-RESTR
```

The check of dimension restrictions

```
check:      DIM-RESTR # TYPE-LIST     -> BOOL
```

checks if the restriction is satisfied for a particular type list. If the dimension restriction consists of two constant ranges *cr1* and *cr2*, these constant ranges are used to retrieve the appropriate dimension ranges from the type list *tl* and these dimension ranges are compared.

```
[132] check(dim-restr(cr1,cr2),tl)
            = eq-range(arg-range(cr1,tl),arg-range(cr2,tl))
```

The type of the result of a function is specified to be either a simple type or a matrix type. In the latter case the dimension ranges are retrieved from the type list.

```
res-type: TYPE              ->RESULT-TYPE
res-type: CONST-RANGE-SEQ -> RESULT-TYPE
```

How the result is defined in term of the argument types in the type list is specified in module *Result-types* in appendix H. Example 8.5 demonstrates a set of symbolic type restrictions that can be checked by the type checker in CONDUCTOR.

Example 8.5: Type restrictions on functions.

Assume we impose the following restrictions on a function:

1. Skeleton restrictions on arguments:
   - the first argument must be a scalar.
   - the second argument must be a two dimensional matrix.
2. Dimension restrictions on arguments:
   - the first dimension of the second argument must be equal to the second dimension of the second argument.
3. The type of the result of the function:
   - the result is a one dimensional matrix.
   - the dimension range of the result is equal to the first dimension range of the second argument.

In terms of the functions in the formal specification the skeleton restrictions read[3]

```
restr(1,tech-type(scalar-type))
restr(2,matrix(add-item(r1,add-item(r2,null-range)))
```

where *r1* and *r2* are ranges. The symbolic dimension restrictions read

```
dim-restr(c-range(1,2),c-range(2,2))
```

The result is specified as

```
res-type(add-item(cr-range(2,1),null-cr-range))
```

{end example 8.5.}

The type restrictions on functions are sequences of skeleton restrictions and dimension restriction. A function *check* specifies the checking of all skeleton restrictions.

```
check: SKELET-RESTR-SEQ # TYPE-LIST -> BOOL
```

with equations

[3] Note that 2 does not exist in the formal specification and should be written as increm(1).

```
[126] check(add-item(sr,null-seq),tl) = check(sr,tl)
```

and

```
[127] check(add-item(sr,srs),tl)
          = and(check(sr,tl),check(srs,tl)
```

where *sr* is a skeleton restriction and *srs* is a sequence of skeleton restrictions. In a similar way, module *Dim-restrictions* specifies the check of dimension restrictions (the sort *DIM-RESTR-SEQ*) in appendix H.
All type information concerning functions is combined in the sort *FUNC-TYPE-INFO* in module *Function-types*

```
    f-info: INDEX # SKELET-RESTR-SEQ #
            DIM-RESTR-SEQ # TECH-TYPE    -> FUNC-TYPE-INFO
```

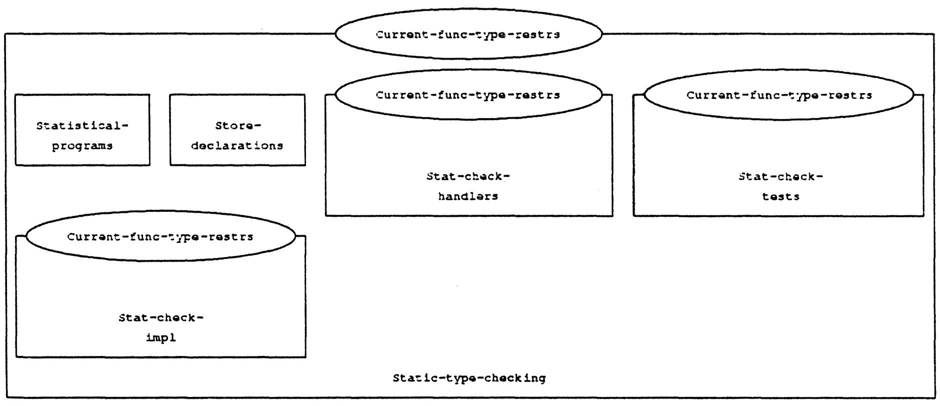
The type information is stored in the function type table. The type of a function call in the statistical langauge, with function identifier *f-id* and argument list *argl*, can now be specified as:

```
[153] expr-type(abs-f-call(f-id,argl),tst)
          = result-f(f-id,arg-types(argl,null,tst))

    when check-f-restr(f-id,arg-types(argl,null,tst)) = true
```

The function *check-f-restr* defines the checking of both the skeleton and dimension restrictions; the function *result-f* returns the type of the result of the function call in the statistical language. The type of the arguments is evaluated by the function *arg-types, using information in the symbol table ts.*
The above mechanism only describes equality and constant restrictions on dimension ranges of matrix types. More complicated restriction on these dimension ranges can be created using the range expressions specified in module *Ranges*. Range expressions will not always yield correct ranges. These problems, caused by the lack of order on the equivalence classes of index expressions, are discussed in chapter 9.

8.7. A TYPE CORRECT STATISTICAL PROGRAM

In chapter 7 we specified a syntactically correct statistical program. Type checking of a statistical program, using the mechanisms discussed in this chapter, gives the next step in the definition of a correct statistical program: *a type correct statistical program*. The symbolic type checking of a statistical program is specified in module *Static-type-checking*.



The parameter *Current-func-type-restrs* is left unbound is this module. This specifies that the design of CONDUCTOR is independent of the functions defined in the statistical language.

9. SYMBOLIC DIMENSION BOUND CHECKS AND THE GENERATION OF INPUT RESTRICTIONS

In a correct statistical program, matrix element and submatrix references may never refer to elements outside the declared dimensions of a matrix. Matrix element reference can be compared with references to elements of a dynamic array in a higher-level programming language program. In practice it is difficult to check that such references are correct for all possible values of the input variables of such a program. A higher-level programming language program may have infinitely many execution trees, i.e. there are infinitely many ways to evaluate a program written in these languages. In practice, therefore, one often has to rely on testing instead of proving the correctness of a program. A way to test a program with infinitely many execution trees is called symbolic evaluation (see King [1976]). For each variable, after each statement, a symbolic denotation of the value is given and for each path in the program a conditional expression is calculated, indicating under which condition the path is taken. King [1976] suggests giving a symbolic value to the input variables and shows that, for a simple programming language, EFFIGY, symbolic evaluation is possible. The mechanism suggested by King, is used in CONDUCTOR to generate input restrictions on the values of the input variables of a statistical program.
The symbolic evaluation of a statistical program is facilitated by the fact that the structure of the statistical language in CONDUCTOR is less complicated than the structure of higher-level programming languages. Important differences in comparison with higher-level programming languages are:

- the language does not have conditional statements,
- the only loop control statement is the for-statement,
- a for-loop is at least executed once,
- loop control variables may not be reassigned inside a loop,
- the input index variables in the statistical program may not be reassigned inside a statistical program,
- index expressions are a subset of general integer expressions.

Due to these restrictions a statistical program has a *unique symbolic execution tree*, i.e. a symbolic range can be calculated for each index variable after each statement in the program. This symbolic range describes the upper and lower bound of the index variable in terms of the input index variables of the statistical technique.

The symbolic evaluation of a statistical program consists of the following steps:

- symbolic ranges are assigned to input index variables,
- symbolic ranges are assigned to index variables in for-statements and index assignment statements,
- for each index expression a symbolic range is calculated using the symbolic ranges of the index variables in the expression,
- the symbolic ranges of index expressions in a matrix element reference are compared with the symbolic dimension ranges of the referenced matrix; if these symbolic ranges are not equivalent, input restrictions on the values of the input index variables are generated.

Consider the following statistical program

statistical program

```
INTERFACE
      INPUT
            INDEX                       n,m,p;
            VECTOR [n+m TO p+m]         X

VARIABLES   INDEX                       i

EQUATIONS   X := 0;
            X[i+m] := 1/m               i := n,..,p
```

In this example, first the symbolic ranges **[n TO n]**, **[m TO m]** and **[p TO p]**, are assigned to the input variables **n**, **m** and **p**. Then a symbolic range **[n TO p]** is assigned to the index variable **i**. And finally a symbolic

range **[n+m TO p+m]** is calculated for the expression **i+m,**, by adding the bounds of the symbolic ranges of **i** and **m**. This range is equivalent to the symbolic dimension range of the vector **X**, showing that the element reference **X[i+m]** is correct for all possible values of the input variables.
The symbolic ranges for an index expression are calculated by applying the operator in the index expression to the bounds of the symbolic ranges of the operands. The example above, of course, obscures the problems that may occur in less trivial index expressions. Symbolic ranges can only be calculated with the range calculation mechanism for monotone increasing or decreasing index expressions. And as an additional restriction all symbolic ranges in the symbolic calculations must be strictly positive.

9.1. AN ORDERING ON INDEX EXPRESSIONS

In the symbolic dimension check mechanism, it has to be determined whether the symbolic range of an index expression is a subrange of the declared symbolic range. For example, the vector **X**, declared in the statistical program in the beginning of this chapter, has dimension bounds **[n + m TO p + m]**. For the index expression **i+m**, in the vector element reference **X[i+m]**, we calculated the same range. And we had to determine whether the restriction

$$n+m \leq n+m \leq n+p \leq n+p$$

holds for all possible values of the variables in the expressions. Thereto, we have to define an ordering on the index expressions. This ordering is based on the ordering of integers. If two expressions always yield the same integer value after evaluation, for all possible values of the variables in these expressions, they are equivalent. If one index expression always yields a greater/smaller integer than another index expression, the order is greater/smaller. The order of two expressions is said to be undecided if the comparison is undecided. The order on index expressions is specified in module *Ind-expr-order*.

```
module Ind-expr-order
begin
     exports
       begin
         functions
           order:     IND-EXPR # IND-EXPR  -> RELATION
       end

     imports Order, Ind-expr-abstr-syntax

end Ind-expr-order
```

The function *order* specifies the order relation of two index expressions, *ie1* and *ie2*, based on the order relation on the integers. If the difference between two index expression is a constant *c*, it can be decided if the order relation between the two expressions is equal, greater or less, by comparing the constant *c* with *0*. If *c* is equal to *0* the boolean function *eq(c,0)* returns the value *true*, if *c* is greater than *0*, the boolean function *ge(c,0)* returns the value true.

```
[190]   order(ie1,ie2) = if (eq(c,0), equal,
                                     if(ge(c,0), greater,less)

        when eq-result(abs-minus(ie2,ie1),abs-const(c)) = true
```

If the difference between two index expressions is not a constant, the expressions are not comparable, and the order is said to be undecided.

```
[191]   order(ie1,ie2) = undecided

        when eq-result(abs-minus(ie2,ie1),abs-const(c)) = false
```

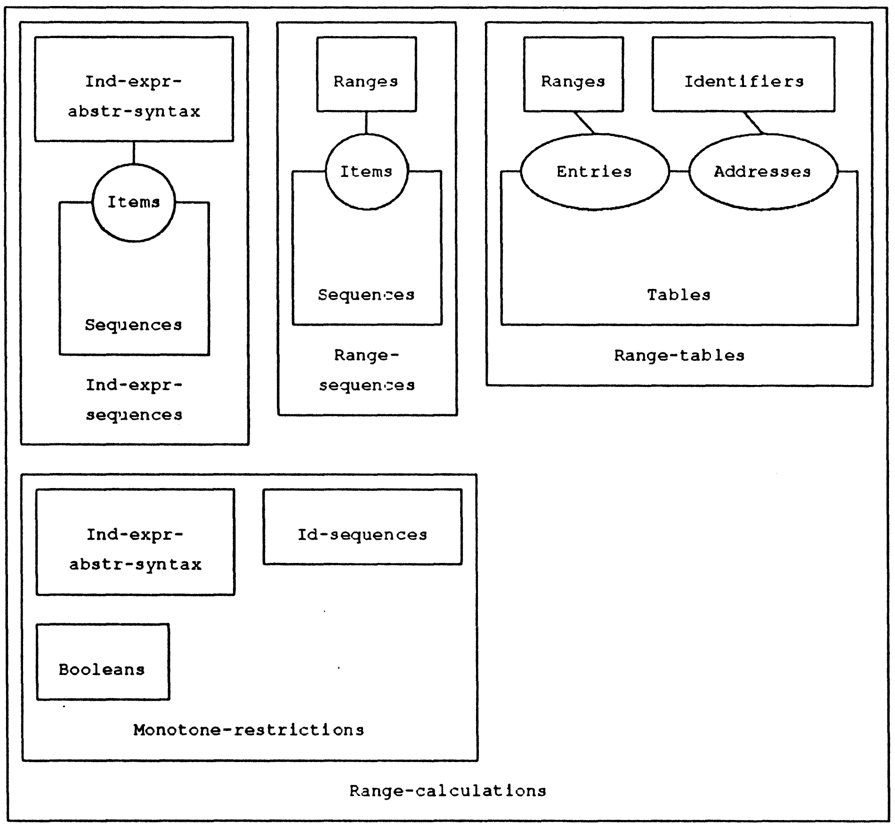
If the difference between two index expressions consist of the sum of quadratic terms, also the order relation can be determined. If all quadratic terms are positive, the order relation is greater and if all quadratic terms are negative, the order relation is smaller. If the resulting difference is a mixture of positive and negative quadratic terms, the order is again undecided. In the formal specification of the function *order* in appendix I, the order relation based on quadratic terms is not incorporated, it can however easily be added.

9.2. CALCULATION OF A SYMBOLIC RANGE FOR AN INDEX EXPRESSION

For each index expression in a statistical program a symbolic range is calculated during the symbolic evaluation. A symbolic range for an index expression is calculated by applying the operator in the index expression to the bounds of the symbolic ranges calculated for the operands.
The range operator $+_r$, $-_r$ and $*_r$ are defined as follows:

$$[n \text{ to } m] \; +_r \; [p \text{ to } q] \; = \; [(n + p) \text{ to } (m + q)]$$
$$[n \text{ to } m] \; -_r \; [p \text{ to } q] \; = \; [(n - q) \text{ to } (m - p)]$$
$$[n \text{ to } m] \; *_r \; [p \text{ to } q] \; = \; [(n * p) \text{ to } (m * q)]$$

An index expression that only consists of a variable (x) or constant (c) the function range results in a range with the variable or constant as lower and upper bound. The calculation of symbolic ranges is specified in the function *calc-range* in module *Range-calculations.*



For instance, the equation *[205]* specifies the calculation of a range for the plus operator in an index expressions.

```
[205] calc-range(abs-plus(ie1,ie2),rt)
            = range-plus(calc-range(ie1,rt),
                         calc-range(ie2,rt))

              when is-monotone(abs-plus(ie1,ie2)) = true
```

Where *ie1* and *ie2* are index expression and *rt* is a range table. This equation states that if we want to calculate the symbolic range of an expression that is the sum of two expressions, we first calculate the symbolic ranges for the two sub-expressions (*ie1* and *ie2*) and than add these two symbolic ranges. The restriction that the index expressions must be monotone increasing or decreasing is discussed in the next section of this chapter. The addition of two symbolic ranges is specified in module Ranges in equation *[77]*. The addition of two ranges, *range(ie1,ie2)* and *range(ie3,ie4)* results in a new range where the lower and upper bounds are added.

```
[77] range-plus(range(ie1,ie2),range(ie3,ie4))
           = range(abs-plus(ie1,ie3),abs-plus(ie2,ie4))
```

Similar equations are given for multiplication and subtraction.
Ranges for index expressions can only be calculated if the intermediate ranges in the calculations are strictly positive. If this can not be verified, input restrictions are generated as specified in module *Range-calc-restr*. For example, the restriction that a range retrieved from the range table *(id^ts)* must always be positive is specified in function *restr-calc-range*.

```
   restr-calc-range: IND-EXPR # RANGE-TABLE #
                     TECH-SYMTAB # INP-RESTR-SEQ ->INP-RESTR-SEQ


[218]  restr-calc-range(abs-ind(id),rt,ts,irs)
            = conc(pos-range-restr(id^rt,ts),irs)
```

The function *conc* concatenates the generated restrictions and the already existing input restrictions *irs*.

9.3. RESTRICTIONS ON INDEX EXPRESSIONS

A symbolic range for an index expression is calculated by applying the operator in the index expression to the bounds of the symbolic ranges of the operands. This mechanism can only be applied to positive monotone increasing or decreasing index expressions. Furthermore all ranges in the calculations should be positive.

Example 9.1: A not monotone increasing index expression.

Consider the following simple index expression

$$p(i) = i*i - 4*i + 4$$

This is expression is neither monotone increasing nor monotone decreasing.

$$\frac{\delta}{\delta i}\, p(i) > 0 \qquad i > 2\ ,$$

$$\frac{\delta}{\delta i}\, p(i) < 0 \qquad i < 2\ ,$$

Assume the variable **i** is assigned the symbolic range **[n to p]**. The symbolic range calculation mechanism will first assign a range to the variables and constants in the expression, then it will use these ranges to assign ranges to the terms **i*i**, **4*i** and **4**, etc.

$$\begin{tikzcd}
{[n*n \text{ to } p*p]} \arrow[d, Rightarrow] & {[4*n \text{ to } 4*p]} \arrow[dl, Rightarrow] & {[4 \text{ to } 4]} \arrow[ddl, Rightarrow] \\
{[n*n-4*p \text{ to } p*p-4*n]} \arrow[dr, Rightarrow] & & \\
 & {[n*n-4*p+4 \text{ to } p*p-4*n+4]} &
\end{tikzcd}$$

The calculated range is obviously wrong. Substitution of **n = 1** and **p = 10** leads to a calculated range of -35 to 100, while the actual range of this quadratic function is 0 to 64.
{end example 9.1.}

For monotone increasing expression it can easily be shown that the range calculation mechanism is correct. A normalized index expressions (see algorithm 8.1 in chapter 8) is a polynomial in the integer variables $x_1,..,x_m$. Calculating the upper bound of a normalized index expression is equal to finding the maximum of the polynomial for the variables $x_1,..,x_m$. For the variable x_1 this reads

$$\max_{x_1,..,x_m} \sum_{i=1}^{n} c_i \, (x_1)^i \, f_i(x_2,...,x_m)$$

where c_i is the constant factor in a term of the polynomial, and f_i a function of the other variables in each term. The function f_i is a product of powers of the variables $x_2,..,x_m$. All variables in f_i are positive

$$f_i(x_2,...,x_m) >= 0$$

The variable x_1 is also restricted to a positive range, with maximum $\max(x_1)$ and minimum $\min(x_1)$

$$0 <= \min(x_1) <= x_1 <= \max(x_1)$$

If also $c_i >= 0$, the partial derivative of the polynomial with respect to x_1 is positive for all values of $x_1,..,x_m$, and we may simply substitute the maximum of x_1 for every occurrence of x_1 to get the maximum of the polynomial with respect to x_1.

$$\max_{x_2,..,x_m} \sum_{i=1}^{n} c_i \, (\max(x_1))^i \, f_i(x_2,...,x_m)$$

It can also be shown that, for strictly negative terms, where $c_i <= 0$, one may substitute the minimum of x_1 to get the maximum of the polynomial. This shows that we can use the symbolic evaluation mechanism under the restriction that a given polynomial p can be written as the sum of two polynomials p_1 and p_2

$$p(x_1,..,x_m,y_1,..,y_t) = p_1(x_1,..,x_m) + p_2(y_1,..,y_t)$$

where the function p_1 only contains the positive terms of polynomial p

and p_2 the negative terms. The sets of variables $x_1,\ldots,x_m$ and $y_1,\ldots,y_t$ must be disjoint. The check whether an index expression is monotone increasing or decreasing, amounts to a check whether the set of variables in negative terms and the set of variables in positive terms of a normalized index expression is disjoint. The monotonicity restriction is specified by the function *is-monotone* in module *Monotone-restrictions*.

```
module Monotone-restrictions
begin
   exports
       begin
           functions
            is-monotone: IND-EXPR                      -> BOOL
            var-list: BOOL # BOOL # IND-EXPR # ID-SEQ -> ID-SEQ
       end

   imports Ind-expr-abstr-syntax, Id-sequences,
           Boolean { renamed by [true -> pos, false -> neg] }

end Monotone-restrictions
```

The function *is-monotone* makes two lists of identifiers. One list contains the variables in the positive terms, the other the variables in the negative terms. The first boolean flag in this function indicates if a variable must be added to the positive list or to the negative list. The second flag indicates the sign of the term that is examined. To increase the readability of the module the boolean functions true and false are renamed. If the two lists are disjoint the expression is monotone increasing or decreasing [1].

```
[204]  is-monotone(ie) = disjoint(var-list(pos,pos,ie,null),
                                  var-list(pos,neg,ie,null))
```

A complete specification of the function *var-list* in is found in module *Monotone-restrictions*.

All symbolic ranges in the symbolic evaluation must be positive. In general this restriction is not satisfied. Consider the input declarations

[1] Here we pay the price for not having specified a normalized index expression. The is-monotone function is only correct if applied to a normalized index expression. For other representations it may lead to a wrong result.

```
declarations
INPUT
    INDEX                n,p;
    VECTOR [n to p]      X
```

The range **[n to p]** is positive if **n >= 1** and **p >= n**. CONDUCTOR generates during the compilation of a statistical program a sequence of these input restrictions.
The positive restriction of symbolic ranges is specified in the module *Range-restrictions* in equation *[193]* for function *pos-range-restr*.

```
[193]  pos-range-restr(range(ie1,ie2),ts)
            = add-inp-restr(ie1,abs-const(ind(1)),ts,
                 add-inp-restr(ie2,ie1,ts,no-restrictions))
```

The function *add-inp-restr* adds an input restriction (*inp-restr (ie1,ie2,ts)*) to the input restriction sequence *irs*, when the order relation of the two index expressions in the range is undecided.

```
[192] add-inp-restr(ie1,ie2,ts,irs)
           = if(eq(order(ie1,ie2),undecided)),

                add-item(inp-restr(abs-minus(ie1,ie2),ts),irs),

                if(eq(order(ie1,ie2),less),

                   error-inp-restr-seq,
                   irs)
```

9.4. INPUT RESTRICTIONS

All input restrictions in CONDUCTOR are given as a positive restriction on an index expression.

index expression ≥ 0

The variables in the index expression in a restriction may only be input variables. CONDUCTOR checks whether all variables are input variables by examining the technique symbol table. The specification of input restrictions reads

inp-restr: IND-EXPR # TECH-SYMTAB   -> INP-RESTR

with equation

[179]   inp-restr(ie,ts) = pos-restr(ie)

when are-input-vars(ie,ts) = true

The function *are-input-vars* specifies that all the variables in the expression (*ie*) must be declared in the input section of a statistical program. This function uses information stored in the technique symbol table (*ts*) and is specified in module *Are-input-vars*. The function *pos-restr(ie)* specifies that the restriction *ie >= 0* must hold.

9.5 ASSIGNMENT OF A SYMBOLIC RANGE TO AN INDEX VARIABLE

During the symbolic evaluation of a matrix language program symbolic ranges are assigned to index variables in:

- an INPUT declaration; an index variable **n** declared in an input section is assigned the range **[n to n]**,
- an index assignment statement; a non-input index variable p in the assignment **p = ie** is assigned the range calculated for the index expression ie,
- a for-statement; a loop control variable **i** in the for loop

for i := ie1 to ie2 do statements
is assigned the range determined by the lower bound of the range calculated for index expression **ie1** and the upper bound calculated for the index expression **ie2**.

To ensure that the for-statement is at least executed once, it must be checked if the upper bound of the range calculated for index expression **ie1** is smaller or equal to the lower bound of the range calculated for index expression **ie2**. If this restrictions is undecided input restrictions are generated for the input index variables of the matrix language program in which the for-statement occurs.

These symbolic ranges assigned to index variables are stored in the symbolic range table, specified by the sort RANGE-TABLE in the module *Range-tables*; the assignment is specified in module *Range-assignments*.

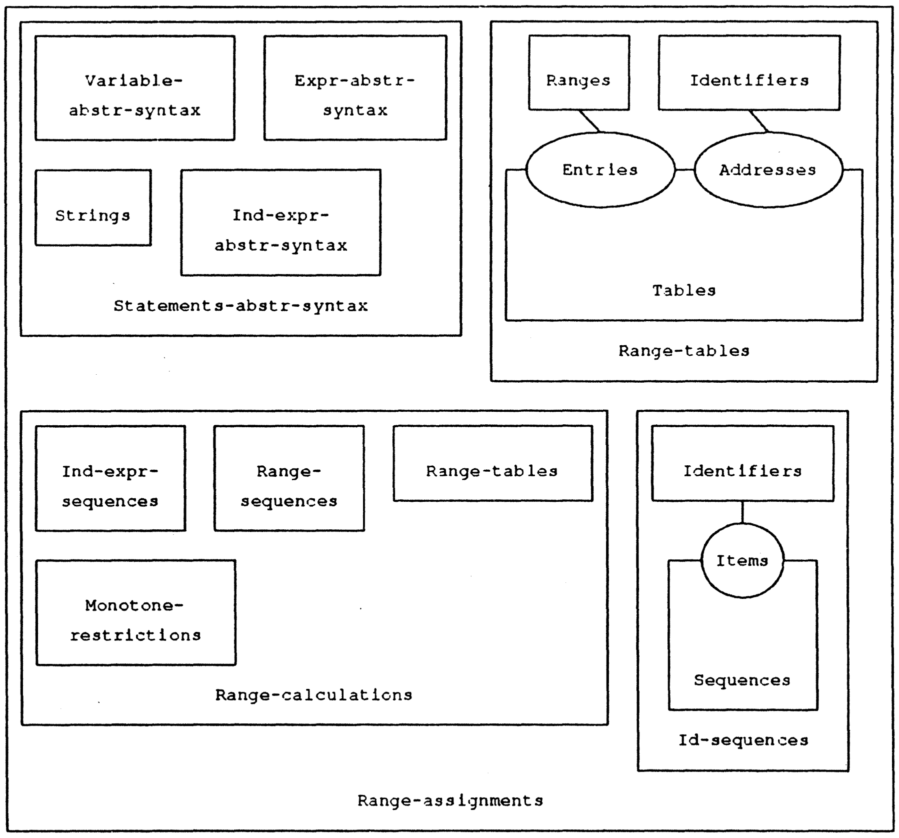
The abstract syntax of a for-statement is given in the module *Statements-abstr-syntax* in the function *abs-for*

```
    abs-for:  INDEX-VAR # INDEX-EXPR
              # INDEX-EXPR # STATEMENTS  -> STATEMENT
```

The symbolic range calculated for the range consisting of the two index expressions (*ie1* and *ie2*), is stored in the range table (*rt*) for the control index variable (*id*) in the for-loop.

```
[229]  assgn-range(abs-for(i,ie1,ie2,stmts),rt)
          = assgn-ranges(
                stmts,
                insert(id,
                       range-calc-range(range(ie1,ie2),
                       rt)))
```

Similar equations are specified for the range assignments in index assignment statements and INPUT index declarations in module *Range-assignments*.



9.6. MATRIX ELEMENT REFERENCES

The restriction that a symbolic range (*range(ie1,ie2)*) is a subrange of another symbolic range (*range(ie3,ie4)*) is specified in module *Range-restrictions* in function

```
    range-within-range: RANGE # RANGE # TECH-SYMTAB
                        -> INP-RESTR-SEQ
```

with equation

```
[104] range-within-range(range(ie1,ie2),range(ie3,ie4),ts)
          = add-inp-restr(ie1,ie3,ts,
                add-inp-restr(ie2,ie1,ts,
                    add-inp-restr(ie4,ie2,ts,no-restrictions)))
```

Equation *[104]* specifies that, if the subrange restriction can not be verified, a sequence of input restrictions, **ie3 <= ie1 <= ie2 <= ie4**, is generated.
The generation of input restrictions for variables is specified in module *Gen-restr-variables* in function *gen-restr-var*.

```
      gen-restr-var: INP-RESTR-SEQ # RANGE-TABLE #
                     TECH-SYMTAB # VARIABLE
                     -> INP-RESTR-SEQ
```

A matrix element reference *abs-var(id,inds)* consists of an identifier *id* and an index expression sequence *inds*. For each of these index expressions a symbolic range is evaluated, and the static type checker checks if these symbolic ranges are subranges of the declared dimension ranges of the matrix. If the static type checker can not determine if this restriction holds, input restriction are generated. Note that the generated input restrictions consist of:

- input restrictions that guarantee that all symbolic ranges in the symbolic evaluation of an index expression *ie* are positive; this was specified in the function *restr-calc-ranges*,
- input restrictions that guarantee that the calculated range lies within the referenced range; this was specified in function *range-within-range*.

```
[234] gen-restr-var(irs,rt,ts,abs-var(id,inds))
           = conc(restr-calc-ranges(inds,rt,ts,irs),
                  range-within-range(calc-ranges(inds,rt),rngs,ts))

      when eq-type(type(id^ts),matrix-type(rngs)) = true
```

9.7. A TYPE AND DIMENSION BOUND CORRECT STATISTICAL PROGRAM

In module *Input-restr-generator* it is specified how all input restrictions are generated during the symbolic evaluation of a statistical program.

gen-restr-pro: STAT-PRO # TECH-SYMTAB -> INP-RESTR-SEQ

The module *Input-restr-generator* imports modules that specify how input restrictions are generated for sections of the statistical program. These modules in turn import modules that specify how restrictions are generated for for-statements and declarations, etc. A complete specification of the input generation mechanism can be found in appendix I. A type correct statistical program in combination with the generated input restrictions is called a *type and dimension bound correct statistical program.*

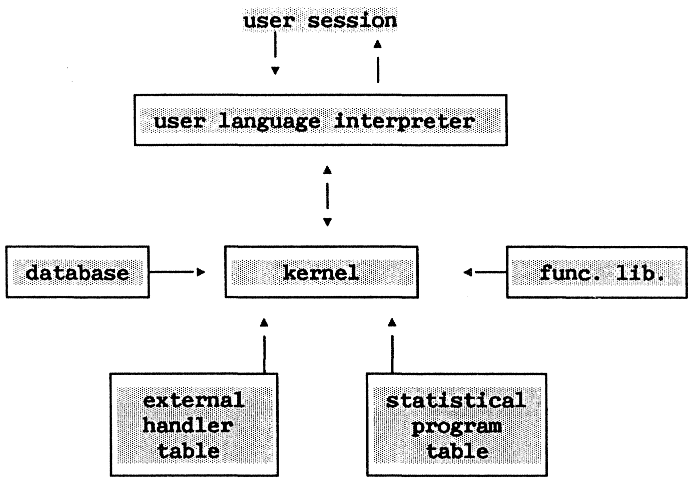
PART III

FORMAL SPECIFICATION OF THE KERNEL

10. THE KERNEL

The kernel is a simple virtual machine, with a processor that can execute kernel instructions. The instruction set of the kernel (*the kernel language*) shows close resemblance with a simple language, called postfix notation, used for intermediate code generation in compilers for higher-level programming languages. A kernel (language) program is a *postfix representation* of a statistical program. Postfix notation is, for example, used as intermediate code in most SNOBOL compilers (see Griswold [1972]).
The kernel instructions can be regarded as the semantical actions of the statistical language. There is, for example, an instruction that can access and execute numerical functions written by computer scientists, and an instruction that can prompt the user of a statistical technique for input. A statistical techniques, implemented by the technical statistician, is compiled into a kernel program and stored in the *statistical technique table*. This table contains the collection of available statistical techniques in the generated software. If in a user session a statistical technique is executed, the user language interpreter instructs the kernel to retrieve the technique from the statistical technique table, and to execute the kernel instruction sequence of the technique.
Besides the processor, important parts of the kernel are the *data memory, the data stack and the external handler tables*. The data memory of the kernel contains the values of the variables in the user session, and the value of the variables of an executing statistical technique. Expressions in the kernel language are evaluated on the data stack. Stacks are, for

example, also used in the design of the VAX-11 assembly language (see Peeters [1985]). The external handlers, written by either a computer scientist or a data expert, are stored in *the external handler table.*
In this chapter we will give an informal description of the various parts of the kernel and the kernel language. A complete formal specification of the kernel can be found in appendix K.



10.1. MEMORY ORGANIZATION

The data memory of the kernel consists of, on the one hand, the symbol table of the user session and, on the other hand, the technique symbol table of an executing statistical program. In these tables the values of the variables in, respectively, a user session, and an executing statistical technique are stored. Note that these tables also contain type information for the static type checking. The use of the technique symbol table in type checking has already been discussed in section 8.4.
In the memory of the kernel, data transfer is possible between the user symbol table and the technique symbol table. This makes it possible for a user to initialize variables in a statistical technique. In the formal specification of the memory, in module *Memory,* this is represented by the functions *user-store* and *user-load.* Data transfer is also possible between the technique symbol table and the data stack. Recall that the

expressions in the kernel language are evaluated on the data stack. In the formal specification, transfer between the data stack and the technique symbol table is represented by the functions *get-data* and *store-data*.

$$\begin{tikzcd}
\boxed{\text{user symbol table}} \arrow[d, shift left, "user\text{-}load"] \\
\boxed{\text{technique symbol table}} \arrow[u, shift left, "user\text{-}store"] \arrow[d, shift right, "get\text{-}data"'] \\
\boxed{\text{data stack}} \arrow[u, shift right, "store\text{-}data"']
\end{tikzcd}$$

10.2. THE INSTRUCTION SET

In the following subsections we give a short description of each of the kernel instructions. A formal specification of the effect of the instructions on the state of the kernel is discussed in section 10.5.

10.2.1. User interface instructions.

The USER-LOAD and USER-STORE instructions form the user interface of an implemented statistical technique. The USER-LOAD instruction prompts the user to initialize an input variable. If no message is specified in the declaration of that variable in the statistical program, its identifier is displayed followed by a question mark and an assignment symbol:

```
variable_id ?=
```

A user can either enter a constant value or a variable declared earlier in his session. The user answer, of course, must be of the correct type. The value assigned to an input variable of a statistical program is stored in the technique symbol table. The address of the variable in this table is a parameter of the USER-LOAD command. Matrices can also be initialized using series of observation. The observations of the series will form the columns of the matrix (see chapter 13).

Example 10.3: loading a matrix on the data stack.

Two index variables n and m and a matrix X are declared in the INPUT/OUTPUT section of a statistical program.

```
input/output section of statistical program
INPUT
     INDEX     n,m;

     MATRIX    [1 TO n, 1 TO m]       X
               MESSAGE: "Enter a matrix identifier:"
```

At the kernel level this is represented as three USER-LOAD instructions.

```
kernel instructions
     USER-LOAD       n
     USER-LOAD       m
     USER-LOAD       X
```

These three instructions are the first to be executed when the statistical technique is called by a user. Because no messages are specified in the declaration of the variables **n** and **m**, the identifiers are used in the prompt in combination with the prompt of the user command language (the string '>u:'). After the user has entered the values for **n** and **m**, the run-time type of the matrix **X** is calculated, and the user is prompted with the specified message to enter a matrix identifier. In this example a matrix with dimension ranges **[1 TO 2, 1 TO 3]** is expected.

user session

```
            n ?=
>u: 2
            m ?=
>u: 3

Enter a matrix identifier:
>u: usermat
```

The matrix **usermat** should be declared and initialized in a user session before the statistical technique is called (see chapter 14). A copy of the matrix usermat is stored in the technique symbol table entry of X.
{ end example 10.3 }

The USER-OUTPUT instruction transfers the calculated output variable back to the user. The output variable is copied from the technique symbol table to the user symbol table, and the output variable is displayed. If a message is specified in the declaration of the output variable, this message will accompany the output.

10.2.2. Data transfer between the symbol tables and the data stack.

A LOAD instruction pushes a copy of the value of a variable from the technique symbol table on the data stack. A STORE instruction pops the top element of the data stack, and stores it in the technique symbol table. In both instructions the address of the variable in the technique symbol table is a parameter. Separate load and store instructions are defined for submatrices and matrix elements. These instructions have as an additional parameter the number of dimensions of the matrix. This parameter tells the processor how many element or subrange references are on the data stack.

10.2.3. Function calls.

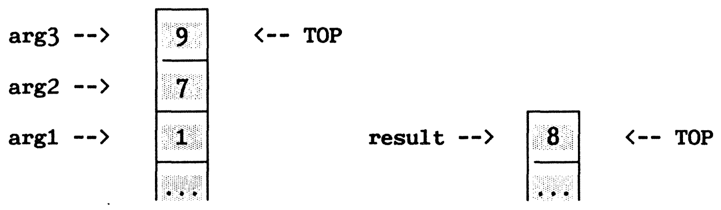
Functions operate on the data stack of the kernel. When a function is called, the results of evaluating its arguments, have already been pushed on the data stack. The FUNCION-CALL instruction calls the numerical function, and replaces the argument values on the stack by the result. The address of the numerical function in the function code table is a parameter of this instruction.

Example 10.4. Execution of a FUNCTION-CALL instruction.

Assume that in the statistical language there exists a predefined function **f4** with 3 arguments of type index and a result also of type index.

f4(arg1,arg2,arg3) = min(arg1 + arg2, arg3)

When the function **f4** is called, during the execution of a program, the data stack contains the values of the arguments. After execution of the function the arguments are popped from, and the result is pushed on the stack.



{ end example 10.4 }

10.2.4. The JUMP instruction.

To alter the sequential execution of instruction sequences both conditional and unconditional jump instructions exist. The conditional jump instructions, JUMP_TRUE and JUMP_FALSE, inspect the top of the data stack. If the top of the data stack contains a boolean with value true, respectively false, the address of the next instruction to be executed is obtained by adding the relative address specified in the jump instruction to the absolute address of the current instruction. Otherwise, the next instruction in the instruction sequence is executed. The unconditional JUMP instruction always jumps to the calculated address.

10.2.5. Instructions for exception handling.

Exception handlers in CONDUCTOR are represented at the kernel level as sequences of kernel instructions. The RAISE instruction transfers control from the instruction sequence of the statistical program to the instruction sequence of an exception handler. The address of the instruction after the raise instruction is saved. The exception handler is identified by its name, which is an argument of the RAISE instruction. This identifier is the address of an exception handler instruction sequence in an exception handler table. An UNRAISE instruction transfers control back to the interrupted instruction sequence of the statistical program. No nested exception handling is allowed in CONDUCTOR: if an exception handler invokes yet another exception the execution of the statistical technique is stopped.

10.2.6. The DISPLAY instruction.

The DISPLAY instruction prints a message. The only argument in this instruction is a message string.

10.2.7. The input restriction check.

The index input variables of a statistical technique must obey the generated input restrictions. This check is represented at the kernel level by the CHECK-RESTR instruction. This instruction evaluates the index expression in the input restrictions, and checks if they obey the restriction. If the input restrictions are not satisfied the user can restart or abort the technique. The input restrictions are not a parameter of the CHECK-RESTR instruction, but are part of the state definition of the kernel.

10.2.8. Evaluating index expressions.

The evaluation of index expression is represented at the kernel level by the EVAL-EXPR instruction. This instruction evaluates the index expression that is a parameter of this instruction, using the values of the variables in the technique symbol table.

10.2.9. A halt instruction.

The halt instruction terminates the execution of the kernel instruction sequence.

10.2.10. Algebraic specification of instructions.

The complete instruction set of the kernel is specified in module *Instructions*.

```
module Instructions
begin
   exports
     begin
       sorts INSTR
       functions:
          -- user interface --
          user-load:  ID                    -> INSTR
          user-store: ID                    -> INSTR

          -- load instruction --
          load:       TECH-DATA             -> INSTR
          load:       ID                    -> INSTR
          load-elem:  ID # INDEX            -> INSTR
          load-subm:  ID # INDEX            -> INSTR

          -- store instruction --
          store:      ID                    -> INSTR
          store-elem: ID # INDEX            -> INSTR
          store-subm: ID # INDEX            -> INSTR

          -- index expression instructions --
          check-restr: INP-RESTR-SEQ        -> INSTR
          eval-expr:   INDEX-EXPR           -> INSTR

          -- other instruction --
          jump:       INDEX                 -> INSTR
          jump-true:  INDEX                 -> INSTR
          jump-false: INDEX                 -> INSTR
          increm:     ID                    -> INSTR
          f-call:     ID # INDEX            -> INSTR
          display:    STRING                -> INSTR
          raise:      ID                    -> INSTR
          unraise:                          -> INSTR
          halt:                             -> INSTR

     imports Technique-data, Identifiers,
             Ind-expr-abstr-syntax, Strings

end Instructions
```

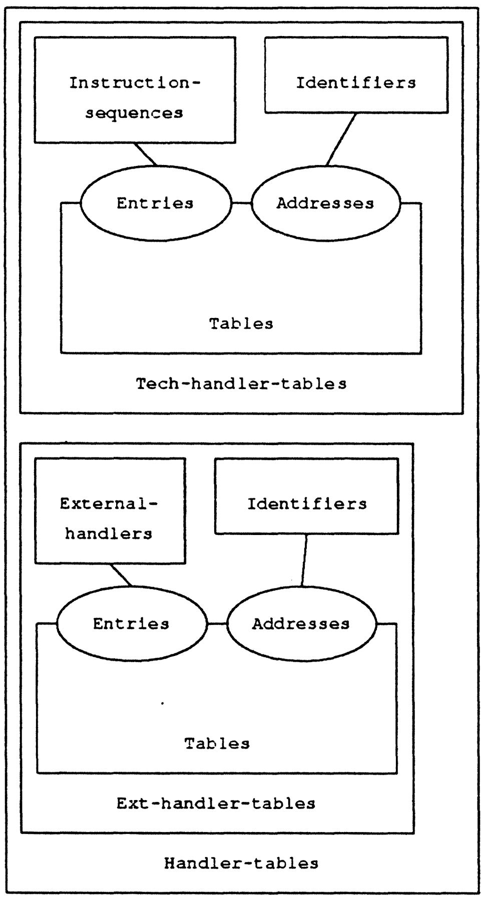
A sequence of instructions is specified by binding the sort *ITEM* in the parameter *Items* of the module *Sequences* to the sort *INSTR* in the module *Instructions*. The sort *INSTR-SEQ* is the formal abstraction of an instruction sequence at the kernel level. An instruction address in the kernel is specified as an instruction sequence combined with an integer indicating the position in the sequence. Given the address a function *fetch* can retrieve an instruction from a sequence of instructions.

[293] *fetch(addr(is,i)) = item-no(i,is)*

Where *is* is an instruction sequence and *i* an index.

10.3. EXCEPTION HANDLER TABLES

The kernel has tables to store exception handlers. The exception handler tables for exception handlers specified by a technical statistician are specified in module *Tech-handler-tables*. Exception handlers created by a technical statistician are stored as sequences of kernel instructions, and each exception handler is uniquely determined by its name.
Two exception handler tables are defined in the kernel: a table with handlers specified by the technical statistician, and a table with external handlers specified by either the data expert or the computer scientist. The RAISE-instruction searches the tables to find a handler with a given name. If a handler with that name is found in the handler table of the technical statistician this handler is returned, otherwise the external handler table is searched. If no handler is found in either of these tables, a sequence containing only a halt instruction is returned, causing the statistical program that is calling the exception handler to terminate (a detailed discussion can be found in chapter 12).



10.4. THE DATA STACK

Expressions in the statistical language are evaluated on the data stack. A stack is a sequence of data. Items can be popped from the stack or pushed on the stack. This is specified in the module *Data-stacks*.

10.5. THE PROCESSOR

The state of the kernel is determined by (1) the data memory, (2) the data stack, (3) the current instruction, (4) the address of the next instruction, (5) the information needed to restart an interrupted instruction sequence, (6) the exception handler tables, (7) the input restrictions and (8) the user display, as specified in the function *state* in module *Kernel-states*.

```
state: MEMORY #
       DATA-STACK #

       INSTR #             -- current instruction --
       INSTR-ADDR #        -- address next instruction --

       RESET-INFO #
       HANDLER-TABLES #

       INP-RESTR-SEQ #
       STRING              -- display --

                        -> STATE
```

The execution of a kernel instruction modifies the state of the kernel.

```
execute: STATE       -> STATE
```

The working of the individual instructions is specified in the equations for the function *execute* in module *Processor*. Consider, for instance, the JUMP-TRUE instruction. This instruction checks the top element of the data stack. If the flag (*b*) on top of the data stack (*ds*) is true, the relative address (*int1*) is added to the current address (*addr(is,int2)*), and the instruction at this address is executed next. The flag is popped from the data stack. The other information remains unchanged.

```
[324] execute(state(m,
                    push(t-data(b),ds),
                    jump-true(int1),
                    addr(is,int2),
                    ri,ht,irs,dis)
            =state(m,
                    ds,
                    if(b,
                       fetch(addr(is,add(int1,int2)),
                       fetch(addr(is,int2))),
                    if(b,
                       next(addr(is,add(int1,int2)),
                       next(addr(is,int2))),
                     ri,ht,irs,dis)
```

The specification of other kernel instructions is given in module *Processor* in Appendix K.

10.6. KERNEL PROGRAMS

A statistical technique is reduced at the kernel level to the following abstract notions:

- the name of the technique,
- a sequence of instructions,
- a sequence of input restrictions,
- an exception handler table,
- a technique symbol table.

Such a representation of a statistical technique at the kernel level is called a kernel program. A kernel program is formalized in the module *Kernel-programs*.

```
module Kernel-programs
begin
  export
      begin
        sorts KERNEL-PRO
        functions
          kern-pro: INSTR-SEQ #
                    INP-RESTR-SEQ #
                    TECH-SYMTAB #
                    TECH-HANDLER-TABLE  -> KERNEL-PRO
      end

imports Techn-symtabs,Instruction-sequences,
        Tech-handler-tables, Input-restr-sequences
end
```

The collection of all kernel programs is stored in the statistical technique table. The kernel can execute the kernel programs in this table, as specified in the functions *run-technique* in module *Kernel.*

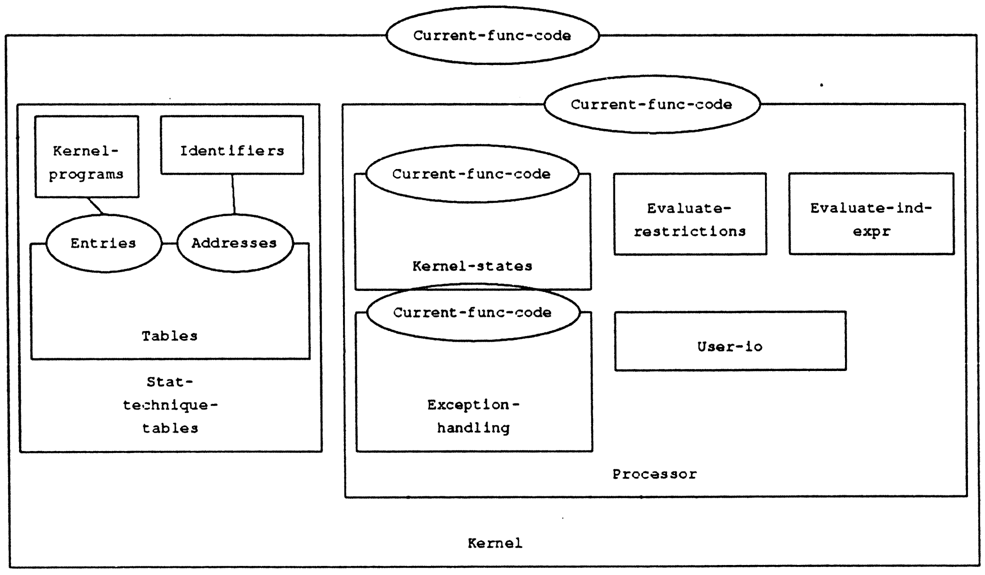
```
   run-technique: USER-SYMTAB # KERNEL-PRO #
                  EXT-HANDLER-TABLE          -> USER-SYMTAB
```

The kernel retrieves the statistical technique from the table. The symbol table (*ts*) and exception handler table (*eht*) of the technique are entered in the kernel, and the processor starts executing the first instruction in the instruction sequence (*is*) of the statistical technique.

```
[331] run-tech(us,kern-pro(is,irs,ts,tht),eht)
          = result(
              execute(
                state(memory(ts,us),empty-stack,
                      first(is), addr(is,1),
                      no-reset-info, handlers(tht,eht),
                      irs,string(blank))))
```

The result statistics, as calculated by a statistical technique, are stored in the user symbol table (*us*). Execution of a statistical technique, therefore, can be regarded as the modification of a user symbol table.

The code of actual functions in the statistical language is left unspecified. The function code table in the kernel is not instantiated. The parameter *Current-func-code* emphasizes that the design of CONDUCTOR is independent of these functions.



11. TRANSLATION OF A STATISTICAL PROGRAM

A statistical program in CONDUCTOR is translated into a kernel program. This translation process is divided in five steps: lexical analysis, parsing, type checking, the generation of input restrictions, and code generation.

$$\begin{tikzcd}
\text{lexical analysis} \arrow[d] \\
\text{parsing} \arrow[d] \\
\text{type checking} \arrow[d] \\
\text{input restriction generation} \arrow[d] \\
\text{code generation}
\end{tikzcd}$$

The formal specification of the translation process starts with the abstract syntax of the statistical language. The lexical analysis and parsing is left unspecified. A complete formal specification of these processes, for a simple programming language, can be found in Bergstra et al. [1986]. Type checking and the generation of input restrictions were already discussed in the chapters 8 and 9. The formal specification of the code generation is discussed in this chapter.
A sequence of kernel instructions is a postfix representation of a statistical program. The generation of kernel instructions, in CONDUCTOR, can be compared with syntax directed intermediate code generation for

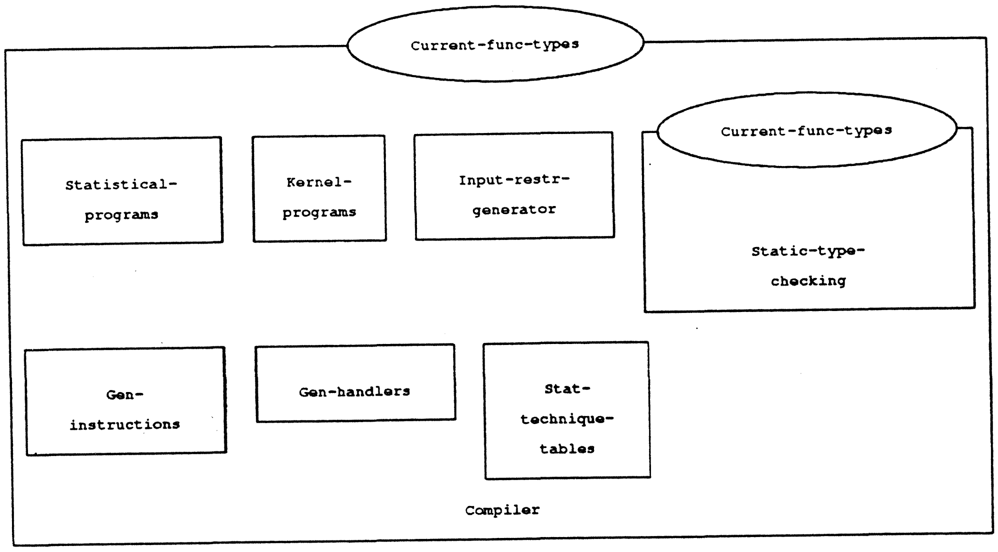
higher-level programming languages. The intermediate code in compilers for these languages hide the details of a particular target machine (see Davidson and Fraser [1984] and Tanenbaum et al. [1983]). The only difference is, that most of these compilers use three-address statements as intermediate language, and CONDUCTOR uses postfix notation. The code generation for PASCAL, for example, is discussed in Wirth [1971] and Ammann [1977].

11.1. GENERAL STRUCTURE OF THE COMPILER

A compiler is characterized by three languages: the source language, the target language and the implementation language (see McKeeman [1974]). The source language of the specified compiler is the abstract syntax of the statistical language, and the target language the kernel language. By specifying of the compiler in **ASF**, we, by definition, leave the implementation language undetermined.
The kernel language has been formalized in the previous chapter (see section 10.6). The compiler constructs each of the parts of a kernel program for a particular statistical program *sp*, as specified in the function *compile* in module *Compilers*.

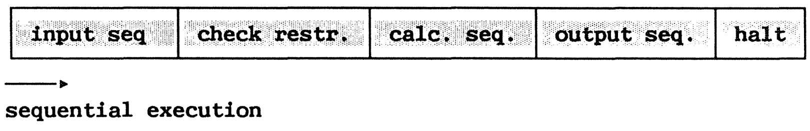
```
[402] compile(sp) = kern-pro( gen-instr-seq(sp),
                              gen-restr-pro(sp,type-check-pro(sp)),
                              type-check-pro(sp),
                              gen-handlers(sp))
```

All the functions on the right hand side of this equation are specified in the imported modules of module *Compilers*. The function *gen-restr-pro* was discussed in section 9.8 and the function *type-check-pro* in section 8.7. In this chapter we describe the functions *gen-instr-seq*, that specifies the generation of kernel instructions, and *gen-handlers* that specifies the generation of an exception handler table.



11.2. GENERATION OF INSTRUCTION SEQUENCES

An instruction sequence in a kernel program consists of five parts: a sequence of input instructions, an instruction that checks the input restrictions, a sequence of calculation instructions, a sequence of output instructions and a halt instruction.



The generation of the instruction sequence is specified in the module *Gen-instructions*. The function *gen-inp-instr*, in this module, specifies how input instructions are generated. The function *check-restr* specifies that an instruction to check the input restrictions is added to this sequence. The calculation instructions are added as specified by the function *gen-calc-instr*, and the output instructions by the function *gen-out-instr*. At the end of each instruction sequence a halt instruction is place. Note that in the formal specification the inner-most function call is evaluated first.

```
[401]    gen-instr-seq(sp) = add-item(halt,
                               gen-out-instr(
                                gen-calc-instr(
                                 gen-check-restr(
                                  gen-inp-instr(null-instr-seq,
                                                sp),sp),sp),sp)
```

11.2.1. Generation of input and output instructions.

For each variable in the input/output section of a statistical program a *user-load/user-store* instruction is generated. The function *gen-code-id*, in module *Gen-code-declarations*, specifies that a *user-load* instruction is added to the sequence (*is*) for each input variable (*id*)

```
[362] gen-code-id(is,id,load)
           = add-item(user-load(id),is)
```

and a *user-store* instruction is added for each output variable

```
[363] gen-code-id(is,id,store)
           = add-item(user-store(id),is)
```

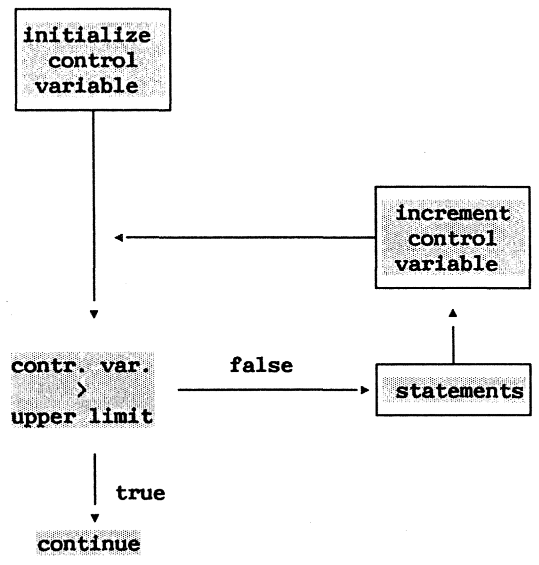
The complete specification of the generation of *user-load/user-store* instructions is given in module *Gen-code-declarations*.
Note that the functions *user-load* and *user-store* are only instructions to load and store user variables. The actual loading and storing is done when the kernel executes these instructions.

11.2.2. Generation of instructions for the calculation of the statistics.

The function *gen-code-stm* in module *Gen-code-statements* specifies how kernel instructions are generated for each type of statement in the statistical language. The generation of instructions for a for-statement with control variable *id*, index expressions *ie1* and *ie2* and loop statements *stmts*, is specified as follows

```
[351]   gen-code-stm(abs-for(id,ie1,ie2,stmts)
            = increm-code(id,stmts,
                  gen-code-stmts(
                       check-up-limit(id,ie2,stmts,
                         ini-control-var(id,ie1,is)),stmts))
```

This is represented by the well-known flow chart of a for-statement.


initialize control variable
increment control variable
contr. var. > upper limit
false
statements
true
continue

The instruction sequence for the initialization of a control variable is specified in function *ini-control-var*. This sequence contains an instruction to evaluate the index expression (*iel*) and an instruction to store the result in the control variable *id*.

```
[352] ini-control-var(id,iel,is)
        = add-item(store(id),
                   add-item(eval-expr(iel),is))
```

After these instructions, the instructions must be generated that check if the control variable is larger than the upper limit of the for loop. A conditional jump will force the kernel to jump to the statement following the for-statement instructions, in case the control variable is larger than the upper limit. Also the size of the jump and the incrementing of the control variable is specified . These equations can be found in module *Gen-code-statement*, where also the specifications are given for the generation of instructions for the other types of statements in the statistical language.

11.2.3. Generation of exception raise instructions.

A test consists of internal declarations, statements and exception raises. The function *gen-instr-test-sec* in module *Gen-code-tests-section* specifies that first instructions are generated for the statements (*stmts*) and than for the for exception raises (*rs*).

```
[365] gen-code-test-sec(abs-test-sec(decl,stmts,rs),is)
          = gen-code-raises(rs,
               gen-code-stmnts(stmts,is))
```

Recall that the instruction sequence of a statistical technique is interrupted if the test indicates that the data violates the assumptions of the statistical technique.
An exception raise statement consists of a boolean expression and an identifier. First the instructions for the evaluation of expression *expr* are generated, then a *jump-false* and a *raise* instruction are added to the instruction sequence.

```
[368] gen-code-raise(is,abs-raise(expr,id)
          = add-item(raise(id),
                  add-item(jump-false(increm(1)),
                       gen-code-expr(is,expr)))
```

If the boolean expression in the raise statement is true the exception will be raised. Otherwise the kernel will continue with the instruction following the raise instruction. The *jump-false* instruction will force the kernel to skip the *raise* instruction.

11.3. GENERATION OF A HANDLER TABLE

Instruction sequences generated for an exception handler section are not added to the instruction sequence of the statistical program, but are stored in an exception handler table. The statements (*stmnts*) in the handler are translated and the resulting instruction sequence is inserted in the handler table (*ht*), at the given address (*id*).

```
[371] gen-code-handl(ht,abs-handl(id,decl,stmts))
          = insert(id,
                 gen-code-stmnts(null-instr-seq,stmts),
                 ht)
```

This function is specified in module *Gen-code-handler-section.*

11.4. OPTIMISING KERNEL INSTRUCTIONS

The problem of optimising instruction sequences was not discussed in this chapter. For an efficient implementation of CONDUCTOR, however, the instruction sequences must be optimised. In the current prototype, the only optimization technique that is applied is common subexpressions elimination. This improves the execution time of the statistical techniques considerably. Recall that kernel instructions are instructions for a virtual machine. These instructions may involve time consuming functions, such as, matrix inversion. The common subexpression elimination is specified completely independent of the specification in this book, because it is considered to be an implementation problem, instead of a fundamental part of the specification of CONDUCTOR. Also other optimization techniques, such as, constant folding or removal of loop invariant computations, will probably increase the performance of the implemented statistical techniques.

12. EXCEPTION HANDLING

Technical statisticians, data experts and computer scientists each look at a statistical technique from their own level of abstraction. Software implemented by one of these experts might detect that the execution of a statistical technique must be interrupted, because necessary conditions for execution are not satisfied. In computer science such conditions are called exceptions and the interrupt is called *the raising of an exception*. Especially in systems, that must remain in continuous operation, it is important that the execution of a program is not stopped when an exception occurs. Before a program can continue with the 'normal' operations, it must deal with the unexpected situation, without completely terminating execution.
The response to an exception condition is called the *handling of an exception* (see Goodenough [1975] and Wiener[1983]). Two basic approaches in exception handling can be distinguished:

- when an exception occurs, normal program flow is interrupted and control is passed to the exception handler; after completion of the exception handling, control is returned to the point at which the exception occurred,
- when an exception occurs, this makes normal flow of the program impossible, and the program is terminated.

In the first approach it is possible to make a repair action and thereafter continue operation. Examples of programming languages with exception handlers that allow repair actions are ADA, PL/1 and PL/C.
The process of statistical analysis in CONDUCTOR is seen as a continuous process. In statistical analysis, an applied statistician uses

statistical techniques to analyze his data set. These statistical techniques can only be applied under certain conditions. Therefore, during statistical analysis, it often happens that a statistical technique can not calculate the required statistics. Such situations can be seen as the occurrence of an exception. Yet, the applied statistician wants to continue the analysis and needs advice. Are there any alternative statistical techniques that can tackle the problem? Or are there data preprocessing techniques that do make analysis of his data set possible? The answers, of course, can only be given by the experts. The exception handling mechanism allows the experts to provide this information. The problem of given proper advice when an exception occurs, can be compared with the problem of giving understandable error messages, when an error occurs in procedures, that are hidden for the user of the software, in higher-level programming languages. For a discussion on this topic see Efe [1987].
In CONDUCTOR an exception is raised when :

- test results at the statistical level indicate that a basic assumption of a statistical technique is violated,
- inconsistencies are detected in the series used in a statistical technique,
- numerical problems occur in function calls.

In this chapter we will discuss exceptions in tests and in numerical functions. Exceptions raised due to inconsistencies in the data are discussed in chapter 13.

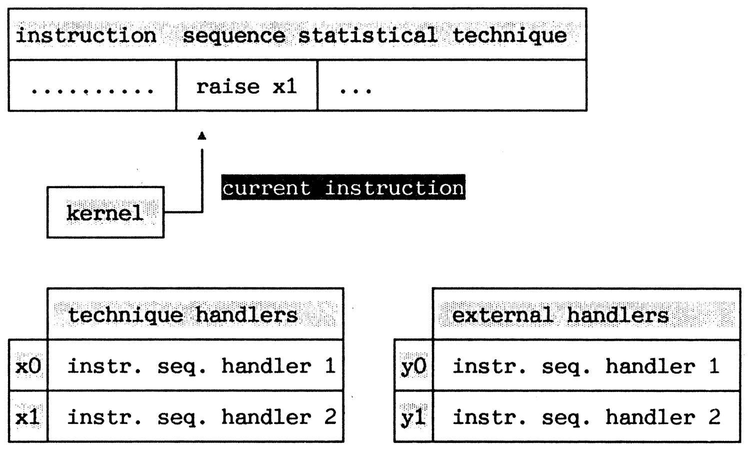
12.1. RAISING OF EXCEPTIONS

To explain the exception handling mechanism in CONDUCTOR we start at the kernel level. The kernel instruction *raise(h-id)* forces an interrupt of the instruction sequence of the statistical technique. Where *h-id* is the exception identifier. If an exception handler, with the given identifier, is found in the exception handler tables, the next instruction executed by the kernel is the first instruction in this exception handler.
Two groups of exception handlers are distinguished in CONDUCTOR:

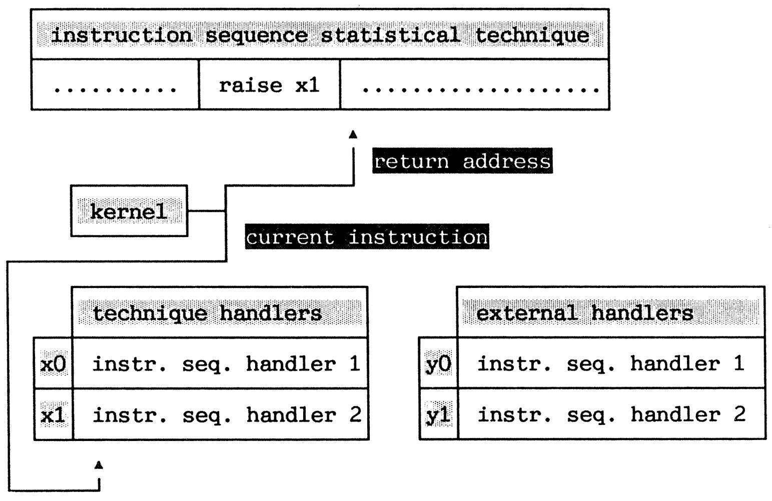
- *the technique handlers*: the exception handlers specified by the technical statisticians inside a statistical technique,

EXECUTION OF A RAISE INSTRUCTION

A. State before execution.



B. State after execution.



- *the external handlers*: exception handlers specified outside a statistical technique by either a technical statistician, a data expert or a computer scientist.

The technique handlers have the highest priority. If an exception is raised the technique handler table is searched first. If no handler is found in this table the external handler table is searched. If no handler is found in either of these tables the execution of the statistical technique is aborted.

After the exception handler is executed the kernel returns to the interrupted instruction sequence of the statistical technique. To make this return possible the address of the next instruction to be executed in the statistical technique is saved, when the technique is interrupted. If an external exception handler is taking over control, also the symbol table of interrupted statistical technique is saved.

The exception handling mechanism is specified in the function *handle* in module *Exception-handling*. For example, it is specified that if the processor is in a state 1.

state description of the kernel 1	
memory	*memory(ts,us)*
data stack	*ds*
current instruction	*raise(id)*
address next instruction	*ia*
reset information	*no-reset-info*
handler tables	*handlers(tht,eht)*
input restrictions	*irs*
display	*dis*

and if the handler is found in the exception handler table of the technical statistician tht, execution of the raise instruction will force the kernel to execute the first instruction of the exception handler *first(h-id^tht)*, set the address of the next instruction to the address of the second instruction in the exception handler *addr(h-id^tht,1)* and save the next instruction of the interrupted instruction sequence *reset-info(ia)*. The resulting state of the kernel is state 2.

state description of the kernel 2	
memory	*memory(ts,us)*
data stack	*ds*
current instruction	*first(h-id^tht)*
address next instruction	*addr(h-id^tht,1)*
reset information	*reset-info(ia)*
handler tables	*handlers(tht,eht)*
input restrictions	*irs*
display	*dis*

On the other hand, if the handler is found in the external handler table *eht*, the kernel starts executing the first instruction in the external handler table *first(han-instr-s(h-id^eht))*, sets the address of the next instruction to the second instruction in this instruction sequence *addr(han-instr-s(h-id^eht),1)* and saves both the return address and the symbol table of the interrupted statistical technique *reset-info(ia,ts)*. The symbol table of the external handler replaces the symbol table of the statistical technique in the memory of the kernel *memory(han-symtab(h-id^eht),us)*. This results in state 3.

state description of the kernel 3	
memory	*memory(han-symtab(h-id^eht),us)*
data stack	*ds*
current instruction	*first(han-instr-s(h-id^eht))*
address next instruction	*addr(han-instr-s(h-id^eht),1)*
reset information	*reset-info(ia,ts)*
handler tables	*handlers(tht,eht)*
input restrictions	*irs*
display	*dis*

If the exception handler identifier is found in neither of the tables the execution is stopped. The formal specification of the exception handling in the module *Exception-handling* reads

```
[305] handle(state(memory(ts,us),ds,raise(h-id),ia,
                 no-reset-info,handlers(tht,eht),irs,dis)
        = if(found(h-id,tht),

          -- handler found in the technique symbol table --

            state(memory(ts,us),ds,
                  first(h-id^tht),addr(h-id^tht,1),
                  reset-info(ia),handlers(tht,eht),
                  irs,dis),
            if(found(h-id,eht),

          -- handler found in the technique symbol table --

               state(memory(han-symtab(h-id^eht),us),ds,
                     first(han-instr-s(h-id^eht),
                     addr(han-instr-seq(h-id^eht),1),
                     reset-info(ia),handlers(tht,eht),
                     irs,dis),

          -- handler found in the technique symbol table --

               stop))
```

For a complete specification of the function *handle* see appendix J, where also the unraise instruction that returns control to the interrupted instruction sequence is specified.

12.1.1. Raising exceptions in the statistical language.

At the statistical level a technical statistician may specify tests. The syntax of these tests was already discussed in chapter 7. For example, a statistician may test the significance of an estimated parameter, using "the rule of thumb", as discussed in chapter 4.

```
test section statistical program

TEST
     WHEN   abs(beta) <= 2*sqrt(b_var)

     RAISE  insignificant_coefficients
```

A test at the statistical level is translated into a raise instruction at the kernel level. For the statistical language this translation process is specified in module *Gen-code-test* in the function *gen-code-raise*.

```
[368] gen-code-raise(is,abs-raise(id,expr))
         = add-item(raise(id),
                    add-item(jump-false(1),
                             gen-code(is,expr)))
```

Equation *[368]* specifies that first the code for the boolean expression (*expr*) is generated, then a *jump-false* instruction is added, and finally a *raise* instruction. The raise instruction will only be executed if the given condition is true.

12.1.2. Raising exceptions in function calls.

The kernel calls functions implemented on the higher level programming. Function calls are specified in the kernel instruction *f-call*. If an exception occurs during the execution of a function an exception is raised. The function *excep-raised* in the formal specification signals if an exception is detected in a function call.

```
   excep-raised:  ID # DATA-STACK -> BOOL
```

The function *exception-f* returns the exception identifier

```
   exception-f:   ID # DATA-STACK -> ID
```

The possible states of the processor after the execution of a function call is executed are given in equation *[325]*.

```
[325] execute(state(m,ds,f-call(f-id,i),ia,ri,ht,irs,dis)

               = execute(
                   if(excep-raised(f-id,ds),

                     -- raise the exception --

                     state(m,
                           push(execute(f-id,ds),pop(ds,i)),
                           raise(exception(f-id,ds),ia,
                           ri,ht,irs,dis),
-- execute function and continue  with --
-- the next instruction                --

state(m,
      push(execute(f-id,ds),pop(ds,i)),
      fetch(ia),next(ia),
      ri,ht,irs,dis)))
```

After a successful completion of the *f-call* instruction the data stack is modified. The arguments are popped from the stack and the result is pushed on the stack. If an error is detected an exception is raised using the identifier returned by *f-excep*.

12.2 HANDLING A RAISED EXCEPTION

If an exception is raised, the kernel will look for an exception handler. This can be an exception handler created by either the technical statistician or one of the other experts. In this section we will discuss how technique handlers and external handlers are created.

12.2.1. Exception handlers in the statistical language.

The technical statistician can create exception handlers in a statistical program. The syntax of exception handlers was already discussed in chapter 7. An exception handler consists of an exception identifier and the statements that must be executed if the exception occurs. The statements in an exception handler may only consist of a message statement, that explains to the user of the statistical technique what caused the exception. A call of a matrix inversion, for example, raises an exception if the matrix is near singular. This exception can be handled by an exception handler written by a statistical expert. The statistical expert is often able to give a higher level interpretation of an exception. If, for instance, the inversion routine is called in the ordinary least squares estimation of regression coefficients, the higher level interpretation of near-singularity is multicollinearity. The technical statistician can communicate this interpretation in his exception handler.

```
exception handler section statistical program

WHEN
  near_singular:

  MESSAGE:
    "THE INDEPENDENT VARIABLES IN SOME BOOTSTRAP SAMPLES ARE
     STRONGLY CORRELATED ( MULTICOLLINEARITY). DUE TO THIS THE
     NUMERICAL RESULTS OF THE TECHNIQUE MAY BE INACCURATE."
```

The statistician may also specify that the execution of the statistical technique is stopped after the exception handler is executed by adding a stop instruction at the end of his exception handler.

The fact that an exception can be handled that is raised at an other level of abstraction introduces complications. Sometimes, a technical statistician will use the same function several times in a statistical technique. Consider the following simple statistical program

```
statistical program

  INPUT
            index n,m;
            matrix [1 to n, 1 to n] X;
            matrix [1 to m, 1 to m] Y

       EQUATIONS

       X = inv(X) ; Y = inv(Y);
```

If an exception is raised by the function inverse it can not be determined if it was caused by the first or the second call of the function. In an exception handler, therefore, a technical statistician can specify different handlers for different calls.

```
exception handler section

WHEN near-singular

        CALL(inv,1):
             {
               MESSAGE:" The X matrix is near singular"
               STOP
             }
        CALL(inv,2):
            {
              MESSAGE:" The Y matrix is near singular";
            }
```

In the formal specification of CONDUCTOR, only exception handlers without the call mechanism are specified. An exception handler consists of an identifier, local declarations and statements as given in function *abs-handler* in module *Handler-abstr-syntax* in appendix F.

```
   abs-handler: ID # LOC-DECLS # STATEMENTS -> HANDLER
```

At the kernel level an exception handler is an kernel instruction sequence. The statements in an exception handler are translated and stored in the technique handler table as specified in function *gen-code-hndl* in module *Gen-code-handler-section*.

```
   gen-code-hndl: TECH-HANDLER-TABLE # HANDLER ->
                  TECH-HANDLER-TABLE
```

with equation *[371]*.

```
[371] gen-code-hndl(ht,abs-handler(id,ld,stmts)
        = insert(id,
                 gen-code-stmts(null-instr-seq,stmts),
                 ht)
```

12.2.2. External handlers.

Data experts and computer scientists can also write exception handlers in CONDUCTOR. The only differences with the exception handlers written by the technical statistician are:

- it is not possible for these experts to distinguish between different function calls,
- variables inside a statistical technique are not allowed.

The external handlers are stored in the external handler table. External handlers are created in external sessions. At the kernel level external handlers consists of an instruction sequence and a symbol table.

```
   ext-handl: INSTR-SEQ # TECH-SYMTAB -> EXT-HANDL
```

In module *Gen-ext-handlers* in appendix L it is shown how an exception handler section in an external session is translated into an external handler.

```
   gen-ext-handler: HANDLER     -> EXTERNAL-HANDLER
```

with equation

```
[404] gen-ext-handler(abs-handler(id,ld,stmts))
         = ext-handl(gen-code-stmts(null-instr-seq,stmts),
                     typcheck(abs-handler(id,ld,stmts),empty-mem))

           when  eq-inp-restr-seq(
                     gen-restr-handler(handler(id,ld,stmts),
                                       no-restrictions),
                     no-restrictions) = true
```

The *when* clause specifies that the external handler may not generate input restrictions for the statistical technique.

12.2.3. All handlers.

The technique handler table and the external handler table are combined in the sort *HANDLER-TABLES* in module *Handler-tables* in appendix J. Together they form the collection of all exception handlers available during the execution of a statistical technique.

PART IV

FORMAL SPECIFICATION OF THE USER LANGUAGE AND DATA INTERFACE

13. AN INTERFACE BETWEEN DATA AND STATISTICAL TECHNIQUES

An applied statistician often relies on data collected by specialized agencies. David [1985] has pointed out that additional information is available at these agencies that could lead to more intelligent use of the data. One could think of information such as the sample design, the context in which the observations are made, and the type of measurement instruments used to collect the data. An applied statistician only sees data as published in official publications of the specialized agencies. To make things even more complicated, he analyses his data set using statistical techniques developed by a technical statistician, who looks at the data from yet another perspective. For a technical statistician a data set is a rectangular matrix and a set of assumptions on the type and distribution of the data.
In this chapter the interface between the different views on data is discussed. The main design goals of this data interface are:

- to transfer knowledge available in the data production process in an understandable way to users of data,
- to offer technical statisticians and data experts the possibility to suggest appropriate statistical techniques if inconsistencies are detected in data.

13.1. THE DATA PRODUCER'S VIEW ON DATA

Producing statistical data is a complicated and time consuming job. Often it is impossible to make exact measurements. Therefore, sample techniques are used to make estimates of the required information. The resulting figures are of course conditional on the sample design chosen. Besides the sample design, also the measurement instrument used to collect the data and the context play an important role. David [1985] sees a set of observations **X** as conditional on the design **D**, the instrument **S** and the context **C**.

$$\{X \mid D,S,C\}$$

Comparing two observations measured at different moments in time **T** and **T'**, one compares

$$\{X_T \mid D_T, S_T, C_T\}$$

and

$$\{X_{T'} \mid D_{T'}, S_{T'}, C_{T'}\}$$

Of course, one hopes for the best that **D**, **S** and **C** do not change over the sample period, but using estimation periods of several years this seems hardly realistic.

To get an indication of the quality of the data, a user must have access to all information gathered in the data production process. Also evaluation functions combining X and external information, calculated to give an indication of the quality of the data, such as bias, mean square error and reliability, etc., should be part of the documentation for any data set.

13.2. THE USER'S VIEW ON DATA

An applied statistician, with a little common sense, will not try to collect large data sets on his own. By the time he is finished with the data production, the question he initially wanted to answer will surely be out of date or even no longer existent. An applied statistician can spend only limited time worrying about all the details involved in the gathering of data. The view of the user of data, therefore, is restricted to the resulting data of a data production process.

13.3. THE TECHNICAL STATISTICIAN'S VIEW ON DATA

Statistical techniques are developed to make estimates of parameters in theoretical models which must explain the observed data. Griliches [1984], for example, classifies the data along several different dimensions:

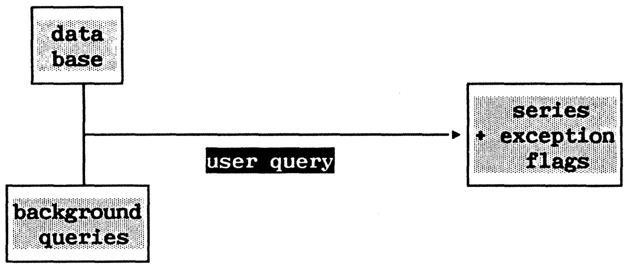
- objective versus subjective: for example, prices versus expectations about them,
- type and periodicity: for example, time series versus cross-section; monthly, quarterly or annual data; nominal, ordinal, interval or cardinal data,
- level of aggregation: for example, data on individuals, firms, districts or states,
- quality: for example, the reliability or validity of data,
- level of fabrication: for example, primary or secondary data.

For each of the different categories of data, special statistical techniques are developed. An excellent overview of the econometric techniques that are developed for different dimensions and types of economic data is given in Judge et al. [1980].

If a statistical technique is used, it is assumed that data set has certain properties. For example, the technique might expect outliers, missing data, ordinal data or cardinal data. If the data set has other properties the statistical inference with the use of that particular technique may be incorrect. The technical statistician assumes that the user of his statistical technique is aware of this fact. And in traditional statistical software indeed this responsibility lies completely in the hands of the user. For the technical statistician the properties of the data are not a problem but a premiss.

13.4. THE INTERFACE BETWEEN THE DATA PRODUCER'S VIEW AND THE USER'S VIEW

Information and documentation collected during the data production process can be stored in a relational database system. Though other database models exist, the relational database model has the advantage of clarity and ease of use (see Ullman [1980] and Kroenke [1983]). The software produced in CONDUCTOR, can, on request of the user, retrieve data from this database. When data is retrieved, the additional information on the retrieved data is checked by a background implemented by the data expert. If an inconsistency is detected by the background query, this inconsistency is signalled by adding an exception flag to the retrieved series.



Example 13.1:

A large organization produces data on consumption, production and export in the Netherlands. Besides the figures published by the organization,

Table 1. published data

Name	Sample	Data
consumption	1982	1312.4
production	1980	400.7
consumption	1980	2000.4
export	1979	487.9
production	1981	500.5
consumption	1981	900.8
consumption	1983	1200.4
production	1982	700.3

also additional information is available about the sample design used to estimate the figures. This information is stored in three tables (called relational schemes in relational database theory). The first table contains the estimated figures.

Table 2. Design.

Name	Sample	Design number
consumption	1982	DS2
production	1982	DS2
consumption	1980	DS1
export	1979	DS1
consumption	1981	DS1
consumption	1983	DS2
production	1981	DS2
production	1982	DS2

Table 3. Design description.

Design_number	Description
DS1	....
DS2	....

In the following queries two basic operations are applied: selection and projection. The first operation simply means that all rows of the table are selected that satisfy a given condition. Projection stands for extracting given columns from a table. In the queries projection is indicated by the operator π, and selection by the operator σ. The published data on consumption in the period 1980 and 1982 is retrieved by the query

$$\pi_{\text{Rawdata}}(\sigma_{\text{name=consumption,1980<=Sample<=1982}}(\text{Observations}))$$

resulting in the table

Data
2000.4 900.8 1312.4

The background query that checks if the same design is used over the sample period

$$\pi_{\text{Design\_name}}(\sigma_{\text{Xname=x1,1980<=Sample<=1982}}(\text{Documentation}))$$

returns table

Design_name
DS1 DS2

The resulting relation contains more than one row and an exception flag **"design-unequal"** is added to the series, indicating that the retrieved sample is based on different designs DS1 and DS2.
The user of software produced by CONDUCTOR will not see much of these queries. In the resulting software he simply sets the sample and requests retrieval of the series consumption

```
user session

>u:  set sample 1980 to 1982
>u:  retrieve consumption
```

The database design and the background queries are completely hidden for the user.
{ end example 13.1. }

In the formal specification of CONDUCTOR the user data is represented by the sort *SERIES*, specified in module *Series*. The observations are represented by the sort *SCALAR-SEQ*, the sample description is represented by the sort *CONST-RANGE-SEQ*, and the exception identifiers by the sort *ID-SEQ*.

ser: SCALAR-SEQ # CONST-RANGE-SEQ # ID-SEQ -> SERIES

A well-formed series *ss* satisfies the restriction that the number of samples described in the constant range sequence *crs* is equal to the number of observations in the scalar sequence *scs*.

[19] ser(sas,scs,ids) = wf-ser(scs,crs,ids)

when eq(n-of-items(scs),length(crs)) = true

and

[20] ser(sas,scs,ids) = error-ser

when eq(n-of-items(scs),length(crs)) = false

The retrieval process of CONDUCTOR is specified in module *Database-interface* in the function *query*.

```
data-query: DATA-BASE # ID # CONST-RANGE-SEQ
            -> USER-DATA
```

which invokes the functions

```
bg-query:   DATA-BASE # ID # CONST-RANGE-SEQ
            -> ID-SEQ

retrv-data: DATA-BASE # ID # CONST-RANGE-SEQ
            -> SCALAR-SEQ
```

A query of a database db for a series with identifier *id* over the sample given by constant range sequence *crs* will invoke a background query to check the consistency of the data.

```
[416] data-query(db,id,crs)  = u-data(
                                  series(retrieve-data(db,id,crs),
                                         crs,
                                  bg-query(db,id,crs)))
```

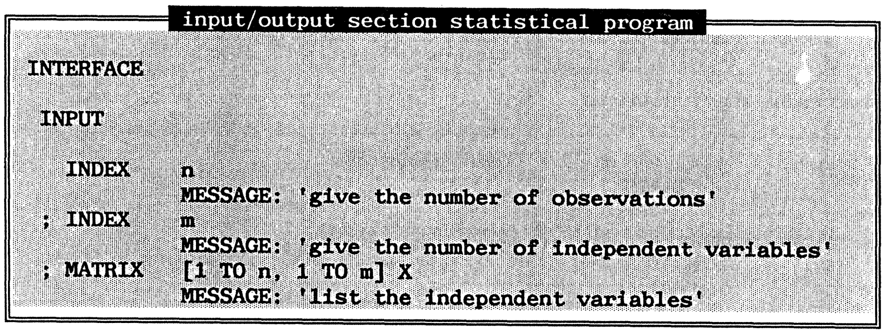
The data retrieval process is not further specified in CONDUCTOR. It is assumed that a database, with the appropriate queries, can be constructed.

13.5. THE INTERFACE BETWEEN THE USER'S VIEW AND THE TECHNICAL STATISTICIAN'S VIEW ON DATA

The series retrieved from the database can be used in statistical techniques, that is implemented by a technical statistician. The observations in the series will form the columns of the matrix.

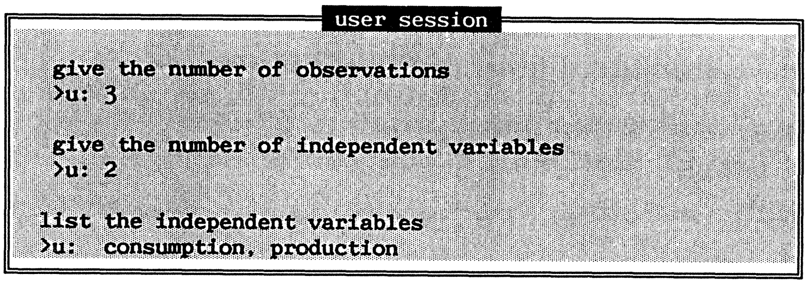
Example 13.2:

Assume that the user has retrieved the consumption and production series from the database in example 1, and he wants to use these series in a statistical technique. The user interface of the statistical technique is specified by the technical statistician in the INPUT section of a statistical program specifying the technique



```
input/output section statistical program
INTERFACE
 INPUT
   INDEX    n
            MESSAGE: 'give the number of observations'
 ; INDEX    m
            MESSAGE: 'give the number of independent variables'
 ; MATRIX   [1 TO n, 1 TO m] X
            MESSAGE: 'list the independent variables'
```

When the user calls the statistical technique n,m and X must be initialized. The user is prompted to enter values for these variables.



```
user session
  give the number of observations
  >u: 3

  give the number of independent variables
  >u: 2

 list the independent variables
 >u:  consumption, production
```

At the kernel level the prompting of user information and the loading of the matrix X is seen as the execution of a **USER_LOAD** instruction. The input of the variables **n**, **m** and **X** at the kernel level is represented by the instruction sequence

USER_LOAD n	**USER_LOAD m**	**CHECK_RESTR**	**USER_LOAD X ..**

$\longrightarrow$

sequential execution

To calculate the required output of a statistical technique other instructions will follow the USER_LOAD instructions. The result of the USER_LOAD instructions in the above example is a (3 x 2) matrix with the observations on consumption and production in the columns.

$$X = \begin{bmatrix} 2000.4 & 400.7 \\ 900.8 & 500.5 \\ 1312.4 & 700.3 \end{bmatrix}$$

{ end example 13.2 }

A matrix in CONDUCTOR is specified in module *Matrices* in the function *mat*.

mat: SCALAR-SEQ # CONST-RANGE-SEQ -> MATRIX

A matrix consists of a scalar sequence containing the values of the matrix elements and a constant range sequence, describing the dimensions. A matrix is well-formed if the product of the length of the dimension ranges *crs* is equal to the number of items in the scalar sequence *scs*.

[16] mat(scs,crs) = wf-mat(scs,crs)

when eq(n-of-items(scs),product(crs)) = true

A complete specification of all function and sorts involved can be found in appendix D. The USER_LOAD command has a special facility to transfer sequences of series into a matrix as specified in module *Series-matrix-interfaces* in function *data-matrix*.

data-matrix: ID-SEQ # USER-SYMTAB -> MATRIX

The details of this operation are left unspecified. When a series, with an exception flag attached to it, is loaded in a matrix an exception is raised. The software will look for an exception handler for this particular exception. This behaviour is specified in equations for the user-load command in module *Processor* in appendix J.

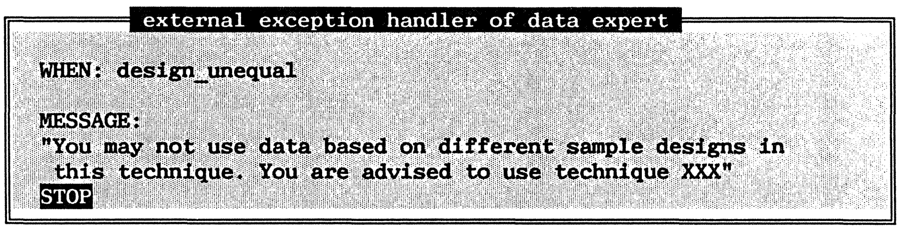
```
[312] execute(state(m,ds,user-load(id),ia,ri,ht,irs,dis)

              = execute(
                 if(user-load-excep(id,m),

                   -- raise the exception --
                   state(user-insert(id,m),ds,
                         raise(load-excep-id(id,m),ia,
                         ri,ht,irs,dis),

                   -- else load the variable and continue  --
                   -- with the next instruction            --


                   state(user-insert(id,m),ds,
                         fetch(ia),next(ia),
                         ri,ht,irs,dis)))
```

Example 13.3:

In example 13.2 we loaded two series in a matrix X. The consumption series, however, was produced using different sample designs. This inconsistency was detected by the background query and a exception flag "**design_unequal**" was added to the retrieved series. When the series is loaded in the matrix, the exception flag "**design_unequal**" is raised. The data expert may have written an external exception handler for this exception

```
external exception handler of data expert

WHEN: design_unequal

MESSAGE:
"You use a series which is based on different sample designs.
 Be careful with the interpretation of the results based on
 these series"
```

After this warning the execution of the interrupted statistical technique continues. For a particular statistical technique a statistician may overrule an external exception handler by writing a exception handler in the specification of the statistical technique.



```
external exception handler of data expert

WHEN: design_unequal

MESSAGE:
"You may not use data based on different sample designs in
 this technique. You are advised to use technique XXX"
STOP
```

When the exception **"design_unequal"** occurs the statistical technique is stopped and the user is advised to use the more appropriate technique XXX.

{ end example 13.3.}

14. THE USER LANGUAGE

In CONDUCTOR, data experts, technical statisticians and computer scientist each can add their knowledge to the software in a language close to there problem domain. To demonstrate the use of software created in such an environment, a user command language is developed. To keep the prototype of CONDUCTOR simple, it has not been attempted to create a revolutionary new user interface. The user command language is needed only to demonstrate the fact that the software implemented by the different experts can be used by an applied statistician. In the prototype of CONDUCTOR the user language consists of commands to

- declare and initialize input variables,
- retrieve series from a database,
- call a statistical technique.

Of course, many facilities should be added in order to compete with the interfaces of modern statistical packages. One could think of help facilities, multiple screens, graphical facilities
etc. Adding these facilities however is merely a matter of hard work and is not considered to be within the scope of this book.
After giving an informal description of the user language in the first three section of this chapter, we will discuss its formal specification.

14.1 INITIALIZING VARIABLES

The input variables of a statistical technique must be initialized by the user of the statistical technique. In the user language, therefore, an applied statistician must be able to declare and initialize variables of the types scalar, index, boolean, vector, matrix or matrix. In the user session given below, a scalar **x** and a matrix **m** are declared. In contrast to the statistical language the dimension bounds of matrices in a user session are fixed. The variables are initialized in a user session using the **set** command. The values of **x** and **m** can be displayed using the **display** command.

```
                        user session
>u:  scalar x
>u:  matrix [1 to 2, 1 to 2] m
>u:  set x  12.4
>u:  set m

     Enter the values of m row by row:

>u:  1.0 2.0
>u:  3.0 4.0
>u:  format  10  2
>u:  display x

     x =       12.40

>u:  display m

     m =

          1.00       2.00
          3.00       4.00
```

In the prototype of CONDUCTOR the user may also change the format of the reals using the **format** command. The format instruction in the example above changes the output format of a real to a total of 10 positions and 2 position after the decimal point.

14.2 INITIALIZING SERIES

In statistical software, series of observations play an important role. At the user level a series can be initialized in two ways. It can initialized in a way similar to other variables, or it can be retrieved from a database. To initialize a series the user must set the current sample in his session. After this must declare the identifiers of the series. In the example session below a series **consumption** is declared over the sample 1952 to 1958 and 1962 to 1964.

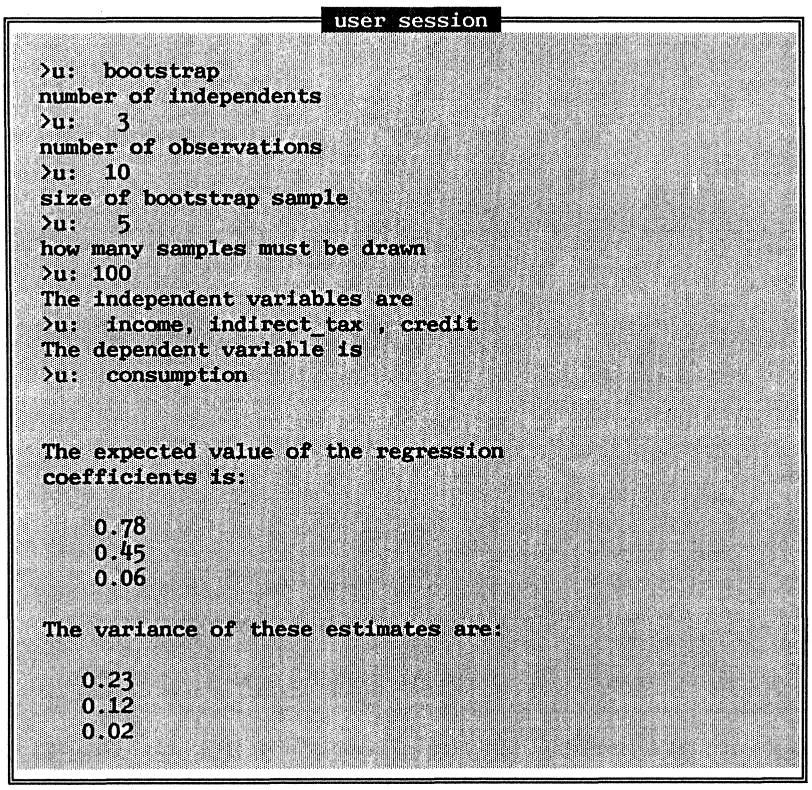
user session

```
>u: set format  1952 to 1958, 1962 to 1964
>u: series consumption
>u: set consumption

    Enter the observations of the series consumption

>u: 1.2 2.4 5.6 5.9 3.5
>u: 1.2 3.4 1.9 3.4 0.2
>u: display consumption

    The observations of consumption are

     1952       1.20
     1953       2.40
     1954       5.60
     1955       5.90
     1956       3.50
     1957       1.20
     1958       3.40

     1962      1.90
     1963      3.40
     1964      0.20

 >u: set sample 1957 to 1960, 1963 to 1965
 >u: display consumption

    The observations of consumption are

     1957      1.20
     1958      3.40
     1959      NA
     1960      NA

     1963      3.40
     1964      0.20
     1965      NA
```

The values of this series are initialized using the **set** command. The resulting series **consumption** is displayed using the **display** command.The display command always prints the series for the current sample. If the current sample contains missing values the symbol NA is printed. The other way to initialize a series is by retrieving it from a database, as discussed in chapter 13.

14.3 EXECUTING A STATISTICAL TECHNIQUE

After having initialized variables a user can call a statistical technique specified by a technical statistician. The user only has to enter the name of a statistical technique. For example if a technique called **bootstrap_olsq** is specified by the technical statistician the user can apply this technique to his data set. The statistical technique will prompt the user to initialize the input variables. Then it will calculate the specified statistics and return the results.



```
>u:  bootstrap
number of independents
>u:   3
number of observations
>u:  10
size of bootstrap sample
>u:   5
how many samples must be drawn
>u: 100
The independent variables are
>u:  income, indirect_tax , credit
The dependent variable is
>u:  consumption


The expected value of the regression
coefficients is:

     0.78
     0.45
     0.06

The variance of these estimates are:

    0.23
    0.12
    0.02
```

14.4. FORMAL SPECIFICATION OF THE USER LANGAUGE

In the formal specification of CONDUCTOR the **display** and **format** commands are omitted. The abstract syntax of the other user language commands can be found in module *User-programs* in appendix L in the functions.

```
user-decl:   USER-TYPE  ID      -> USER-COMMAND
set:         ID  # USER-DATA    -> USER-COMMAND
set-sample:  CONST-RANGE-SEQ    -> USER-COMMAND
retrieve:    ID                 -> USER-COMMAND
call-tech:   ID                 -> USER-COMMAND
```

The semantics of these instructions is defined in module *Resulting-software*. In this module it is specified how a user command modifies the state of a user program given the set of implemented statistical techniques and external handlers.
The state of a program is defined by the user symbol table, represented by the sort *USER-SYMTAB*, and the current sample, represented by the sort *CONST-RANGE-SEQ*. Recall that the user symbol table contains both the type description and the value of each variable declared in a user session.

```
user-state: USER-SYMTAB # CONST-RANGE-SEQ -> USER-STATE
```

The semantics of the set sample instruction is defined in function *exec-user-com* in module *Resulting-software*. If the user state is determined by the user symbol table *us* and the sample *crs1*, the execution of a set-sample instruction results in a user state where the sample description *crs1* is replaced by *crs2*.

```
[410]  exec-user-com(user-state(us,crs1),it,set-sample(crs2))
              = user-state(us,crs2)
```

The specification of the semantics of the other user commands can be found in appendix L.

PART V

IMPLEMENTATION AND EVALUATION OF CONDUCTOR

15. IMPLEMENTATION OF CONDUCTOR

CONDUCTOR is implemented in C and runs under both the UNIX and the MS-DOS operating systems. Given the formal specification of CONDUCTOR, still many practical problems had to be solved to get a program performing the specified tasks. In this chapter we will discuss some of these problems to wit the implementation of the formal modules, the hashing mechanism for associating information with identifiers, storage problems, and error recovery.

15.1. IMPLEMENTATION OF FORMAL MODULES

An implementation of a formal module consists of
- *the internal definition file*; this file contains the definitions of the hidden data types; such data types are hidden from the other modules in the system,
- *the external definition file*; this file contains the definitions of the data types that are visible for the other modules,
- *the C-code file*; this file contains the C-implementation of the functions.

The data types specified in the internal definition files and the implementation of the functions in the C-code files can be changed without influencing other modules. The implementation of the module Tables, for example, consists of the files tables.def containing the internal definitions, tables.h containing the external definitions, and tables.c containing the C code.

The modules that describe the separate passes of the compilation and/or interpretation of the languages are not implemented using the method described above. The parser for the languages in CONDUCTOR is implemented using the parser generator *yacc* (see Johnson [1975]). Using *yacc*, all the separate passes in the formal specification of the statistical language (type checking, input restriction generation and code generation) are implemented in one pass. The formal specification in these modules describes at which point actions must be added to the syntax rules in *yacc*.

<u>Example 15.1</u>: Implementation of the assignment statement in the statistical language.

The concrete syntax of an assignment statement in the statistical language reads

```
assignment  -> variable ':=' expression
```

the abstract syntax is specified in the function *abs-assgn*.

```
abs-assgn:      VARIABLE   # EXPR   ->  STATEMENT
```

The type checking of an assignment statement is specified in module *Stat-check-statements* in equation

```
[156]  type-check-stmnt(abs-assgn(var,expr),tst)
           = eq-type(var-type(var,expr),
                     expr-type(expr,tst))
```

where *tst* is the technique symbol table. The equation specifies that for each assignment statement it must be checked that the type of the left-hand side variable is equal to the type of the expression.
The generation of input restrictions for a assignment statement is specified in module *Gen-restr-statements* in appendix I and the code generation in module *Gen-code-statements* in appendix K.
The formal specifications in these modules are reflected in the following rules for *yacc*

```
assignment: variable BECOMES expression
            {
             -- code to check the type restrictions & --
             -- add store instruction for left-hand   --
             -- side variable                         --
            }
```

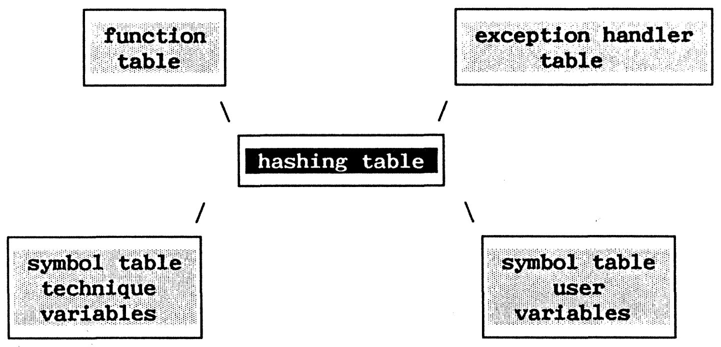
No actions are added for the generation of input restrictions, because the only input restrictions for assignment statements are the input restrictions generated by the variable and the expression in the assignment statement. The actions which generate input restrictions for variables and expressions, of course, accompany the syntax rules for variables and expressions.
{ end example 15.1.}

15.2. FINDING INFORMATION THROUGH IDENTIFIERS

In most tables in the formal specification we used identifiers to locate an entry in a table. Information on, for example, exception handlers, variables and functions is located through an identifier. A simple hashing scheme, called 'open hashing' as defined in Aho, Sethi and Ullman [1986], is used to search identifiers. Such a scheme consists of:

- a hash table: a fixed array of **m** records (hash entries) pointers.
- hash entries organized into **m** separate linked lists called buckets (some buckets may be empty).

A hashing function can calculate in which bucket the hash entry of an identifier can be found. For each identifier the character string and the type of the identifier is stored in a hash entry. An identifier can be the name of a statistical technique, a keyword, a function name, the name of a variable or the name of an exception. Additional information for each of these types of identifiers is stored in separate tables. The hash entries contain pointers to the table entries for each of these identifier types and vice versa.



15.3. STORAGE PROBLEMS

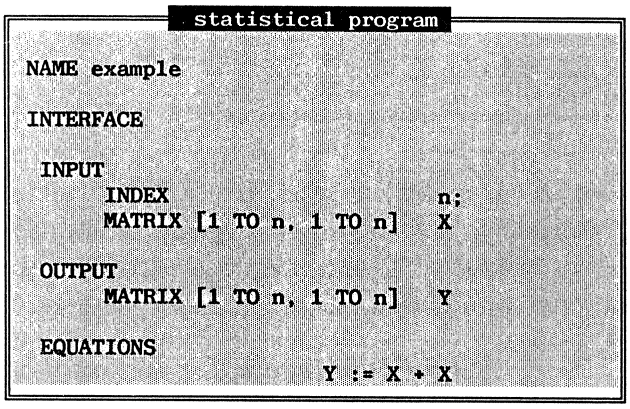
Making abundant copies of large matrices during the execution of a statistical technique will rapidly create storage problems. Sharing storage instead of making complete duplicates of matrices, reduces the storage problem considerably. A matrix is implemented as a record containing a pointer to the data, and a pointer to the dimension descriptions. The data and the dimension descriptions can be shared by more than one matrix. The pointers in the record of several matrices may point to the same address. Each matrix record, therefore, contains a flag indicating if it is the original record or a copy. To avoid dangling pointers one of course must be sure that the original matrix is not removed as long as copies exist. In C the definition reads

```
typedef struct matrix
             {
              bool:    original;
              float:   *data;
              pointer: dims;
             }
```

When an input matrix at the statistical level is initialized by a user, or a matrix is pushed on the data stack, a copy is made of the matrix involved. In the first case the original matrix is stored in the user symbol table and a copy is placed in the technique symbol table. In the second case the original matrix is stored in the technique symbol table and a copy is pushed on the data stack. In both case the copy is removed before the original matrix can be removed. A complete copy of the matrix, including a full copy of the data and dimension descriptions, is made when a matrix at the data stack is stored in the technique symbol table, or when a matrix operator or function creates a new matrix.

Example 15.1: storage of matrices.

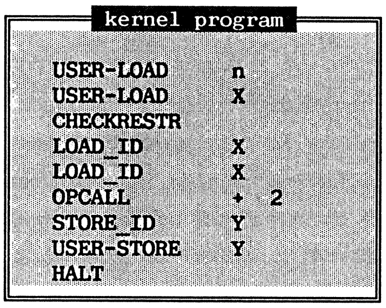
A statistical program



statistical program

```
NAME example

INTERFACE

 INPUT
      INDEX                    n;
      MATRIX [1 TO n, 1 TO n]  X

 OUTPUT
      MATRIX [1 TO n, 1 TO n]  Y

 EQUATIONS
                        Y := X + X
```

is represented at the kernel level as



kernel program

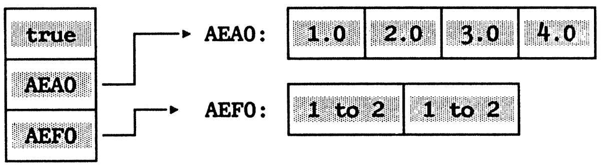
```
USER-LOAD     n
USER-LOAD     X
CHECKRESTR
LOAD_ID       X
LOAD_ID       X
OPCALL        +  2
STORE_ID      Y
USER-STORE    Y
HALT
```

In a user session a user can declare and initialize a matrix MAT and call the technique **example**.

```
                    user session

  u>: matrix [1 to 2, 1 to 2] MAT
  u>: set MAT

 initialize MAT row by row

  u>: 1.0 2.0
  u>: 3.0 4.0

  u>: example

          n =?
  u>: 2

          X =?
  u>: MAT
```

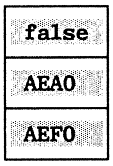
The matrix MAT is stored in a record of type matrix. This record contains a boolean flag, indicating that it is an original matrix, and pointers to the data and the dimension description. The values of the pointers, such as AEAO and AEFO are addresses on an imaginary computer.

record matrix MAT



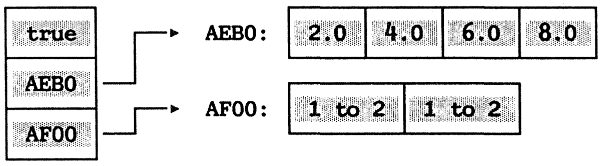
When the user runs the statistical technique 'example' he must initialize the input variables. The initialization is represented in the kernel by the USERLOAD-instruction. The USERLOAD-instruction stores a copy of the original matrix **MAT** in the entry of the matrix **X** in the technique symbol table. The pointers in the record of **X** point to the same data and dimension description as the pointer in the record of matrix **MAT**. A flag indicates that the matrix is a copy.

record matrix X



After the input restrictions of the statistical technique are checked, two **LOAD_ID** instructions push copies of the matrix **X** on the data stack. Now three copies of the matrix **MAT** exist, the data and the dimension description however are stored only ones. The OPCALL instruction uses the two copies to create the sum of the matrix. This instruction creates a new matrix containing the resulting sum of **X** + **X**.

record matrix X + X



The storage used for the records of the two copies of the **X** matrix on the data stack is freed. The **STORE_ID** instruction stores the resulting matrix in the entry of **Y** in the technique symbol table.
{ end example 15.1.}

Besides avoiding copying the data of matrices, the storage problems can further be reduced by making use of the structure of the matrices. One can distinguish:

- *A normal matrix.* Every element of the matrix may be different and must be stored separately,
- *A sparse matrix.* The matrix contains many zero elements and only the position of an element in combination with the value are stored,
- *An identity matrix.* All elements on the main diagonal are one. Elements not on the main diagonal are zero,
- *A symmetric matrix.* Element[i,j] is equal to element[j,i],

- *A triangular matrix.* All elements below or above the main diagonal are zero,
- *A constant matrix.* All elements of the matrix are equal,
- *A large matrix.* The matrix is so large that it is not stored in central memory,
- *A binary matrix.* The elements of the matrix are either 0 or 1.

On the one hand, special storage types will complicate the implementation of the matrix functions. For every type of matrix, separate functions must be written. On the other hand, it will increase the speed of the matrix functions dramatically. In CONDUCTOR, special storage types for matrices are not implemented.

15.4. ERROR RECOVERY

CONDUCTOR is an interactive system. It is inevitable that users make mistakes during interactive sessions. Therefore the system should offer the user the possibility to undo the effect of erroneous commands. Important in this respect is that the system should give proper error messages, making it possible for a user to find errors during a session. Once the error is located the user must be able to undo the effects of the command and restart the system as efficient as possible.
A model for error recovery is given in Archer et al. [1984]. An interactive session S is viewed as a sequence of commands

$$\underbrace{c_1 \; ; \; \ldots\ldots \; ; \; c_n}_{S}$$

Assume that after j commands an error is detected. This leads to the situation that j commands are already executed. This set of executed commands is called E. The remaining commands are in P the set of pending commands

$$\underbrace{c_1 \; ; \; \ldots\ldots \; ; \; c_{j-1}}_{E} \; ; \; \underbrace{c_j \; ; \; \ldots\ldots \; ; \; c_n}_{P}$$

Modification of S will lead to S', a modified session. This session can be separated into a set U, a prefix of S, which remained unchanged and a set **M** containing the modified commands.

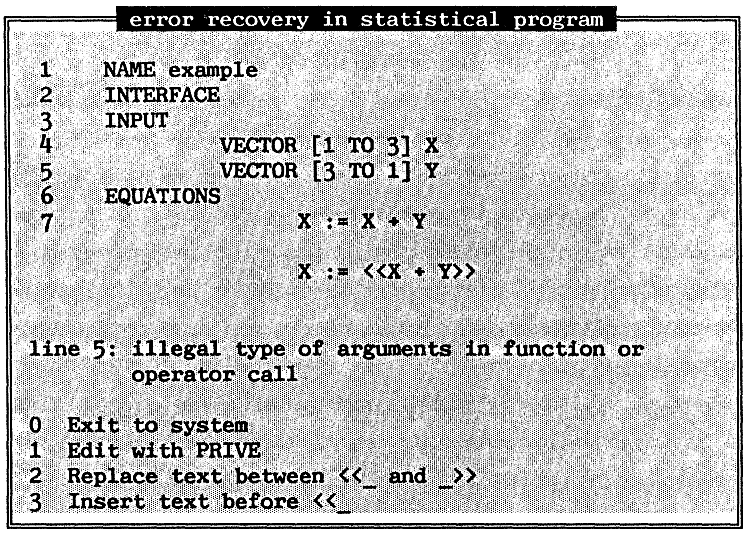
$$\underbrace{\mathbf{c}_1 \; ; \; \ldots\ldots \; ; \; \mathbf{c}_{i-1}}_{\mathbf{U}} \; ; \; \underbrace{\mathbf{c'}_i \; ; \; \ldots\ldots \; ; \; \mathbf{c'}_n}_{\mathbf{M}}$$

In error recovery clearly the most complicated situation is the restart of the system with command $\mathbf{c'}_i$ if $\mathbf{i} < \mathbf{j}$. The problem is in this case how to return to the state prior to $\mathbf{c}_i$. The simplest solution is a complete restart. In Archer et al. [1984] it is suggested that the introduction of checkpoints ('saved intermediate states') is profitable. If in our model we had saved the state after command $\mathbf{c}_{i-1}$ was executed we could restart the system with command $\mathbf{c'}_i$. Saving all states after each command however is space and time consuming, therefore, only a few checkpoints should be used in a system.
In CONDUCTOR error recovery is possible. When an error occurs, CONDUCTOR displays the last commands entered by the user and indicates the position where the error occurred. Also the possible options to modify the commands are given. In an external handler session and a statistical session four options are available
1. Stop executing CONDUCTOR and return to the operating system.
2. Edit the commands with the use of an editor.
3. Replace the text that caused the errors.
4. Insert text before the text that caused the errors.

In a user session also the last command can be aborted. Editing commands in a user session, however is not possible. Options 3 and 4 are only available if the error is still in the input buffer of the system. After the correction is made the system will start from the latest saved checkpoint.
The error recovery is build into the parser generated by the *yacc* parser generator. A check point in *yacc* amounts to an error flag on the parse stack. Checkpoint are placed at the beginning of sessions, the beginning of expressions in the statistical language, and at the beginning of user commands. These checkpoints allow the user to recover faster from minor errors.

Example 15.3:

If a technical statistician makes an error in a expression CONDUCTOR will give the following error message:



```
error recovery in statistical program

 1     NAME example
 2     INTERFACE
 3     INPUT
 4             VECTOR [1 TO 3] X
 5             VECTOR [3 TO 1] Y
 6     EQUATIONS
 7                     X := X + Y

                       X := <<X + Y>>


line 5: illegal type of arguments in function or
        operator call

0  Exit to system
1  Edit with PRIVE
2  Replace text between <<_ and _>>
3  Insert text before <<_
```

If the user chooses option 2 or 3 and repairs the error, for example by replacing **Y** by the transpose **Y'**, only the expression is recompiled. If however the error was due to improper declaration of **Y** the program is recompiled.
{ end example 15.3.}

16. THE USE OF FORMAL SPECIFICATIONS IN CONDUCTOR

In this book a formal specification of a large software project is discussed. Of course, the specification presented is a product that has gone through many stages. In this chapter we will discuss our experience in using the specification formalism **ASF**, and demonstrate how it helped to modularize and define CONDUCTOR. We also will discuss a major deficiency of **ASF**: the impossibility to hide error cases in specifications. And finally we will discuss the problem of keeping the formal specification consistent with the prototype.

16.1. USING ASF IN THE DESIGN OF CONDUCTOR

In a large software project one can not just start writing procedures in a higher-level programming language, and hope that, at the end, the software performs the ideas one had in mind. The problem is that half way through the project it might be discovered that previously written procedures should be modified, and that the efforts in writing the procedures were a waste of time. As in all large projects, one has to carefully plan before one starts working. In the CONDUCTOR project this plan is described in **ASF**. This formalism provides the means to give a rigourous specification of CONDUCTOR, and one can abstract from many of the problems one has to tackle in a higher-level programming language. A few of these problems were discussed in the previous chapter. The modules in the formal specification can be seen as an abstract descriptions of the higher-level programming language implementation of the modules

(hlpl-modules). Of course, this abstraction is only useful if the formal specification of a module is considerable smaller than the hlpl-module. The hlpl-modules in the CONDUCTOR project were, on average, a factor 10 larger than their abstract counterparts.
During the design of CONDUCTOR **ASF** was used as a bookkeeping mechanism. Modules were specified at a certain stage in the design and later on, functions and sorts were added, or the specifications of existing functions and sorts were modified. Thus we knew all functions and sorts of a module before we implemented it. For instance, the module *Ranges*, given in appendix E, is described in 30 lines in the formal specification. The implementation of this module in the programming language C (Range.c) consists of 340 lines of code, including comments. In the module Range.c functions had to be added for storage management and I/O of ranges. In the first version of module *Ranges* the functions *range-plus*, *range-min* and *range-mul* did not exist. Later on, when we designed the static type checking and the generation of input restriction, these functions turned out to be needed. Thus, the formal module *Ranges* provided an excellent abstract model that could be used during the implementation of the hlpl-module Range.c.

16.2. MODULARIZATION

In chapter 3 of this book we listed the goals of modularization as given by Parnas [1985]. We will discuss how **ASF** contributed to each of these goals.

16.2.1. Small modules.

The first goal of modularization, as given by Parnas [1985], is that each module should be small so that it can be understood fully. The formal specification language **ASF** does not really force such modularization. One could think of the horrible solution to specify the CONDUCTOR in just one module with hundreds of sorts and functions. By restricting the number of newly exported sorts and functions in a module - the average number of newly defined sorts in the specification of CONDUCTOR is near 1 and the newly defined functions near 5 - one obtains **ASF** specifications consisting of simple modules.

16.2.2. Implementation changes.

One must be able to change the implementation of a module without affecting other modules. Again **ASF** is, albeit indirectly, of help in this respect. The formal specification contains a complete specification of the interfaces of all modules involved. If the modules are indeed implemented using only the knowledge of the interfaces of imported modules, the implementation of modules can be changed without influencing other modules.
In our implementation (see 15.1), the data types defined in internal definition files and the implementation of the functions in the C-code files can be changed without influencing other modules. Changes in the parameter lists or the result of functions in the C-code files and changes in the external definition files do influence other modules because they are part of the interface of a module.

16.2.3. Unlikely changes.

Only unlikely changes should require changes in the interface of the modules. The benefit of **ASF** in this respect is that the use of formal specification languages forces the system analyst to define the system thoroughly before he implements it. Evidently this does not guarantee that later on in the development process some aspects of the problem change or new wishes occur. If, however, simple modules are specified as discussed in 16.1.1, a change in the interface of a module will only add new functions to the interface. In using ASF to specify CONDUCTOR, we found that the specification of basic sorts such as the data types (appendix D), the type descriptions (appendix E), the abstract syntax of the statistical language (appendix F) hardly changed during the specification process. More delicate parts like the symbolic evaluation of index variables (appendix I) and the exception handling mechanism (appendix J) were changed several times before they got their final shape. This indicates, that if we had started the implementation of these modules too early, we would have been forced to make changes in these modules. By using **ASF** as a design language, the need for such changes is reduced considerably.

16.2.4. Major changes.

Major changes in the software should only result in independent changes in the individual modules. A major change in the software leads to a major change in the specification of the software. In the formal specification the following changes can be distinguished:
- adding new modules,
- changing the equations that specify sorts and functions in the module,
- modifying the exported sorts and functions of the module.

If a major change only consists of the addition of modules or the changes of equations, this change can be regarded as a complete independent modification of modules. Of course the use of **ASF** does not guarantee that no modifications of module interfaces are necessary. In the development process of CONDUCTOR these changes turned out to be inevitable. Most modifications, however, were either changes in the parameter lists of functions or the addition of new functions.

16.2.5. The responsibility of a module.

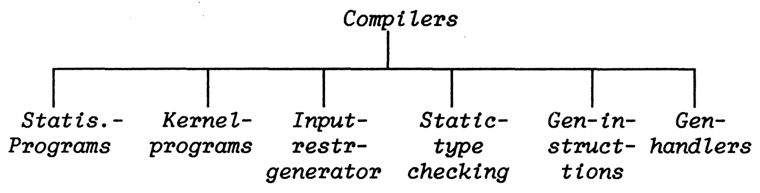
A software engineer should be able to understand the responsibility of a module without understanding the details of its internal design. If a software engineer is capable of reading the formal specification language **ASF**, the formal specification of a module gives him an exact description of its responsibility. For example, the responsibility of hlpl-modules, such as, Range.c, Type.c, Table.c ar.d Seq.c can be understood be studying the formal specification as given in the formal modules *Ranges*, *Types*, *Tables* and *Sequences*.

16.2.6. Relevant modules.

A reader with a well-defined question is able to find the relevant modules in the formal specification. The formal specification introduces a strict hierarchy in the modules. A module can be understood by studying its formal specification and the specification of its imported modules.

For example, the module *Compilers* can be understood by studying besides this module, module *Statistical-programs* (the source language), module *Kernel-programs* (the target language), module *Input-restr-generator* (the input restriction generator), module *Static-type-checking* (the static type checking), module *Gen-instructions* (the generation of kernel instructions) and module *Gen-handlers* (the generation of exception handlers).

figure 16.1. The Compiler



16.2.7. The number of branches.

The submodules imported by a module should not have overlapping responsibilities. In order to avoid this, the number of submodules should be small. In **ASF** this goal is reached by keeping the number of imported modules small. The average number of imported modules in the specification of CONDUCTOR is between 2 and 3. **ASF** does not force the number of branches to be small; this remains the responsibility of the system analyst.

16.3 FORMAL DESCRIPTION

We also used the formal specification technique to avoid "the seven sins" as discussed in Meyer [1985] and to get an unambiguous definition of CONDUCTOR.

16.3.1. Noise.

Noise is the problem that the specification contains elements that do not add information. Deliberately some noise is left in the specification of module *Scalars*. This module contains the exported functions *add*, *sub*, *mul* and *div*. These functions do not add any information, because they are not imported by any other module in the specification, nor are they used in the equations of the module *Scalars*. So neither in the definition of scalars nor in the use of scalars these functions are needed. We suggest to add a simple bookkeeping mechanism to the implementation of **ASF** to list all unused exported functions.

16.3.2. Silence.

It is even more difficult to detect whether some aspects of the problem are not specified. Examples of silence in the specification are the database, and the inconsistencies and redundancy in the input restrictions. The database is, in the specification, represented by the sort *DATABASE* and a function *current-db*. How a database is constructed and how additional data is stored and interrogated is left unspecified. The problem with input restrictions is, that CONDUCTOR may generate restrictions that are contradicting, such as,

$$n - 1 >= 0 \qquad \text{and} \qquad - n - 1 >= 0$$

or redundant, such as,

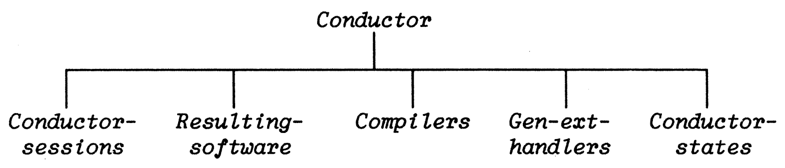
$$n - 7 >= 0 \qquad \text{and} \qquad n - 3 >= 0$$

In the first case, the contradicting restrictions, there is no feasible set of input variables, and in the second case the second restriction need not be tested.
The level of abstraction chosen to describe CONDUCTOR, makes these problems implementation problems. Nonetheless, a complete specification of these problems would help to get a good implementation. The same is true for other implementation problems like the optimization of the instruction sequences in the kernel, or the error recovery used in the prototype.

16.3.3. Overspecification.

Where does the specification of a problem end and where does the implementation begin? This is probably one of the most difficult problems for a system analyst. Defining the problem is a matter of finding a good level of abstraction to describe the problem. For example, at the highest level of abstraction CONDUCTOR is defined by the module *Conductor* and its imported modules.

figure 16.2: The overall CONDUCTOR system.



Limiting the specification to 6 modules, it would only slightly contribute to the definition of CONDUCTOR. One must admit that also the level of abstraction chosen in the formal specification, presented in the appendices of this book, is arbitrary. In this specification the problems involved in optimising the kernel instructions, the hashing mechanism, the construction of the database, etc., are omitted. On the one hand, by defining these problems to be implementation problems this can be defended, on the other hand if one needs to implement code optimization or hashing mechanisms one could surely benefit from a thorough specification of these problems. A natural solution is to make an independent specification of these subproblems, emphasizing that these are implementation problems for the "main" specification.

16.3.4. Contradiction.

ASF does not offer the possibility to check the consistency of the specification. The equations may contain contradictions in terms of the logic of the specified sort, for example,

eq-result(abs-minus(e1,e2),abs-minus(e2,e1)) = *true* is obviously wrong. It specifies that the minus operator in index expressions is symmetric. Using the equations in module *Ind-expr-abstr-syntax* it can be derived from this equation that an index value is equal to its opposite

```
eq-result(abs-const(c),abs-const(neg(c))) = true
```

which is of course not what we had in mind. Even worse, if we added two equations to module *Ind-expr-abstr-syntax*

```
eq-result(abs-minus(e1,e2),abs-minus(e2,e1)) = true
eq-result(abs-minus(e1,e2),abs-minus(e2,e1)) = false
```

the contradiction in these equations is not detected by **ASF**. We actually specified that the constant functions *true* and *false* are identical. The problem is that, by giving meaningful names to functions, the system analyst assumes that the specified mathematical constructs have the suggested semantics. The examples above make it clear that **ASF** does not guarantee that the suggested semantics of a specification is without contradictions.
The examples also make it clear that modifications of the semantics of sorts, outside the module where they are declared should be avoided. These modifications make it, in a large specification, almost impossible to verify if the suggested and mathematical semantics coincide. A book-keeping mechanism that warns the user of ASF if the semantics of a sorts is redefined outside the module where it is declared should be of great help to a system analyst.

16.3.5. Ambiguity.

From the mathematical point of view, a specification always has a meaning. This meaning is certainly not the intended one if the specification contains contradictions. The discussion in 16.3.4. showed that in practice contradictions still exist in the suggested semantics, and thus we still have to face an ambiguous specification. This implies that we have to face ambiguities due to this difference between mathematical and intended meaning.

16.3.6. Forward reference.

In the formal specification language **ASF** forward reference is impossible. A module can only import functions and sorts of previously defined modules.

16.3.7. Wishful thinking

Is wishful thinking impossible if one uses ASF? Of course not. By remaining silent over the exact specification of a particular subproblem a delicate problem may remain unspecified and the system analyst may hope that someone might solve the problem. In CONDUCTOR an example of wishful thinking is the database. The database is left completely unspecified and the author of this book simply hopes and believes that a database in combination with background queries can be implemented.

16.4. SPECIFICATION OF ERRORS

At many points in the formal specification we had to use conditional equations. That is, we had to impose conditional restrictions on the sorts used as arguments in formal functions. An example is the specification of a constant range in module *Constant-ranges*. A constant range consists of two indices under the restriction that the first argument is equal to or less than the second argument. Legal examples of constant ranges are

```
   [ 1  to  5 ]     or    [ 27  to  217 ]
```

In the formal specification a range is represented by the sort *CONST-RANGE* through the function *c-range*.

```
   c-range:          INDEX # INDEX -> CONST-RANGE
```

To specify a well-formed constant range the functions *wf-c-range* and *error-c-range* are introduced

```
   wf-c-range:       INDEX # INDEX -> CONST-RANGE
   error-c-range:                  -> CONST-RANGE
```

These functions make it possible to specify a well-formed constant range consisting of two indices *ind1* and *ind2* as given in equation *[1]*.

[1] c-range(ind1,ind2) = wf-c-range(ind1,ind2)

when ge(ind2,ind1) = true

A constant range is specified to be illegal when the greater equal condition *ge(ind2,ind1)* does not hold.

[2] c-range(ind1,ind2) = error-c-range

when ge(ind2,ind1) = false

In this specification we reduced all combinations of indices to a set of correct constant ranges and one error case. The other functions in module *Constant-ranges*, such as *length* and *eq-range*, were all specified in terms of well-formed constant ranges (the function *wf-c-range*). For example the equation *[3]* specifies the length of a constant range.

[3] length(wf-c-range(ind1,ind2)) = add(1,sub(ind2,ind1))

The error case is left unspecified. Taking into account all error cases would considerably increase the length of the specification. For example, all functions specified for the sort *INDEX* would need extra equations for the constant function *error-index*. Probably the best solution is to incorporate the error propagation in **ASF** in such a way that the error cases due to conditional equations can only be seen in the module where such errors occur. At all points, in the specification of CONDUCTOR, where we used conditional equations, we made a comment that the error case is left unspecified. If a conditionally specified sort is imported by an other module one may assume that the well-formed sort is imported. Thus we avoided specifying lengthy error cases.

16.5. KEEPING THE PROTOTYPE CONSISTENT WITH THE SPECIFICATION

It is important to develop a prototype to check the suggested semantics of the specification. Of course this introduces the problem of keeping

the specification and the prototype consistent. One of the problems in this respect is that the exported sorts and functions should indeed form the interface of the implemented modules of the prototype.
The implemented prototype is by-hand kept consistent with the specification. This approach naturally introduces errors. For example, function names in the formal specification may differ from the names of their counterparts in the implementation. The only way to avoid this would be, that a specification is automatically translated into an implementation. This approach is taken by Arsac [1979] and Balzer [1981]. An other way to check the consistency of specification is the use of term-rewriting systems, as implemented by Hendriks [1988]. In such a system the equations in an ASF specification are used as rewriting rules. Given a term, the generated prototype for a specification tries to find the smallest possible representation, using the rewriting rules. This approach up till now only works for small examples.
To automatically produce an implementation from a specification of the size of the specification of CONDUCTOR still seems to be a dream. A small step in making this dream come true is the following. In many specifications one will need modules for structures such as booleans, sequences, and tables. There seems to be no need to specify these modules over and over again, and an efficient implementation of these modules can be made and stored in a module library. These modules can be used as building blocks for other specifications, that are also using these structures. And the efficient implementations of these modules should be linked automatically in the implementation of these specifications.

16.6. EQUALITY OF CONSTANT FUNCTIONS

At many places in the specifications we defined a function **eq** for equality of specified sorts. If constant functions were defined for these sorts, we had to specify that all these functions were unequal. This was boring work, and the result can almost be called noise. If we wanted constant functions to be equal we would have specified only one constant function. Therefore it seems worthwhile to incorporate an equal function in **ASF**, in order to avoid lengthy specifications of the inequality of constant functions. For instance, the specification of equality of three terms *t1*, *t2* and *t3* in the current version of **ASF** reads

```
eq: SORT # SORT -> BOOLEAN
```

with equation

```
eq(t1,t2) = true
eq(t1,t3) = false
eq(t2,t3) = false
```

In a new version this should be replace by one equation

```
t1 = t2
```

in combination with a built in function *eq*, that is specified to return the value *true* if two terms are equivalent in terms of the equational logic in a module, and otherwise returns the value *false*.

16.7. ADVICE FOR USERS OF ASF

The use of **ASF** by now means solves all problems of system analysts. If, however, the system analyst obeys a few simple rules, **ASF** opens the possibility to thoroughly define and modularize his problem before he starts implementing it. These rules are:

- keep the number of sorts specified in each module small,
- keep the number of imported modules small.

Furthermore, it is important to develop a prototype to check the suggested semantics of a specification. Of course, this introduces the problem of keeping the specification and the prototype consistent.
Also, realize that a specification should be an abstraction of a problem. There is no such thing as the one and only correct abstraction. Important is to keep the specification readable by abstracting from subproblems (implementation problems). A useful approach is to

- make the specification of subproblems completely independent of the specification of the 'main' problem.

Finally, finding better ways to deal with error cases in the specifications, and incorporating a function for equality of terms, will surely improve the specification formalism **ASF** used in this book.

17. CURRENT STATUS OF CONDUCTOR AND FUTURE DEVELOPMENTS

CONDUCTOR is not a commercial product. The current prototype, that runs under both the operating systems UNIX and MS-DOS, can compile and run the statistical techniques discussed in this book. It still has to be tested on a larger set of examples. These tests probably will reveal that the efficiency of some of the implemented modules can be improved. Furthermore, a collection of statistical functions, comparable to the functions in languages like S and IML, should be added in order to make it possible to implement large sets of statistical techniques. Also the user language has to be improved if one wants to make CONDUCTOR competitive with modern statistical software packages. Important in this respect are virtual screens and pull-down menus. The virtual screens and the pull-down menus were first introduced in SMALLTALK (see Goldberg [1983-1984]), and have strongly influenced the user interface of modern operating systems and software packages. Though adding such features is merely a matter of time, it is necessary if one wants to compete with existing statistical software packages.
Besides these improvements, other more fundamental concepts can be added to CONDUCTOR. One can think of:

- Making it possible to create *statistical expert systems* in CONDUCTOR. An exception, in software generated by CONDUCTOR, signals when the execution of a statistical technique must be interrupted. In the current version of CONDUCTOR, the generated software reacts on a single exception. For example, the software generated in CONDUCTOR may

detect that the data used in a statistical technique is constructed using different measurement instruments. The execution of the statistical technique is interrupted, and the software looks for an appropriate exception handler.
If the exception handling mechanism is given a 'memory', a combination of exceptions can be examined, using inference rules given by the technical statistician. Thus, different combinations of exceptions may invoke different exception handlers. The created statistical expert system, would be able to give a user advice in more complicated cases in statistical analyses. For a discussion on the slow emergence of expert systems in statistics see Gale et al. [1986].

- Adding *a syntax directed editor*. A statistical technique written in the statistical language shows close resemblance with the notation used in statistical textbooks. However, partitioned matrices can not be entered the way they are written in these books. The reason is that the input of a statistical technique is line oriented rather than screen oriented. With the use of syntax directed editors that know the lay out of a partitioned matrix, partitioned matrices can be entered the way they are written in textbooks. A second advantage of syntax directed editors is that the abundant typing of keywords can be avoided. Syntax directed editors are for example used in the programming language B[1] (see Geurts et al. [1985] and Meertens [1981]).

- Generating code for *a parallel processor*. In a statistical language the techniques are described in a matrix langauge. Some of the matrix functions can be executed much faster by parallel than by sequential processors. Parallel algorithms for matrix functions are discussed in O'Leary and Stewart [1985]. Instead of generating instructions for the kernel, as in the prototype of CONDUCTOR, code could be generated for a parallel processor. This would, of course, dramatically improve the performance of the resulting software.

Besides these 'dreams', also other problems remain open. The most

[1] The current version of this language is called ABC.

important 'open ends' in the CONDUCTOR project are:

- *Contradicting input restrictions*. For each statistical technique, a sequence of input restrictions on the value of the input index variables is generated. These input restrictions may be contradicting. For example, the input restrictions

  $$n - 1 >= 0 \quad \text{and} \quad - n - 1 >= 0$$

  both must hold. In this case no feasible input exists. This simple case is detected by the prototype, however more complex cases should be considered.

- CONDUCTOR will detect many exceptions during statistical analysis. Probably there will occur *combinations of exceptions for which statistics does not offer any solution*. Statistical theory focusses on single exceptions to a general model. In applied statistics many exceptions occur simultaneously. For some of the combinations the statistical theory has no answers. We hope that the statistical software generated by CONDUCTOR will stimulate statisticians to find answers for such questions.

The current version of CONDUCTOR is already a useful piece of software. It can generate efficient software for small statistical applications. By adding new features to it we also hope to improve it in the near future.

References.

AHO, V.A., SETHI, R. & ULLMAN, J.D. (1986), Compilers Principles Techniques and Tools, Reading, MA: Addison-Wesley.

AHO, V.A. & ULLMAN, J.D. (1977), Principles of Compiler Design, Reading, MA: Addison-Wesley.

AMMANN, U. (1977), On Code Generation in a Pascal Compiler, Software, Practice and Experience, 391-423.

ARCHER, J.E. (1984), User Recovery and Reversal in Interactive Systems, ACM Transactions on Programming Languages and Systems, 1-19.

ARSAC, J.J. (1979), Syntactic source-to-source Program Manipulation, Communications of the ACM, 43-54.

BALZER, R.M. (1981), Transformational Implementation, USC/ISI, report RR-79-79.

BECKER, R.A. & CHAMBERS, J.M. (1984a), S: An Interactive Environment for Data Analysis and Graphics, Belmont, CA: Wadsworth.

BECKER, R.A. & CHAMBERS, J.M. (1984b), Design of the S system for Data Analysis, Communications of the ACM, 486-495.

BERGSTRA, J.A., HEERING, J. & KLINT, P. (1985), Algebraic Definition of a Simple Programming Language, Amsterdam: Centre for Mathematics and Computer Science, Report CS-R8504.

BERGSTRA, J.A., HEERING, J. & KLINT, P.(1987), ASF: An Algebraic Specification Formalism, Amsterdam: Centre for Mathematics and Computer Science, Report CS-R8705.

BERGSTRA, J.A. & TUCKER, J.V. (1982), Algebraic Specification of Computable and Semi-computable Data Structures, Theoretical Computer Science, 137-181.

BMDP (1985), Statistical Software Manual, Berkeley, CA: University of California Press.

BOX, G.E.P. (1969), The Challenge of Statistical Computation, in MILTON, R.C. & NELDER, J.A. (editors), Statistical Computation, New York: Academic Press.

CHENG, T.T., LOCK, E.D. & PRYWES, N.S (1984), Use of Very High Level Languages and Program Generation by Management Professionals, IEEE Transactions on Software Engineering, 552-563.

CERI, S. & PELAGATTI, G. (1985), Distributed Databases Principles & Systems, New York: Mcgraw-Hill.

DATE, C. J. (1977), An Introduction to Database Systems, Reading, MA: Addison & Wesley.

DAVID, M.H. (1985), The Language of Panel Data and Lacunae in Communication About Panel Data, Madison, WI: Draft for Discussion, Institute for Research on Poverty.

DAVIDSON, J.W. & FRASER, C.W. (1984), Code Selection through Object Code Optimization, Transaction on Programming Languages and Systems, 505-526.

DEREMER, F. (1969), Practical Translators for LR(k) Languages, Cambridge, MA.: Ph. D. Thesis, M.I.T.

DEREMER, F. (1971), Simple LR(k) grammars, Communications of the ACM, 453-460.

DRUD, A. (1983), A Survey of Model Representation and Simulation Algorithms in Some Existing Modelling Systems, Journal of Economic Dynamics and Control, 5-35.

EFE, K. (1987), A Proposed Solution to the Problem of Levels in Error-message Generation, Communications of the ACM, 948-955,

EFRON, B. & GONG, G. (1983), A Leisurely look at the Bootstrap, the Jackknife and Cross-validation, The American Statistician, 36-48.

FOSTER, J.M.(1968), A Syntax Improving Device, Computer Journal, 31-34.

FRANCIS, I. (1981), A Comparative Review of Statistical Software, New York: North Holland.

GALE, W.A.(editor) (1986), Artificial Intelligence & Statistics, Reading, MA: Addison-Wesley.

GAUDEL, M.C. (1984), A first introduction to PLUSS, Université de Paris-Sud, Orsay.

GEURTS, L., MEERTENS L. & PEMBERTON S. (1985), The B Programmers's Handbook, Amsterdam: Centre for Mathematics and Computer Science.

GOLDBERG, A. & ROBSON, D. (1983-1984), SMALLTALK-80, 4 vols, Reading, MA: Addison-Wesley.

GOLDBERGER, A.S. (1964), Econometric Theory, New York: Wiley.

GOMAA, H. & SCOTT, D.B.H. (1981), Prototyping as a Tool in the Specification of User Requirements, Proc. 5th Conf. on Software Engineering, IEEE, 333-342.

GOODENOUGH, J.B. (1975), Exception Handling, Issues and Proposed Notation, Communications of the ACM, 683-696.

GRILICHES, Z. (1984), Data Problems in Econometrics, Cambridge, MA: Discussion Paper Series, Harvard Institute of Economic Research.

GRISWOLD, R. (1981), A History of the SNOBOL Programming Languages, in WEXELBLAT, R. (editor), History of Programming Languages, New York: Academic Press.

HENDRIKS, P. (1988), ASF System User's Guide, Amsterdam: Centre for Mathematics and Computer Science, Report forthcoming.

HOROWITZ, E. & MUSON J.B.(1984), An Expansive View of Reusable Software, IEEE Transactions on Software Engineering, 477-487.

IVERSON, K. (1962), A Programming Language, New York: Wiley.

IVIE, E.L. (1977), The Programmer's Workbench, Communications of the ACM, 746-753.

JACKSON, M.A. (1975), Principles of Programming Design, New York: Academic Press.

JACKSON, M.A. (1983), System Development, Englewood Cliffs, NJ: Prentice-Hall.

JOHNSON, S.C. (1975), YACC-Yet Another Compiler Compiler, Murray Hill: Bell Laboratories, CSTR 32.

JÖRESKOG, K.G. & SÖRBOM, D. (1981), LISREL V, Analysis of Linear Relationships by Maximum Likelihood and Least Squares Methods, Uppsala: Department of Statistics, University of Uppsala.

JUDGE, G.G., GRIFFITHS, W.E., CARTER HILL, R., LEE T. (1980), The Theory and Practice of Econometrics, New York: Wiley.

IVIE, E.L. (1977), The Programmer Workbench, Communications ACM, 746-753.

KENNEDY W.J. & GENTLE J.E. (1980), Statistical Computing, New York: Marcel Dekker.

KERNIGHAN, B.W. (1984), The UNIX System and Software Reusability, IEEE Transactions on Software Engineering, 513-518.

KERNIGHAN, B. W. & PIKE R. (1984), The UNIX Programming Environment, Englewood Cliffs: Prentice-Hall.

KING, (1976), Symbolic Evaluation, Communications of the ACM, 385-394.

KLAEREN, H.A. (1983), Algebraische Spezifikationen: Eine Einfuhrung, Berlin: Springer Verlag.

KNUTH, D.E. (1965), On the Translation of Languages from Left to Right, Information and Control, 607-639.

KNUTH, D.E. (1971), Top-Down Syntax Analysis, Acta Informatica, 79-110.

KOSTER, C.H.A. (1976), Two Level Grammar, in BAUER, F.L. & EIKEL, J. (editors), Compiler Construction an Advanced Course, New York: Springer Verlag, 2nd edition.

KROENKE, D. (1983), Database Processing: Fundamentals, Design, Implementation, Chicago: SRA, 2nd edition.

McKEEMAN, W.M. (1974), Compiler Construction, in BAUER, F.L & EICKEL, J., Compiler Construction: an Advanced Course, Berlin: Springer.

LEWIS, P.M., ROSENKRANTZ, D.J, & STEARNS R.E. (1974), Attributed Translations, Journal of Computers and System Science, 297-307.

LITTLE, R.J.A. (1982), Models for Non-Response in Sample Surveys, Journal of American Statistical Association, 237-250.

LIENTZ, B.P., & SWANSON, E.B. (1980), Software Maintenance Management, Reading, MA: Addison-Wesley.

LOECKX, J. (1984), Algorithmic Specifications: a Constructive Method for Abstract Data Types, University of Saarland, Report A84/03.

LOVELL, M. (1983), Data Mining, Review of Economics and Statistics, 1-11.

MACSYMA (1977), Reference Manual, Cambridge, MA: Laboratory of Computer Science, MIT.

MADDALA, G.S. (1977), Econometrics, New York: McGraw-Hill.

MALINVAUD, E. (1980), Statistical Methods of Econometrics, New York: North-Holland.

MAYER, T. (1975), Selecting Economic Hypothesis by Goodness of Fit, The Economic Journal, 877-883.

McGETTRICK, A.D. (1980), The Definition of Programming Language, London: Cambridge University Press.

MEERTENS, L. (1981), Draft Proposal for the B programming language, Amsterdam: Mathematical Centre.

MESEGUER, J. & GOGUEN, J.A. (1982), An Initiality Primer, in NIVAT, M. & REYNOLDS, J. (editors), Formalizing Programming Concepts, Amsterdam: North-Holland.

MEYER, B. (1985), On the Use of Formalism in Specifications, IEEE, 6-26.

NEIGHBOURS, J.M. (1980), Software Construction Using Components, Irvine: Ph.D. dissertation, Dep. Inform. Comput. Science, University of California, Tech. Rep. 160.

NEIGHBOURS, J.M. (1984), The DRACO Approach to Constructing Software from Reusable Components, IEEE Transactions on Software Engineering, 564-574.

PAKIN, S. (1972), APL-360 Reference Manual, Chicago, SRA.

PARK, J.C.H., CHOE, K.M. & CHANG, C.H. (1985), A New Analysis of LALR formalism, ACM Transactions on Programming Languages and Systems, 159-175.

PARNAS, D.L., CLEMENTS, P.C., & WEISS, D.M. (1985), The Modular Structure of Complex Systems , IEEE Transactions on Software Engineering, 259-266.

PEETERS, J.F. (1985), Assembly Language Programming VAX-11, Reston, VA: Reston Publishing Company.

PINDYCK, R.S., & RUBINFELD, D.L. (1981), Econometric Models and Economic Forecasts, New York: McGraw-Hill.

PRATT, W. (1984), Programming Languages, Design and Implementation, Englewood Cliffs: Prentice Hall, 2nd edition.

PRYWES, N.S. (1979), Automatic Generation of Computer Programs, in RUBINOFF, M. & YOVITS, Advances in Computers, vol. 16 New York: Academic.

PRYWES, N.S., PNEULI A. & SHASTRY, S. (1979), Use of a Nonprocedural Specification Language and Associated Program Generator in Software Development, ACM Transactions on Programming Languages and Systems, 196-217.

REDUCE-2 (1973), User's Manual, Salt Lake City: University of Utah, Report UCP-19.

RODLER, K. (1985), Evaluierung Ökonometrische Programsysteme, Vienna: Institut Fur Ökonometrie und Operations Research, Technical Report 26.

RUBIN, D.B. (1976), Inference and Missing Data, Biometrika, 581-592.

SAS (1982), User's Guide: Basics, Cary: SAS Institute Inc.

SAS (1982), User's Guide: Statistics, Cary: SAS Institute Inc.

SAS (1985), User's Guide: IML, Cary: SAS Institute Inc.

SHERIDAN, P.B. (1959), The Arithmetic Translator-Compiler of the IBM Fortran Automatic Coding System, Communications of the ACM, 9-21.

SPSS (1986), User's Guide, New York: McGraw-Hill.

SRBA, F. (1985), A Survey on Econometric Software, internal report, ESCR Centre in Economic Computing.

TANENBAUM, A.S., VAN STAVEREN, H. & STEVENSON, J.W (1983), A Practical Tool Kit for Making Portable Compilers, Communications of the ACM, 654-660.

TANENBAUM, A.S. (1976), Structured Computer Organization, Englewood Cliffs: Prentice Hall.

TSP, Time Series Processor User's Manual (1980), Ontario: Computing Centre, University of Western Ontario.

ULLMAN, J.D. (1985), Principles of Database Systems, London: Pitman, 2nd edition.

USDoD (1983), Reference Manual for the ADA Programming Language, Washington D.C.: U.S. Department of Defense, ANSI/MIL - STD-1815A-1983.

VAN NES, F. & TEN CATE A. (1987), Software voor econometrisch onderzoek met behulp van de personal computer, Section Statisticial Software of the VVS, internal report.

WIJNGAARDEN, A. van (1965), Orthogonal Design and Description of a Formal Language, Amsterdam: Centre for Mathematics and Computer Science, Report MR 76.

WIJNGAARDEN, A. van (editor) (1973), Almost the Revised Report on the Algorithmic Language ALGOL 68, working paper W.G. 2.1.

WIENER, R. & SINCOVEC, R. (1984), Software Engineering with MODULA-2 and ADA , New York: Wiley.

WIENER, R. & SINCOVEC, R. (1983), Programming in ADA, New York: Wiley.

WIRSING, M. (1983), A Specification Language, Munich: University of Munich, Dissertation.

WIRTH, N. (1971), The Design of the PASCAL Compiler, Software Practice & Experience, 309-333.

WIRTH, N. (1983), Programming in MODULA-2, New York: Springer Verlag, 2nd edition.

WOODWARD, W.A., ELLIOT A.C. & GRAY, H.L. (1985), Directory of Statistical Microcomputer Software, New York: Marcel Dekker.

Index

APPENDIX A: THE CONCRETE SYNTAX

This appendix contains the concrete syntax of the statistical language, the user language, the language that can be used to create external exception handlers. In the syntax rules the following notational conventions are employed. (1) All words written in lower-case letters are non-terminals. (2) All words and symbols between single quotes (' and ') are terminals. (3) All words written in upper-case letters not between quotes are terminals.

A. THE OVERALL CONDUCTOR SYSTEM

```
session                 --> session  sub_session ¦
                            sub_session

subsession              --> 'STATISTICIAN' statistical_program ¦
                            'USER'         user_session ¦
                            'EXTERNAL'     ext_handler_program
```

B. THE STATISTICAL LANGUAGE

1. General Structure.

```
statistical_program     --> statistical_program section ¦
                            section

section                 --> name_section ¦
                            input_output_section ¦
                            implementation_section ¦
                            test_section ¦
                            exception_handler_section
```

2. Name section.

```
name_section            --> 'NAME'  IDENTIFIER
```

3. Import/export section.

```
input_output_section    --> INTERFACE input_declaration
                            output_declaration

input_declaration       --> 'INPUT' declarations

output_declaration      --> 'OUTPUT' declarations
```

4. Implementation section.

```
implementation_section  --> local_declarations
                            'EQUATIONS' statements

statements              --> statements ';' statement¦
                            statement

statement               --> assignment ¦
                            index_assignment ¦
                            message ¦
                            compound_stat ¦
                            for_statement ¦
                            SPECIAL_PROCEDURE

assignment              --> variable ':='  expression

compound_stat           --> '{' statements '}'

for_statement           --> 'FOR' index_variable ':=' index_expr
                             'TO' index_expr 'DO' statement ¦

                          statement index_variable ':='
                              index_expr ',...,' index_expr

index_assignment        --> INDEX_VARIABLE ':=' index_expr
```

5.Test section.

```
test_section            --> 'TEST' local_declarations
                            'EQUATIONS' statements  raises

raises                  --> raises raise ¦
                            raise

raise                   --> 'RAISE' exception_decl 'WHEN' expression

exception_decl          --> IDENTIFIER ¦
                              EXCEPTION_FLAG
```

6. Handler section.

```
exception_handler_section --> 'HANDLERS' handlers

handlers                  -->  handlers handler ¦
                               handler

handler                   --> 'WHEN' ':' EXCEPTION_FLAG
                               local_declarations
                              'EQUATIONS' statements
                            stop_or_continue

stop_or_continue            --> STOP ¦
                                 /* default continue */
```

7. Messages.

```
message                 --> 'MESSAGE' ':' STRING ¦
                             /* empty */
```

8. Expression.

```
expression            --> expression  OP  term ¦
                          term

term                  --> '(' expression ')' ¦
                          function_call ¦
                          factor ¦
                          PREFIC_UNARY_OP term ¦

factor                --> factor POSTFIX_UNARY_OP ¦
                          variable ¦
                          SCALAR_VALUE ¦
                          INDEX_VALUE

variable              --> VARIABLE_NAME ¦
                          VARIABLE_NAME offset

offset                --> '[' subranges ']' ¦
                          '[' mat_indices ']'

subranges             --> subranges ',' subrange ¦
                          subrange

subrange              --> index_expr 'TO' index_expr

mat-indices           --> mat-indices ',' mat-index ¦
                          mat-index

mat-index             --> index_expr
```

9. Function calls.

```
function_call         --> FUNCTION_NAME '(' argument_list ')'

argument_list         --> argument_list ',' expression ¦
                          expression
```

10. Declarations.

```
local_declaration     --> 'VARIABLES' declarations ¦
                           /* empty */

declarations          --> declarations ';' declaration ¦
                          declaration

declaration           --> var_type id_list message

id_list               --> id_list ',' IDENTIFIER ¦
                          IDENTIFIER

var_type              --> 'INDEX' ¦ 'SCALAR' ¦ BOOL' ¦
                          mat_type ¦ vec-type

mat_type              --> 'MATRIX' '[' ranges ']'

vec_type              --> 'VECTOR'  range
ranges                --> ranges ',' range ¦
                          range

range                 --> index_expr 'TO' index_expr
```

11. Index expressions.

```
index_expr            --> '(' index_expr ')' ¦
                           index_expr '+' index_expr ¦
                           index_expr '-' index_expr ¦
                           index_expr '*' index_expr ¦
                           INDEX_VARIABLE ¦
                           INDEX_VALUE
```

C. USER LANGUAGE

```
user_session           --> user_session user_command ¦
                           user_command

user_command           --> 'RETRIEVE'   user_variables ¦
                           'DISPLAY'    user_variables ¦
                           'DELETE'     user_variables ¦
                           'INITIALIZE' user_variables ¦
                           'FORMAT' INDEX_VALUE INDEX_VALUE ¦
                           'SET'        user_variables ¦
                           'SAMPLE'     const_ranges    ¦
                            TECHNIQUE_ID             ¦
                            user_declaration

user_variables        --> user_variables  ',' user_variable ¦
                          user_variable

user_variable         --> MATRIX_ID ¦
                          SCALAR_ID ¦
                          INDEX_ID  ¦
                          SERIES_ID ¦
                          BOOL_ID
user_declaration      --> user_type id_list

user_type             --> 'BOOL'     ¦
                          'INDEX'    ¦
                          'SCALAR'   ¦
                          'VECTOR' '[' const_ranges ']'¦
                          'MATRIX' '[' const_ranges ']'¦
                          'SERIES' '{' const_ranges '}`

const_ranges          --> const_ranges ',' const_range ¦
                          const_range

const_range           --> INDEX_VALUE 'TO' INDEX_VALUE
```

D. EXTERNAL HANDLERS

```
ext_handler_program   --> exception_handler_section
```

APPENDIX B. ASF

A (many-sorted) signature is a set of declarations of sorts and functions over these sorts. A signature defines a language of strongly typed terms (expressions). A basic ASF module consists of a signature, a set of variable declarations, and a set of positive conditional equations in the language defined by the signature and the variable declarations. ASF modules may be parameterized. Parameter binding, and importing modules in other ones, are the two ways in which modules can be combined in ASF.
ASF specifiactions are sequences of modules. A module can be normalized in the context of a specification to which it belongs by eliminating all imports and binding as many parameters as possible. Normalization is a textual operation. The semantics of a module is the initial algebra of its normal form, provided the latter does noet have any remaining unbound parameters.

1. SYNTAX OF ASF

In this section we give a context-free grammar for ASF. The following notational abbreviations are used in this definition:
- [<N>] denotes an optional occurrence of <N>.
- <N>* and <N>+ denote, respectively, zero or more, and one or more occurences of <N>.
- { <N> t }* and { <N> t }+ denote, respectively, zero or more, and one or more occurences of <N> separated by terminal symbol t.

ASF has the following grammar:

```
<specification>      ::= <module>+.
<module>             ::= "module" <module-ident>
                            "begin"
                                [ <parameters> ]
                                [ <exports) ]
                                [ <imports> ]
                                [ <sorts> ]
                                [ <functions> ]
                                [ <variables> ]
                                [ <equations> ]
                            "end" <module-ident> .
<module-ident>       ::= <ident> .
<parameters>         ::= "parameters" { <parameter> ","}+ .
<parameter>          ::= <parameter-ident>
                             "begin"
                                 [ <sorts> ]
                                 [ <functions> ]
                             "end" <parameter-ident> .
<parameter-ident>    ::= <ident> .
<exports>            ::= "exports"
                            "begin"
                                 [ <sorts> ]
                                 [ <functions> ]
                             "end" .
<imports>            ::= "imports" { <module-expression> ","}+ .
<module-expression>  ::= <module-ident> [ "{" <modifier> "}" ] .
<modifier>           ::= <renamed> [ <bound> ] | <bound> [ <renamed> ] .
<renamed>            ::= "renamed" "by" <renamings> .
```

```
<renamings>          ::= "[" { <renaming> "," }+ "]" .
<renaming>           ::= <sort> "->" <sort> /
                         <fun-or-operator-ident> "->"
                         <fun-or-operator-ident> .

<fun-or-operator-ident>
                     ::= <fun-ident> | "_" <operator> "_" |
                         <operator> "_" .
<bound>              ::= ( <parameter-ident> "bound" "by" <renamings>
                           "to" <module-ident> )+ .
<sorts>              ::= "sorts" <sort-list> .
<sort-list>          ::= { <sort> "," }+ .
<sort>               ::= <ident> .
<functions>          ::= "functions" <function>+ .
<function>           ::= <fun-ident> ":" <input-type> "→" <output-type> |
                         <operator> "_" ":" <sort> "→" <output-type> |
                         "_" <operator> "_" ":" <sort> "#" <sort> "->"
                         <output-type> .
<fun-ident>          ::= <ident> .
<input-type>         ::= [ <product> ] .
<output-type>        ::= <product> .
<product>            ::= { <sort> "#" }+ .
<variables)          ::= "variables" <variable-list> .
<variable-list>      ::= ( <var-ident-list> ":" "->" <sort> )+ .
<var-ident-list>     ::= { <var-ident> "," }+ .
<var-ident>          ::= <ident> .
<equations>          ::= "equations" <cond-equation>+ .
<cond-equation>      ::= <tag> <equation-list> <implies> <equation> |
                         <tag> <equation> ["when"  <equation-list> ] .
<tag>                ::= "[" <ident> "]" .
<equation-list>      ::= { <equation> "," }+ .
<equation>           ::= <term> "=" <term> .
<term>               ::= [ <term> <operator> ] <primary> |
                         <operator> <primary> .
<primary>            ::= <fun-ident> ["(" <term-list> ")" ] |
                         <var-ident> | <tuple> |
                         "(" <term> ")" .
<term-list>          ::= { <term> "," }+ .
<tuple>              ::= "<" <term> "," <term-list> ">" .
```

2 LEXICAL SYNTAX

Layout or comment may separate the following lexical notions of ASF: <ident> <operator> and <implies>. Layout has no significance other than separating consecutive lexical tokens that would otherwise not be distinguished. Layout may never occur embedded in a lexical token. In cases of ambiguity, the longest lexical token is preferred. The lexical conventions of ASF are summarized below.

- Layout characters are space, horizontal tabulation, carriage return, line feed and form feed.

- Comments follow a layout character and begin with two hyphens and end with either and end of line (i.e., carriage return or line feed) or another pair of hyphens.

- Identifiers (i.e., <ident>) consist of a non-empty sequence of letters, digits or single quote characters, possibly with embedded hyphens. This is expressed by the following rules:

```
<id-char>     ::= <letter> | <digit> | "'" .
<ident>       ::= <id-char> [ ( <id-char> | "-" )* <id-char> ] .
```

  For example, x, maxl, 2-way, x' ', double--hyphen, Very-Long-Identifier and 6 are legal identifiers, but -a, - and a- are illegal.

- The following identifiers are reserved as keywords and cannot be used as an identifier:

begin	end	functions	parameters	to
bound	equations	imports	renamed	variables
by	exports	module	sorts	when

  For technical reasons we also forbid the names hidden and export as <parameter-ident> (see section 2.6).

- Operators (i.e., <operator>) are denoted by either a sequence of one or more operator symbols or by an identifier surrounded by dots:

```
<op-symbol> ::= "!" | "@" | "$" | "%" | "^" | "&" | "+" | "-" | "*" |
                ";" ¦ "?" ¦ "~" ¦ "/" ¦ "|" ¦ "\" .
<operator>     ::= <op-symbol>+ | "." <ident> "." .
```

  The operators: +, -, &&, .push. and !@%%? are legal.
- The token <implies> consists of two or more consecutive = characters followed by either the character > or a new line:

```
<implies>      :: = "==" "="* ( ">" | "\n" ) .
```

3. SIGNATURES, VARIABLES AND EQUATIONS

3.1. Signatures

Signatures are sets of declarations of sorts and functions over these sorts. Functions without arguments will also be called constants. See, for instance Klaeren [1983] for a description of signatures. The algebra of signatures and normalization of signature expressions are discussed in Bergstra et al [1986]. The notion of signature used in ASF differs in three respects from the usual one:
- Functions, as defined in an ASF signature, may have various syntactical forms.
- Functions may have tuples as output type.
- Functions may be overloaded.

A signature combined with a set of variables and a set of (positive conditional) equations forms a basic ASF module. Variables are typed with a sort in the signature.
In combination with a set of typed variables, a signature allows the construction of well-typed terms, i.e., terms obtained by type-wise correct composition of functions and variables. Due to the possibility of overloading, typing of terms is slightly more complicated than in the traditional case.
Unconditional equations have the form:

[tag] $t_l = t_r$

where t_l and t_r are well-typed terms of the same type. Conditional equations can have two (equivalent) forms:

[tag] $t_{r1} = t_{r1}, \ldots, t_{ln}=t_{rn}$ ==> $t_l=t_r$

or

[tag] $t_l=t_r$ when $t_{r1}, \ldots t_{ln}=t_{rn}$

Variables in equations are implicitly universally quantified. Sound and complete rules of deduction for many-sorted conditional equations are given in Goguen and Meseguer [1982].

3.2. Functions and operators

Depending on the way they are declared, functions are either
- ordinary prefix functions; or
- monadic prefix operators; or
- dyadic infix operators.

Declarations of prefix functions have the form:

```
<ident> ":" <input-type> "->" <output-type>
```

For instance,

```
f : S1 # S2 -> S3
```

defines a prefix function f with argument sorts S1, S2 and output sort S3.
Prefix and infix operators may be used instead of, respectively, monadic and dyadic functions. The corresponding operator declarations have the following form:

```
    <operator> "_" ":" <sort>             "->" <output-type>
"_" <operator> "_" ":" <sort> "#" <sort> "->" <output-type>
```

The position of operands of operators is indicated by underline characters (_). For instance,

```
_ + _ : S1 # S2 -> S3
```

defines the infix operator + with argument sorts S1, S2 and output sort S3, while

```
- _   : S1      -> S1
```

defines the monadic prefix operator - with both argument and output of sort S1. Infix and prefix operators are only a notational device and can always be replaced by ordinary functions. Dyadic operators are left-associative and have a lower priority than monadic ones.

APPENDIX C. ASF SPECIFICATION OF BASIC STRUCTURES

In this appendix the the simple data types, sequences and tables, are specified. It is assumed that each sort and function in the modules corresponds to its "natural" meaning. A full specification of the modules, including equations, can be found in Bergstra et al. [1986].

1. SIMPLE DATA TYPES.

The simple data types are booleans, indices and scalars.

1.1. BOOLEANS.

1.1.a. Global description.

In the module *Booleans* the usual boolean operators are specified, such as, and, or, not, etc. Also the sort *BOOL* can be used in an "if-then-else" construct.

1.1.b. Specification.

```
module Booleans
begin
   exports
      begin
        sorts          BOOL
        functions
           true   :                     -> BOOL
           false  :                     -> BOOL
           and    : BOOL # BOOL         -> BOOL
           or     : BOOL # BOOL         -> BOOL
           not    : BOOL                -> BOOL
           eq     : BOOL # BOOL         -> BOOL
           if     : BOOL # * # *        -> *
      end

end Booleans
```

1.2. SCALARS.

1.2.a. Global description

The module *Scalars* specifies reals. For scalars, the functions add (*add*), subtract (*sub*), multiply (*mul*), divide (*div*) and equal (*eq*) are specified. Even though all these functions are on the export list of the module *Scalars*, only the function *eq* is imported by other modules in the formal specification.

1.2.b Specification.

```
module Scalars
begin
     exports
       begin
          sorts SCALAR
          functions
               add:       SCALAR # SCALAR     ->   SCALAR
               sub:       SCALAR # SCALAR     ->   SCALAR
               mul:       SCALAR # SCALAR     ->   SCALAR
               div:       SCALAR # SCALAR     ->   SCALAR
               eq:        SCALAR # SCALAR     ->   BOOL
       end

     imports Booleans

end Scalars
```

1.3. INDICES.

1.3.a. Global description.

Indices are specified in module *Integers*. The constant functions *0* and *1* are specified to be indices, and also every index that can be constructed using the functions increment (*increm*), round (*round*), add (*add*), subtract(sub), multiply(mul) and negate(neg). Furthermore, the boolean relations less than or equal to (*le*) and equal to (*eq*) are specified for indices.

1.3.b. Specification.

```
module Indices
begin
     exports
       begin
          sorts INDEX
          functions
               0:                            -> INDEX
               1:                            -> INDEX
               increm:   INDEX               -> INDEX
               round:    SCALAR              -> INDEX
               add:      INDEX # INDEX       -> INDEX
               sub:      INDEX # INDEX       -> INDEX
               mul:      INDEX # INDEX       -> INDEX
               neg:      INDEX               -> INDEX
               ge:       INDEX # INDEX       -> BOOL
               le:       INDEX # INDEX       -> BOOL
               eq:       INDEX # INDEX       -> BOOL
       end

     imports Booleans, Scalars

end Indices
```

2. SEQUENCES.

2.a. Global description.

Sequences are linear lists of items; they are parameterized with the type of the item. The following functions are specified for sequences:

add-item:	add an item to a sequence
del-item:	delete an item from a sequence
conc:	concatenate two sequences
eq-seq:	check the equality of two sequences
disjunct:	test if two sequences are disjunct
n-of-items:	count the number of items in a sequence
first:	return the first item in a sequence
last:	return the last item(s) in a sequence
item_no:	return the specified item in a sequence

2.b. Specification.

```
module Sequences
begin
    parameters Items
      begin
        sorts       ITEM
        functions
            eq-item: ITEM # ITEM  -> BOOL
      end Items

    exports
      begin
        sorts       SEQ
        functions
          null      :                     -> SEQ
          add-item  : ITEM # SEQ          -> SEQ
          del-item  : SEQ                 -> SEQ
          del-item  : SEQ  # INDEX        -> SEQ
          conc      : SEQ  # SEQ          -> SEQ
          eq-seq    : SEQ  # SEQ          -> BOOL
          disjunct  : SEQ  # SEQ          -> BOOL
          n-of-items: SEQ                 -> INDEX
          first     : SEQ                 -> ITEM
          last      : SEQ                 -> ITEM
          last      : SEQ # INDEX         -> SEQ
          item-no   : SEQ # INDEX         -> ITEM
      end

    imports Booleans, Indices

end  Sequences
```

3. TABLES.

3.a. General description.

In several parts of the formal specification we need tables to store information. Therefore, a general module *Tables* is specified with parameters *Entries* and *Addresses*. In this module the following functions are specified:

null-table:	an empty table
insert:	insert an entry at a specified address in the table
lookup:	lookup an entry at a specified address and check if entry is found
delete:	delete an entry at the specified address in the table
eq-tab:	equality of a table
_ˆ_:	return an entry if found with lookup

3.b. Specification.

```
module Tables
begin
  parameters
    Entries
      begin
        sorts      ENTRY
         functions
          eq-entry:    ENTRY # ENTRY  -> BOOL
      end Entries,

    Addresses
      begin
        sorts      ADDRESS
         functions
            eq-addr: ADDRESS # ADDRESS  -> BOOL
      end Addresses

  exports
    begin
      sorts         TABLE
      functions
        null-table:                          -> TABLE
        insert:     ADDRESS # ENTRY # TABLE -> TABLE
        delete:     ADDRESS # TABLE         -> TABLE
        eq-tab:     TABLE   # TABLE         -> BOOL
        found:      ADDRESS # TABLE         -> BOOL
        _ˆ_:        ADDRESS # TABLE         -> ENTRY
        lookup:     ADDRESS # TABLE         -> (ENTRY # BOOL)
    end

  imports Booleans

end Tables
```

4.IDENTIFIERS AND STRINGS.

4.a Global description.

Strings are used in CONDUCTOR to communicate messages. Strings are a sequence of characters. Identifiers are a restricted set of strings.

4.b. Specification.

```
module Strings
begin
     exports
       begin
         sorts CHAR,STRING
         functions
            alpha:                       -> CHAR
            num:                         -> CHAR
            blank:                       -> CHAR
            other:                       -> CHAR
            string: CHAR                 -> STRING
            string: CHAR # STRING        -> STRING
            eq-str: STRING # STRING      -> BOOL
       end

     imports Booleans

end Strings


module Identifiers
begin
     exports
      begin
        sorts ID
        functions
           no-id:             -> ID
           an-id: STRING      -> ID
           greater-int:       -> ID
           -- identifier used for greater/equal function --
           -- for integer variables                       --

           eq-id: ID # ID     -> BOOL
      end

    imports Booleans, Strings

end Identifiers


module Id-sequences
begin
   imports Sequences
     {renamed by
       [SEQ        -> ID-SEQ,
        null       -> null-id-seq,
        eq-seq     -> eq-id-seq]
        Items bound by
       [ITEM       -> ID,
        eq-item    -> eq-id]
      to Identifiers
     }
end Id-sequences
```

APPENDIX D. ASF SPECIFICATION OF THE STRUCTURED DATA TYPES

In this appendix the formal specification is given of the structured data types, to wit matrices and series. Also the data types of variables in the user language, and the data types of variables in a statistical program are defined.

1. STRUCTURED DATA TYPES.

1.1. MATRICES.

1.1.a. Global description.

A matrix in CONDUCTOR is a sequence of scalars, containing the values of the individual matrix elements, combined with a sequence of constant ranges describing the dimensions of the matrix. A matrix X:

```
x11  x12
x21  x22
```

is thought of as a sequence of scalars x11, x12, x21, x22 and the dimension description (1 to 2, 1 to 2). In a well-formed matrix the product of the length the dimensions must be equal to the length of the sequence of reals. In the module *Matrices* the functions for matrix element and submatrix reference are not further specified.

1.2.b. Specification.

```
module Scalar-sequences
begin
    imports Sequences
        { renamed by
            [ SEQ          -> SCALAR-SEQ,
              null         -> null-sca-seq,
              eq-seq       -> eq-sca-seq ]
          Items bound by
            [ ITEM         -> SCALAR,
              eq-item      -> eq          ]
            to Scalars
        }
end Scalar-sequences


module Index-sequences
begin
    imports Sequences
        { renamed by
            [ SEQ          -> INDEX-SEQ,
              null         -> null-ind-seq,
              eq-seq       -> eq-ind-seq ]
          Items bound by
            [ ITEM         -> INDEX,
              eq-item      -> eq          ]
            to Indices
        }
end Index-sequences
```

```
module Constant-ranges
begin
  exports
    begin
      sorts CONST-RANGE
      functions
        c-range:    INDEX # INDEX               -> CONST-RANGE
        length:     CONST-RANGE                 -> INDEX
        eq:         CONST-RANGE # CONST-RANGE   -> BOOL
        eq-length:  CONST-RANGE # CONST-RANGE   -> BOOL
    end

    imports Booleans, Indices

    functions

      -- functions used in specification of well-formed --
      -- constant ranges                                --
      wf-c-range:  INDEX # INDEX              -> CONST-RANGE
      error-c-range:                          -> CONST-RANGE

    variables
      ind,ind1,ind2,ind3,ind4       :-> INDEX
      cr,cr1,cr2                    :-> CONST-RANGE

    equations

    [1]   c-range(ind1,ind2)  = wf-c-range(ind1,ind2)
                                  when  ge(ind2,ind1) = true

    [2]   c-range(ind1,ind2)  = error-c-range when
                                  ge(ind2,ind1) = false

    [3]   length(wf-c-range(ind1,ind2)) = add(1,sub(ind2,
                                              ind1))
            -- the error case is left unspecified --

    [4]   eq-length(cr1,cr2)  = eq(length(cr1),length(cr2))

    [5]   eq(wf-c-range(ind1,ind2),wf-c-range(ind3,ind4))
                              = and( eq(ind1,ind3),eq(ind2,ind4))

            -- the error case is left unspecified --

end Constant-ranges

module Const-range-sequences
begin
    exports
      begin
        functions
          product:    CONST-RANGE-SEQ     ->  INDEX
          length:     CONST-RANGE-SEQ     ->  INDEX
          congruent:  CONST-RANGE-SEQ #
                      CONST-RANGE-SEQ     ->  BOOL
      end

    imports Indices,
            Sequences
            { renamed by
              [ SEQ          -> CONST-RANGE-SEQ,
                null         -> null-cr-seq,
                eq-seq       -> eq-cr-seq ]
              Items bound by
              [ ITEM         -> CONST-RANGE,
                eq-item      -> eq-length   ]
               to Constant-ranges
            }

    variables
      crs,crs1,crs2        :->  CONST-RANGE-SEQ
      cr,cr1,cr2           :->  CONST-RANGE

    equations

     [6]  product(null-cr-seq)                = 0
     [7]  product(add-item(cr,null-cr-seq))   = length(cr)
     [8]  product(add-item(cr,crs))
               = mul(length(cr),product(crs))

     [9]  length(null-cr-seq)                 = 0
    [10]  length(add-item(cr,null-cr-seq))    = length(cr)
    [11]  length(add-item(cr,crs))
               = add(length(cr),length(crs))

    [12]  congruent(crs,crs)                         = true
    [13]  congruent(crs1,crs2)                       = congruent(crs2,crs1)
    [14]  congruent(null-cr-seq,add-item(cr,crs))    = false

    [15]  congruent(add-item(cr1,crs1),add-item(cr2,crs2))
               = and(eq-length(cr1,cr2),
                     congruent(crs1,crs2))

end  Const-range-sequences


module Matrices
begin
   exports
     begin
       sorts MATRIX
         functions
           mat:      SCALAR-SEQ # CONST-RANGE-SEQ   -> MATRIX
           eq:       MATRIX # MATRIX                -> BOOL
           element:  INDEX-SEQ # MATRIX             -> SCALAR
           submat:   CONST-RANGE-SEQ # MATRIX       -> MATRIX
     end

   imports Scalar-sequences, Const-range-sequences, Booleans,
           Index-sequences, Indices
```

```
functions

    -- functions used to specify well-formed matrices --
    wf-mat:    SCALAR-SEQ # CONST-RANGE-SEQ      -> MATRIX
    error-mat:                                   -> MATRIX

variables
    i                  :-> INDEX
    is                 :-> INDEX-SEQ
    ss,ss1,ss2         :-> SCALAR-SEQ
    cr                 :-> CONST-RANGE
    crs,crs1,crs2      :-> CONST-RANGE-SEQ
    m                  :-> MATRIX

equations

 [16] mat(ss,crs) = wf-mat(ss,crs)

         when eq(n-of-items(ss),product(crs)) = true

 [17] mat(ss,crs) = error-mat

         when eq(n-of-items(ss),product(crs)) = false

 [18] eq(wf-mat(ss1,crs1),wf-mat(ss2,crs2))
          = and(eq-sca-seq(ss1,ss2),eq-cr-seq(crs1,crs2))

      -- the error case is left unspecified --

end Matrices
```

1.2 SERIES.

1.2a. Global description.

A (time) series of observations consists of a sequence of scalars (the observations in the series) and a sequence of sample indications. For example, a series of observation on consumption over the period 1945-1949 and 1960-1962 is represented as a scalar sequence containing the 8 observations in this period

12.4 13.9 14.8 13.7 15.9 20.9 22.6 21.8

and a constant ranges sequences, containing the sample description

(1945 to 1949, 1960 to 1962)

In a well-formed series the sum of the length the samples is equal to the number of observations (the length of the sequence of reals).

1.2b. Specification.

```
module Series
begin
 exports
   begin
     sorts SERIES
     functions
       ser:        SCALAR-SEQ # CONST-RANGE-SEQ # ID-SEQ -> SERIES
       eq-series: SERIES # SERIES                        -> BOOL
   end

 imports Const-range-sequences, Scalar-sequences,
        Id-sequences, Booleans

 functions

    -- functions used to specify well-formed series --

    wf-ser:  SCALAR-SEQ # CONST-RANGE-SEQ # ID-SEQ  -> SERIES
    error-ser:                                      -> SERIES

 variables
      s                :-> SERIES
      crs,crs1,crs2    :-> CONST-RANGE-SEQ
      scs,scs1,scs2    :-> SCALAR-SEQ
      ids,ids1,ids2    :-> ID-SEQ

 equations

 [19] ser(scs,crs,ids) = wf-ser(scs,crs,ids)
          when eq(n-of-items(scs),length(crs)) = true

 [20] ser(scs,crs,ids) = error-ser
          when eq(n-of-items(scs),length(crs)) = false

 [21] eq-series(wf-ser(scs1,crs1,ids1),wf-ser(scs2,crs2,ids2))
          = and(eq-sca-seq(scs1,scs2),
               and(eq-cr-seq(crs1,crs2),
                   eq-id-seq(ids1,ids2)))
          -- the error case is left unspecified --

end Series
```

2. DATA TYPES.

2.a. Global description.

Variables in a statistical technique can be of type boolean, index, scalar or matrix. Variables declared in a user session can, besides these data types, also be of the type series. The module *Technique-data* gives the specification of the data types in a statistical technique; the module *User-data* specifies data types in a user session. Besides the data types, also data type descriptions are specified. The simple data types boolean, index and scalar are descibed by a constant function as specified in module *Simple-types*. The type description of variables declared in a user session are specified in module *User-types*. A user type description is either a simple type, a matrix type or a series type description. The last two type descriptions contain a constant range sequence, describing respectively the dimensions or the sample of the data type. Type descriptions for variables in the statistical language are discussed in appendix E.

2.b. Specification.

```
module Technique-data
begin
   exports
     begin
       sorts TECH-DATA
       functions
        uninitialized:                 -> TECH-DATA
         t-data: BOOL                  -> TECH-DATA
         t-data: INDEX                 -> TECH-DATA
         t-data: SCALAR                -> TECH-DATA
         t-data: MATRIX                -> TECH-DATA

        element: INDEX-SEQ # TECH-DATA         -> TECH-DATA
        submat:  CONST-RANGE-SEQ # TECH-DATA   -> TECH-DATA

        eq:      TECH-DATA # TECH-DATA -> BOOL

     end

   imports Booleans, Scalars, Indices, Matrices

   variables
     b,b1,b2            :-> BOOL
     ind,ind1,ind2      :-> INDEX
     sca,sca1,sca2      :-> SCALAR
     mat,mat1,mat2      :-> MATRIX
     td,td1,td2         :-> TECH-DATA
     is                 :-> INDEX-SEQ
     crs                :-> CONST-RANGE-SEQ

   equations

     [22] element(is,t-data(mat))   = t-data(element(is,mat))
     [23] submat(crs,t-data(mat))   = t-data(submat(crs,mat))

     -- other cases of functions element and submat --
     -- result in an error; this is not specified.  --

     [24] eq(td1,td2)                         = eq(td2,td1)
     [25] eq(uninitialized,td)                = false
     [26] eq(t-data(b1),t-data(b2))           = eq(b1,b2)
     [27] eq(t-data(b),t-data(ind))           = false
     [28] eq(t-data(b),t-data(sca))           = false
     [29] eq(t-data(b),t-data(mat))           = false
     [30] eq(t-data(ind1),t-data(ind2))       = eq(ind1,ind2)
     [31] eq(t-data(ind),t-data(sca))         = false
     [32] eq(t-data(ind),t-data(mat))         = false
```

```
     [33] eq(t-data(sca1),t-data(sca2))    = eq(sca1,sca2)
     [34] eq(t-data(sca),t-data(mat))      = false
     [35] eq(t-data(mat1),t-data(mat2))    = eq(mat1,mat2)

end Technique-data


module User-data
begin
  exports
   begin
     sorts USER-DATA
     functions
       u-data:  TECH-DATA              -> USER-DATA
       u-data:  SERIES                 -> USER-DATA

       t-data:  USER-DATA              -> TECH-DATA
       eq:      USER-DATA # USER-DATA -> BOOL

   end

  imports Technique-data, Series, Booleans

  variables
    td,td1,td2               :-> TECH-DATA
    ss,ss1,ss2               :-> SERIES
    ud,ud1,ud2               :-> USER-DATA

  equations

    [36] t-data(u-data(td))   = td
    [37] t-data(u-data(ss))   = uninitialized

    [38] eq(ud,ud)                         = true
    [39] eq(ud1,ud2)                       = eq(ud2,ud1)
    [40] eq(u-data(td1),u-data(td2))       = eq(td1,td2)
    [41] eq(u-data(ss1),u-data(ss2))       = eq-series(ss1,ss2)
    [42] eq(u-data(td),u-data(ss))         = false

end User-data


module Simple-types
  begin
    exports
     begin
      sorts SIMPLE-TYPE
      functions
         bool-type:     -> SIMPLE-TYPE
         index-type:    -> SIMPLE-TYPE
         scalar-type:   -> SIMPLE-TYPE

         eq:  IMPLE-TYPE # SIMPLE-TYPE -> BOOL
     end

    imports Booleans
    variables
       st,st1,st2   :-> SIMPLE-TYPE

    equations

    [43]  eq(st,st)                     = true
    [44]  eq(st1,st2)                   = eq(st2,st1)

    [45]  eq(bool-type,index-type)      = false
    [46]  eq(bool-type,scalar-type)     = false
    [47]  eq(index-type,scalar-type)    = false

end Simple-types


module User-types
begin
   exports
     begin
       sorts USER-TYPE
       functions
          user-type:   SIMPLE-TYPE     -> USER-TYPE
          series-type: CONST-RANGE-SEQ -> USER-TYPE
          matrix-type: CONST-RANGE-SEQ -> USER-TYPE

          eq: USER-TYPE # USER-TYPE -> BOOL
     end

  imports Simple-types, Const-range-sequences, Booleans

  variables
     crs,crs1,crs2       :-> CONST-RANGE-SEQ
     st,st1,st2          :-> SIMPLE-TYPE
     ut,ut1,ut2          :-> USER-TYPE

  equations

  [48] eq(ut1,ut2)                                = eq(ut2,ut1)

  [49] eq(user-type(st1),user-type(st2))          = eq(st1,st2)
  [50] eq(user-type(st),matrix-type(crs))         = false
  [51] eq(user-type(st),series-type(crs))         = false
  [52] eq(matrix-type(crs1),matrix-type(crs2)) = eq-cr-seq(crs1,crs2)
  [53] eq(series-type(crs1),matrix-type(crs2)) = false
  [54] eq(series-type(crs1),series-type(crs2)) = eq-cr-seq(crs1,crs2)

end User-types
```

APPENDIX E. ASF SPECIFICATION OF INDEX EXPRESSIONS, SYMBOLIC RANGES AND TYPE DESCRIPTIONS

1. INDEX EXPRESSIONS

1.1.a. Global description.

In index expressions the operators plus (*abs-plus*), minus (*abs-minus*) and multiply (*abs-mul*) are specified. An index expression consists of identifiers (*abs-ind*) or index constants (*abs-const*). A function *eq-result* specifies the equivalence of two symbolic index expression. An empty abstract syntax tree of an index expression is specified by the constant function *nil*.

1.b. Specification.

```
module Ind-expr-abstr-syntax
begin
  exports
    begin
      sorts IND-EXPR
      functions
         abs-plus:      IND-EXPR # IND-EXPR    -> IND-EXPR
         abs-minus:     IND-EXPR # IND-EXPR    -> IND-EXPR
         abs-mul:       IND-EXPR # IND-EXPR    -> IND-EXPR
         abs-const:     INDEX                  -> IND-EXPR
         abs-ind:       ID                     -> IND-EXPR
         nil:                                  -> IND-EXPR
         eq-result:     IND-EXPR # IND-EXPR    -> BOOL
      end

   imports Indices, Identifiers, Booleans

   variables
     e,e1,e2,e3,e4    :-> IND-EXPR
     c,c1,c2          :-> INDEX
     id1,id2          :-> ID

   equations

   [55] eq-result(e,e)              = true
   [56] eq-result(e1,e2)            = eq-result(e2,e1)

   -- symmetry operators --

   [57] eq-result(abs-plus(e1,e2), abs-plus(e2,e1))  =  true
   [58] eq-result(abs-mul(e1,e2),  abs-mul(e2,e1))   =  true

   -- distributivity operators-

   [59] eq-result(abs-mul(e1,abs-plus(e2,e3)),
                  abs-plus(abs-mul(e1,e2),abs-mul(e1,e3)))
             = true
```

```
[60] eq-result(abs-mul(e1,abs-minus(e2,e3)),
               abs-minus(abs-mul(e1,e2),abs-mul(e1,e3)))
          = true

-- associativity operators --

[61] eq-result(abs-plus(e1,abs-plus(e2,e3)),
               abs-plus(abs-plus(e1,e2),e3))
          = true

[62] eq-result(abs-minus(e1,abs-minus(e2,e3)),
               abs-minus(abs-minus(e1,e2),e3))
          = true

[63] eq-result(abs-mul(e1,abs-mul(e2,e3)),
               abs-mul(abs-mul(e1,e2),e3))
          = true

-- arithmetic rules indices --

[64] eq-result(abs-plus(abs-const(c1),abs-const(c2)),
               abs-const(add(c1,c2)))
          = true

[65] eq-result(abs-minus(abs-const(c1),abs-const(c2)),
               abs-const(sub(c1,c2)))
          = true

[66] eq-result(abs-mul(abs-const(c1),abs-const(c2)),
               abs-const(mul(c1,c2)))
          = true

[67] eq-result(abs-minus(e1,abs-mul(abs-const(c),e2)),
               abs-plus(e1,abs-mul(abs-const(neg(c)),e2)))
          = true

-- null element --

[68] eq-result(abs-mul(abs-const(0),e),abs-const(0))
          = true

[69] eq-result(abs-minus(e,e),abs-const(0))
          = true

-- unity --

[70] eq-result(abs-mul(abs-const(1),e),e) = true

-- empty tree --

[71] eq-result(abs-const(0),nil) = true

-- equivalence --

[72] eq-result(abs-plus(e1,e2),abs-plus(e3,e4))
          = and (eq-result(e1,e3),eq-result(e2,e4))
```

```
    [73] eq-result(abs-minus(e1,e2),abs-minus(e3,e4))
              = and (eq-result(e1,e3),eq-result(e2,e4))

    [74] eq-result(abs-mul(e1,e2),abs-mul(e3,e4))
              = and (eq-result(e1,e3),eq-result(e2,e4))

    [75] eq-result(abs-ind(id1),abs-ind(id2))
              = eq-id(id1,id2)

    [76] eq-result(abs-const(c1),abs-const(c2))
              = eq(c1,c2)

end Ind-expr-abstr-syntax


module Ind-expr-sequences
begin
    imports Sequences
      {renamed by
           [SEQ              -> IND-EXPR-SEQ,
            n-of-items       -> n-of-dims,
            null             -> null-ind-expr-seq,
            item-no          -> dim-no,
            eq-seq           -> eq-dims]
       Items bound by
           [ITEM             -> IND-EXPR,
            eq-item          -> eq-result]
       to Ind-expr-abstr-syntax
      }
end Ind-expr-sequences
```

2. SYMBOLIC RANGES.

2.a. Global description.

Two index expressions form a symbolic range as specified in module *Ranges*. Equality of symbolic ranges is specified using the equivalence of index expressions. The restrictions that specify well formed symbolic ranges are not given in this module but in the modules in appendix I. The functions *range-plus*, *range-min* and *range-mul* specify the range operators.

2.b. Specification.

```
module Ranges
begin
   exports
     begin
       sorts RANGE
       functions
         range:          IND-EXPR # IND-EXPR    -> RANGE
         range-plus:     RANGE # RANGE          -> RANGE
         range-min:      RANGE # RANGE          -> RANGE
         range-mul:      RANGE # RANGE          -> RANGE
```

```
        lower:       RANGE                 -> IND-EXPR
        upper:       RANGE                 -> IND-EXPR

        eq-range:    RANGE # RANGE         -> BOOL
    end

  imports Booleans, Ind-expr-abstr-syntax

  variables
     ie1,ie2,ie3,ie4    :-> IND-EXPR

  equations

  [77] range-plus(range(ie1,ie2),range(ie3,ie4))
           = range(abs-plus(ie1,ie3),abs-plus(ie2,ie4))

  [78] range-min(range(ie1,ie2),range(ie3,ie4))
           = range(abs-minus(ie1,ie3),abs-minus(ie2,ie4))

  [79] range-mul(range(ie1,ie2),range(ie3,ie4))
           = range(abs-mul(ie1,ie3),abs-mul(ie2,ie4))

  [80] lower(range(ie1,ie2)) = ie1

  [81] upper(range(ie1,ie2)) = ie2

  [82] eq-range(range(ie1,ie2),range(ie3,ie4))
            = eq-result(abs-minus(ie2,ie1),
                        abs-minus(ie4,ie3))

end Ranges

module Range-sequences
begin
    imports Sequences
      {renamed by
           [SEQ              -> RANGE-SEQ,
            null             -> null-range,
            n-of-items       -> n-of-rngs,
            item-no          -> dim-no,
            eq-seq           -> eq-ranges]
        Items bound by
           [ITEM             -> RANGE,
            eq-item          -> eq-range]
        to Ranges
      }
end Range-sequences
```

3. TYPE DESCRIPTIONS IN THE STATISTICAL LANGUAGE.

3.a. Global description.

Variables in the statistical language can be of type index, scalar, boolean or matrix. The technical statistician can also determine the visibility of a declared variable (input, output or internal). The type descriptions are specified in module *Tech-types*, the visibility in module *User-visibility*. Two matrix type descriptions are equal if their symbolic dimension ranges are equal, as specified in the function *eq-type*. The function *eq-skelet only checks* if two matrix types have the same number of dimensions.

3.b. Specification.

```
module Technique-types
begin
   exports
      begin
        sorts TECH-TYPE
        functions
         tech-type:    SIMPLE-TYPE -> TECH-TYPE
         matrix-type:  RANGE-SEQ   -> TECH-TYPE

         dim-no:       INDEX # TECH-TYPE        -> RANGE
         n-of-dims:    TECH-TYPE                -> INDEX
         2-dim-matrix: TECH-TYPE                -> BOOL

         eq-type:      TECH-TYPE # TECH-TYPE    -> BOOL
         eq-skelet:    TECH-TYPE # TECH-TYPE    -> BOOL
      end

   imports Booleans, Simple-types, Range-sequences

   variables
       tt,tt1,tt2          :-> TECH-TYPE
       rs,rs1,rs2          :-> RANGE-SEQ
       st,st1,st2          :-> SIMPLE-TYPE
       i                   :-> INDEX

   equations

   [83] dim-no(i,matrix-type(rs))      = dim-no(rs,i)
   [84] n-of-dims(matrix-type(rs))     = n-of-rngs(rs)
   [85] 2-dim-matrix(matrix-type(rs))
            = eq(n-of-rngs(rs),increm(1))
   -- the error cases of the above functions        --
   -- refering to simple types are left unspecified --

   [86] eq-type(tt,tt)          = true
   [87] eq-type(tt1,tt2)        = eq-type(tt2,tt1)

   [88] eq-type(tech-type(st1),tech-type(st2))      = eq(st1,st2)
   [89] eq-type(tech-type(st),matrix-type(rs))      = false
   [90] eq-type(matrix-type(rs1),matrix-type(rs2))  = eq-ranges(rs1,rs2)

   -- skeleton restrictions --

   [91] eq-skelet(tt,tt)                  = true
   [92] eq-skelet(tt1,tt2)                = eq-skelet(tt2,tt1)
   [93] eq-skelet(tech-type(st),tt)       = eq-type(tech-type(st),tt)

   [94] eq-skelet(matrix-type(rs1),matrix-type(rs2))
            = eq(n-of-rngs(rs1),n-of-rngs(rs2))
end Technique-types
```

```
module User-visibility
begin
   exports
     begin
       sorts USER-VIEW
       functions
         input:                          -> USER-VIEW
         output:                         -> USER-VIEW
         internal:                       -> USER-VIEW
         eq:     USER-VIEW # USER-VIEW -> BOOL
       end

   imports Booleans

   variables
      u,u1,u2       :-> USER-VIEW

   equations

   [95] eq(u,u)              = true
   [96] eq(u1,u2)            = eq(u2,u1)
   [97] eq(input,output)     = false
   [98] eq(input,internal)   = false
   [99] eq(output,internal)  = false

end User-visibility
```

APPENDIX F. ASF SPECIFICATION OF THE ABSTRACT SYNTAX OF THE STATISTICAL LANGUAGE

The abstract syntax of the statistical language is divided over several modules.

- a group of modules for the basic elements in the language: variables, expressions, and statements,
- a group of modules for the abstract syntax of the program sections,
- a module for the abstract syntax for a complete program in the statistical language.

1. BASIC ELEMENTS.

1.1. VARIABLES

1.1.a. Global description.

A variable in the statistical langauge is either of type boolean, scalar, index or a matrix. A matrix variable identifier can be followed by a range sequence or a index epression sequence, forming respectively a submatrix or matrix element reference.

1.1.b. Specification.

```
module Variable-abstr-syntax
begin
  exports
     begin
       sorts VARIABLE
       functions
         abs-var:    ID                        -> VARIABLE
         abs-var:    ID # IND-EXPR-SEQ         -> VARIABLE
         abs-var:    ID # RANGE-SEQ            -> VARIABLE
     end

   imports Ind-expr-sequences, Range-sequences, Identifiers,
           Indices

end Variable-abstr-syntax
```

1.2. EXPRESSIONS.

1.2.a. Global description.

An expression in the abstract syntax is either a variable, a constant, an index expression or a function call. The arguments of a function call are a list of expressions.

1.2.b. Specification

```
module Expr-abstr-syntax
begin
   exports
      begin
        sorts EXPR, ARG-LIST
```

```
    functions
      abs-expr: VARIABLE                      -> EXPR
      abs-expr: TECH-DATA                     -> EXPR
      abs-expr: IND-EXPR                      -> EXPR

      abs-f-call: ID # ARG-LIST               -> EXPR
      abs-arg-l:  ARG-LIST # EXPR             -> ARG-LIST
      abs-arg-l:  EXPR                        -> ARG-LIST

  end

  imports Technique-data, Variable-abstr-syntax,
          Ind-expr-abstr-syntax, Identifiers

end Expr-abstr-syntax
```

1.3. STATEMENTS.

1.3.a. Global description.

Statements are used in the implementation, test and exception handler section. A statement is either an assignment a message, a for, a compound or an index assignment statement.

1.3.b. Specification.

```
module Statements-abstr-syntax
begin
   exports
     begin
       sorts STATEMENTS, STATEMENT
       functions
        abs-statmts  : STATEMENTS # STATEMENT   -> STATEMENTS
        abs-statmts  : STATEMENT                -> STATEMENTS

        abs-assgn    : VARIABLE # EXPR          -> STATEMENT
        abs-messtmt  : STRING                   -> STATEMENT

        abs-for      : ID  # IND-EXPR # IND-EXPR #
                       STATEMENTS               -> STATEMENT
        abs-compound : STATEMENTS               -> STATEMENT
        abs-ind-assgn: ID # IND-EXPR            -> STATEMENT
     end

   imports Variable-abstr-syntax, Expr-abstr-syntax, Strings,
           Ind-expr-abstr-syntax

end Statements-abstr-syntax
```

2. PROGRAM SECTIONS IN THE STATISTICAL LANGUAGE.

2.1. DECLARATIONS.

2.1.a. Global description.

Variables in a statistical program must be declared in the input/output section of the program, or in the internal declarations in the other sections. An input/output section consists of two lists of declarations as specified in function *abs-io-sect*. The first list contains the input declarations, the second set the output declarations. A declaration consists of a sequence of identifiers (the names of the declared variables), the type description of the declared variables and a message string.

```
module Decl-abstr-syntax
begin
  exports
    begin
      sorts  IO-SEC, INTERN-DECLS, DECLS, DECL
        functions
        abs-io-sect:     DECLS # DECLS     -> IO-SEC
        abs-intern-decl: DECLS             -> INTERN-DECLS
        abs-empty-decls:                   -> DECLS
        abs-decls:       DECLS # DECL      -> DECLS
        abs-decls:       DECL              -> DECLS

        abs-decl:  ID-SEQ # TECH-TYPE # STRING  -> DECL
      end

    imports Strings, Id-sequences,
            User-visibility, Technique-types

end Decl-abstr-syntax
```

2.2. THE IMPLEMENTATION SECTION.

2.2.a. Global description.

The implementation section of a statistical program contains the equations that describe the calculation of the resulting statistics. An implementation section consists of internal declarations and statements (equations that describe the calculation process).

2.2.b. Specification.

```
module Impl-abstr-syntax
begin
   exports
     begin
       sorts IMPL-SEC
       functions
         abs-impl-sec: INTERN-DECLS # STATEMENTS -> IMPL-SEC
     end
   imports Decl-abstr-syntax,  Statements-abstr-syntax

end Impl-abstr-syntax
```

2.3. THE TEST SECTION.

2.3.a. Global description.

In a test section the technical statistician can describe a test and indicate under which conditions the test should interrupt the calculation process (raise an exception). A raise statement consists of an exception identifier and a boolean expression as specified in function *abs-raise*.

2.3.b. Specfication.

```
module Test-abstr-syntax
begin
   exports
     begin
      sorts TEST-SEC, RAISES, RAISE
      functions
       abs-test-sec: INTERN-DECLS # STATEMENTS # RAISES -> TEST-SEC

       abs-raises:   RAISES # RAISE   -> RAISES
       abs-raises:   RAISE            -> RAISES
       abs-raise :   ID  #  EXPR      -> RAISE
     end

   imports Statements-abstr-syntax, Expr-abstr-syntax,
           Identifiers, Decl-abstr-syntax

end Test-abstr-syntax
```

2.4. THE EXCEPTION HANDLER SECTION.

2.4.a. Global description.

An exception handler consists of: the exception identifier, the internal declarations, and the statements. The statements in an excpetion handler are only executed when the exception occurs.

2.4.b. Specification.

```
module Handler-abstr-syntax
begin
   exports
     begin
      sorts HANDL-SEC, HANDLER
      functions
       abs-handl-sec: HANDL-SEC # HANDLER  -> HANDL-SEC
       abs-handl-sec: HANDLER              -> HANDL-SEC

       abs-handler: ID # INTERN-DECLS # STATEMENTS -> HANDLER
      end

   imports Identifiers, Decl-abstr-syntax,
           Statements-abstr-syntax

end Handler-abstr-syntax
```

3. ABSTRACT SYNTAX OF A STATISTICAL PROGRAM.

3.a. Global description.

A statistical program is a list of sections.

3.b. Specfication.

```
module Statistical-programs
begin
   exports
     begin
      sorts STAT-PRO, SECTION
      functions
        abs-prog:  STAT-PRO # SECTION     -> STAT-PRO
        abs-prog:  SECTION                -> STAT-PRO

        abs-name:  ID                     -> SECTION
        abs-sect:  IO-SEC                 -> SECTION
        abs-sect:  IMPL-SEC               -> SECTION
        abs-sect:  TEST-SEC               -> SECTION
        abs-sect:  HANDL-SEC              -> SECTION
     end
   imports Decl-abstr-syntax, Impl-abstr-syntax,
           Test-abstr-syntax, Handler-abstr-syntax,
           Identifiers

end Statistical-programs
```

APPENDIX G. ASF SPECIFICATION OF THE SYMBOL TABLES

In this appendix the symbol tables are specified. First a symbol table is specified for each statistical technique (read statistical program). This table contains the type description, the value and scme additional information for each variable that is declared in a statistical technique. Also a symbol table is defined for the user of a statistical technique. This table contains the type description and the value of each variable declared in a user session. In this appendix it is also specified how declaration sections in a statistical program are stored in a technique symbol table. The storage of declarations in the user languages is specified in appendix N.

1. THE SYMBOL TABLES.

1.a. Global description.

The value, the type, the message string and the visibility of the variables in a statistical program are stored in the technique symbol table. In the user symbol table the value and the type of variables declared in a user session are stored.
All information for a variable is stored in one table. Instead, seperate tables could have been defined for each separate peace of information in these tables. In order to keep the number of tables low, we have chosen for the combined approach, in which both information needed for type checking and information needed for the evaluation of programs is stored in the same table.

1.b. Specification.

```
module Techn-symtab-info
begin
   exports
     begin
       sorts TECH-INFO
       functions
         ts-info: TECH-TYPE # USER-VIEW # TECH-DATA # STRING -> TECH-INFO

         type:    TECH-INFO               -> TECH-TYPE
         data:    TECH-INFO               -> TECH-DATA
         view:    TECH-INFO               -> USER-VIEW
         string:  TECH-INFO               -> STRING
         eq:      TECH-INFO # TECH-INFO   -> BOOL
     end

   imports Technique-data, Technique-types, Booleans,
           User-visibility, Strings

   variables
       t,t1,t2           :-> TECH-TYPE
       td,td1,td2        :-> TECH-DATA
       uv,uv1,uv2        :-> USER-VIEW
       str,str1,str2     :-> STRING

   equations

   [100] data(ts-info(t,uv,td,str))  = td
   [101] type(ts-info(t,uv,td,str))  = t
   [102] view(ts-info(t,uv,td,str))  = uv
   [103] string(ts-info(t,uv,td,str))= str

   [104] eq(ts-info(t1,uv1,td1,str1),ts-info(t2,uv2,td2,str2))
           = and(eq-type(t1,t2),
                 and(eq(uv1,uv2),
                     and(eq(td1,td2),
                         eq-str(str1,str2))))

end Techn-symtab-info

module Techn-symtabs
begin
   exports
     begin
       functions
       -- this function stores values in the symbol table --
       -- during the evaluation of a statistical program. --
       insert-data: ID # TECH-DATA # TECH-SYMTAB -> TECH-SYMTAB
     end
   imports Tables
      { renamed by
         [ TABLE       -> TECH-SYMTAB,
           null-table  -> empty-mem]
        Entries bound by
         [ ENTRY       -> TECH-INFO,
           eq-entry    -> eq]
         to Techn-symtab-info
        Addresses bound by
         [ ADDRESS     -> ID,
           eq-addr     -> eq-id ]
         to Identifiers
      }

   variables
        id            :-> ID
        ts            :-> TECH-SYMTAB
        td1,td2       :-> TECH-DATA
        tt            :-> TECH-TYPE
        uv            :-> USER-VIEW
        str           :-> STRING

   equations

   [105] insert-data(id,td1,
                     insert(id,ts-info(tt,uv,td2,str),ts))
          = insert(id,ts-info(tt,uv,td1,str),ts)

end Techn-symtabs
```

```
module User-symtab-info
begin
   exports
      begin
        sorts USER-INFO
        functions
          us-info: USER-TYPE # USER-DATA -> USER-INFO
          data:    USER-INFO             -> USER-DATA
          type:    USER-INFO             -> USER-TYPE
          eq:      USER-INFO # USER-INFO -> BOOL
      end

   imports User-types, User-data, Booleans

   variables
        ut,ut1,ut2 :-> USER-TYPE
        ud,ud1,ud2 :-> USER-DATA

   equations

   [106] data(us-info(ut,ud))                        = ud
   [107] type(us-info(ut,ud))                        = ut
   [108] eq(us-info(ut1,ud1),us-info(ut2,ud2)) = and(eq(ut1,ut2),
                                                         eq(ud1,ud2))
end User-symtab-info
```

```
module User-symtabs
begin
   exports
     begin
      functions
        -- this function stores types in the symbol table --
        -- during the evaluation of a user program.        --
        insert-type: ID # USER-TYPE # USER-SYMTAB -> USER-SYMTAB
        -- this function stores values in the symbol table --
        -- during the evaluation of a user program.        --
        insert-data: ID # USER-DATA # USER-SYMTAB -> USER-SYMTAB
     end

   imports Tables
        { renamed by
           [ TABLE       -> USER-SYMTAB,
             null-table -> empty-u-symtab]
          Entries bound by
           [ ENTRY        -> USER-INFO,
             eq-entry     -> eq]
           to User-symtab-info
          Addresses bound by
           [ ADDRESS      -> ID,
             eq-addr      -> eq-id ]
           to Identifiers
        }

   variables
         id           :-> ID
         us           :-> USER-SYMTAB
         ud,ud1,ud2    :-> USER-DATA
         ut,ut1,ut2    :-> USER-TYPE

   equations

   [109] insert-type(id,ut1,insert(id,us-info(ut2,ud),us))
           = insert(id,us-info(ut1,ud),us)

   [110] insert-data(id,ud1,insert(id,us-info(ut,ud2),us))
           = insert(id,us-info(ut,ud1),us)

end User-symtabs
```

2.1. STORING DECLARATIONS IN THE TECHNIQUE SYMBOL TABLE.

2.1.a. Global description.

For each variable declared in a statistical program an entry is made in the technique symbol table. This entry contains the type information, the visibility and the message string in the declaration. The value of the variable is uninitialized. The constant function *uninitialized* is specified in module *Technique-data*.

2.2.b. Specification.

```
module Store-declarations
begin
  exports
     begin
       functions
         store-io-sect:      TECH-SYMTAB # IO-SEC       -> TECH-SYMTAB
         store-intern-decls: TECH-SYMTAB # INTERN-DECLS -> TECH-SYMTAB

         store-decls:    TECH-SYMTAB # DECLS # USER-VIEW -> TECH-SYMTAB
         store-decl:     TECH-SYMTAB # DECL # USER-VIEW  -> TECH-SYMTAB

         store-id:    TECH-SYMTAB # ID # TECH-TYPE # USER-VIEW # STRING
                      -> TECH-SYMTAB
     end

  imports Techn-symtabs, Decl-abstr-syntax

  variables
     ts                                  :-> TECH-SYMTAB
     decls,inp-decls,out-decls           :-> DECLS
     decl                                :-> DECL
     ids                                 :-> ID-SEQ
     id                                  :-> ID
     tt                                  :-> TECH-TYPE
     uv                                  :-> USER-VIEW
     mess                                :-> STRING
```

```
equations

[111] store-io-sect(ts,abs-io-sect(inp-decls,out-decls))
        = store-decls(store-decls(ts,inp-decls,input),
                      out-decls,output)

[112] store-intern-decls(ts,abs-intern-decl(decls))
        = store-decls(ts,decls,internal)

[113] store-decls(ts,abs-empty-decls,uv)
        = ts

[114] store-decls(ts,abs-decls(decls,decl),uv)
        = store-decl(store-decls(ts,decls,uv),decl,uv)

[115] store-decls(ts,abs-decls(decl),uv)
        = store-decl(ts,decl,uv)

[116] store-decl(ts,abs-decl(add-item(id,ids),tt,mess),uv)
       = store-id(store-decl(ts,abs-decl(ids,tt,mess),uv),
         id,tt,uv,mess)

[117] store-decl(ts,abs-decl(add-item(id,null-id-seq),
              tt,mess),uv)
       = store-id(ts,id,tt,uv,mess)

[118] store-id(ts,id,tt,uv,mess)
      = insert(id,
               ts-info(tt,uv,uninitialized,mess),
               ts)

end Store-declarations
```

APPENDIX H. ASF SPECIFICATION OF THE STATIC SYMBOLIC TYPE CHECKER

This appendix contains the specification of the static symbolic type checking mechanism of CONDUCTOR. It contains modules to specify:
- the dimension and skelet restrictions on function arguments,
- the type of the result of a function,
- the type of variables and expressions,
- symbolic type checking of a statistical program.

1. SYMBOLIC TYPE CHECKING OF FUNCTION CALLS.

1.a. Global description.

The type restrictions of the function arguments are expressed in terms of skeleton and dimension restrictions. Two skeletons of a type description are equal if the type descriptions have the same simple type or, in case of a matrix type description, the number of dimensions is equal.
Consider a function sum (A,B) that adds two matrices, A and B. A skeleton restriction on the function sum is the restriction, that both arguments must be matrices with an equal number of dimensions. The range restrictions on the function sum are the restrictions that the ranges of the arguments must be equal.
A type list is used to store the type decriptions of the arguments of a function call. A skeleton restriction consists of either two indices, indicating the position of two arguments in the type list that must have equal skeletons, or an index and a type indicating the skeleton of a referenced argument must be equal to the skeleton of a given type. For the function sum the skeleton restriction consists of the constant range (1,2), indicating that the skeletons of the first and second element are equal. The range restrictions are specified in a similar way, but instead of an index a constant range is used. The first element of this constant range gives the argument number, the second element the dimension number. For the function sum the range restrictions consists of two constant range sequences

{(1,1),(2,1)} and {(1,2),(2,2)}.

The result of a function in the statistical language is either a simple type or a matrix type. The ranges describing the dimension of a matrix can be retrieved from the type list, using a constant range as with range restrictions. For the function sum, the type of the result is given by the constant range sequence {(1,1),(1,2)} stating that the resulting type is a matrix type and that the dimensions are the first and second dimension of the first argument.
The restrictions on the arguments and the type of the result of a function are stored in the function type table. The type restrictions of the functions actually defined in the statistical language are left unspecified. The parameter *Current-func-types* in module *Func-type-table* emphasises that the design of CONDUCTOR is independent of the functions actually defined in the statistical language.

1.b. Specification.

```
module Type-lists
begin
   exports
      begin
         functions
           arg-range: CONST-RANGE # TYPE-LIST    -> RANGE
      end
   imports Constant-ranges, Sequences
       { renamed by
             [SEQ      -> TYPE-LIST,
              item-no -> argument]
         Items bound by
             [ITEM    -> TECH-TYPE,
              eq-item -> eq-type]
         to Technique-types
       }
   variables
     ind1,ind2    :-> INDEX
     tl           :-> TYPE-LIST

   equations

  [119]   arg-range(c-range(ind1,ind2),tl)
                 = dim-no(ind2,argument(tl,ind1))

end Type-lists

module Skelet-restrictions
begin
   exports
      begin
         sorts  SKELET-RESTR
         functions
           restr:    INDEX # INDEX       -> SKELET-RESTR
           restr:    INDEX # TECH-TYPE   -> SKELET-RESTR

           eq-restr: SKELET-RESTR # SKELET-RESTR -> BOOL
           check:    SKELET-RESTR # TYPE-LIST    -> BOOL
      end

   imports Indices, Technique-types, Type-lists, Booleans

   variables
      sr,sr1,sr2                    :-> SKELET-RESTR
      ind,ind1,ind2,ind3,ind4       :-> INDEX
      tl                            :-> TYPE-LIST
      t,t1,t2                       :-> TECH-TYPE

   equations

   [120]    eq-restr(sr1,sr2) = eq-restr(sr2,sr1)

   [121]    eq-restr(restr(ind1,ind2),restr(ind3,ind4))
              =  and( eq(ind1,ind3),eq(ind2,ind4))

   [122]    eq-restr(restr(ind1,t1),restr(ind2,t2))
              =  and( eq(ind1,ind2),eq-skelet(t1,t2))

   [123]    eq-restr(restr(ind1,ind2),restr(ind3,t))
              =  false

   [124]    check(restr(ind1,ind2),tl)
              =  eq-skelet(argument(tl,ind1),argument(tl,ind2))
   [125]    check(restr(ind,t),tl)
              =  eq-skelet(argument(tl,ind),t)

end Skelet-restrictions


module Skelet-restr-sequences
begin
   exports
      begin
         functions
           check: SKELET-RESTR-SEQ # TYPE-LIST  -> BOOL
      end
   imports Sequences
              {renamed by
                 [SEQ          -> SKELET-RESTR-SEQ,
                  eq-seq       -> eq-skelet-restr]
               Items bound by
                 [ITEM         -> SKELET-RESTR,
                  eq-item      -> eq-restr]
               to Skelet-restrictions
              }

   variables
      sr      :-> SKELET-RESTR
      srs     :-> SKELET-RESTR-SEQ
      tl      :-> TYPE-LIST

   equations

   [126] check(add-item(sr,null),tl) = check(sr,tl)

   [127] check(add-item(sr,srs),tl)
           = and(check(sr,tl),check(srs,tl))

end Skelet-restr-sequences

module Dim-restrictions
begin
   exports
      begin
         sorts        DIM-RESTR
         functions
           dim-restr: CONST-RANGE # CONST-RANGE  -> DIM-RESTR
           dim-restr: CONST-RANGE # RANGE        -> DIM-RESTR

           eq-restr:  DIM-RESTR # DIM-RESTR      -> BOOL
           check:     DIM-RESTR # TYPE-LIST      -> BOOL
      end
   imports Constant-ranges, Ranges, Type-lists

   variables

      dr,dr1,dr2            :-> DIM-RESTR
      cr,cr1,cr2,cr3,cr4    :-> CONST-RANGE
      tl                    :-> TYPE-LIST
      r,r1,r2               :-> RANGE

   equations

   [128]   eq-restr(dr1,dr2) = eq-restr(dr2,dr1)

   [129]   eq-restr(dim-restr(cr1,cr2),dim-restr(cr3,cr4))
              = and( eq(cr1,cr3),eq(cr2,cr4))

   [130]   eq-restr(dim-restr(cr1,r1),dim-restr(cr2,r2))
              = and( eq(cr1,cr2),eq-range(r1,r2))

   [131]   eq-restr(dim-restr(cr1,cr2),dim-restr(cr3,r))
              = false

   [132]   check(dim-restr(cr1,cr2),tl)
              = eq-range(arg-range(cr1,tl),arg-range(cr2,tl))

   [133]   check(dim-restr(cr1,r),tl)
              = eq-range(arg-range(cr1,tl),r)

end Dim-restrictions


module Dim-restr-sequences
begin
   exports
      begin
         functions
            check: DIM-RESTR-SEQ # TYPE-LIST  -> BOOL
      end
   imports Sequences
              {renamed by
                 [SEQ           -> DIM-RESTR-SEQ,
                  null          -> no-dim-restr,
                  eq-seq        -> eq-dres-seq]
               Items bound by
                 [ITEM          -> DIM-RESTR,
                  eq-item       -> eq-restr]
               to Dim-restrictions
              }

   variables
      dr       :-> DIM-RESTR
      drs      :-> DIM-RESTR-SEQ
      tl       :-> TYPE-LIST

   equations

   [134] check(add-item(dr,no-dim-restr),tl)  = check(dr,tl)

   [135] check(add-item(dr,drs),tl)
            = and(check(dr,tl),check(drs,tl))

end Dim-restr-sequences


module Result-types
begin
    exports
      begin
       sorts RESULT-TYPE
       functions
           res-type: TECH-TYPE               -> RESULT-TYPE
           res-type: CONST-RANGE-SEQ         -> RESULT-TYPE
           result:   RESULT-TYPE # TYPE-LIST -> TECH-TYPE
      end

    imports Technique-types, Const-range-sequences,
            Type-lists

    functions
      res-ranges: CONST-RANGE-SEQ # TYPE-LIST -> RANGE-SEQ

    variables
       t       :-> TECH-TYPE
       tl      :-> TYPE-LIST
       crs     :-> CONST-RANGE-SEQ
       cr      :-> CONST-RANGE

    equations

    [136]  result(res-type(t),tl)   = t
    [137]  result(res-type(crs),tl) = matrix-type(res-ranges(crs,tl))

    [138]  res-ranges(add-item(cr,crs),tl)
              = add-item(arg-range(cr,tl),res-ranges(crs,tl))

    [139]  res-ranges(add-item(cr,null-cr-seq),tl)
              = add-item(arg-range(cr,tl),null-range)

end Result-types


module Function-types
begin
  exports
    begin
     sorts FUNC-TYPE-INFO
      functions
      f-info:  INDEX # SKELET-RESTR-SEQ #
               DIM-RESTR-SEQ # RESULT-TYPE
                                          -> FUNC-TYPE-INFO
      result:  FUNC-TYPE-INFO # TYPE-LIST   -> TECH-TYPE
      n-arg:      FUNC-TYPE-INFO                   -> INDEX
      check:      FUNC-TYPE-INFO # TYPE-LIST       -> BOOL
      eq-f-type:  FUNC-TYPE-INFO # FUNC-TYPE-INFO  -> BOOL
    end

  imports Indices, Technique-types, Dim-restr-sequences,
        Skelet-restr-sequences, Result-types

  variables
    ind          :-> INDEX
    srs          :-> SKELET-RESTR-SEQ
    drs          :-> DIM-RESTR-SEQ
    rt           :-> RESULT-TYPE
    tl           :-> TYPE-LIST

  equations

  [140]  n-arg(f-info(ind,srs,drs,rt))      = ind
  [141]  result(f-info(ind,srs,drs,rt),tl)  = result(rt,tl)

  [142]  check(f-info(ind,srs,drs,rt),tl)
            = and (check(srs,tl),check(drs,tl))

end Function-types
```

```
module Func-type-tables
begin
     parameters Current-func-types
       begin
        functions
         ini-ftype-tab:      -> FUNC-TYPE-TABLE
       end Current-func-types

     exports
       begin
        functions
         result-f:         ID # TYPE-LIST -> TECH-TYPE
         check-f-restr:    ID # TYPE-LIST -> BOOL
         get-n-arg:        ID             -> INDEX
       end

     imports Tables
       { renamed by
           [ TABLE             -> FUNC-TYPE-TABLE ]
         Entries bound by
           [ ENTRY             -> FUNC-TYPE-INFO,
             eq-entry          -> eq-f-type ]
           to Function-types
         Addresses bound by
           [ ADDRESS           -> ID,
             eq-addr           -> eq-id ]
           to Identifiers
       }

     variables
        id            :-> ID
        tl            :-> TYPE-LIST

     equations

     [143]  check-f-restr(id,tl) = check(id^ini-ftype-tab,tl)
     [144]  result-f(id,tl)      = result(id^ini-ftype-tab,tl)
     [145]  get-n-arg(id)        = n-arg(id^ini-ftype-tab)

end Func-type-tables
```

2. THE DATA TYPE OF THE BASIC ELEMENTS OF THE STATISTICAL LANGUAGE.

2.a. Global description.

In this section the type of variables, expressions and statements is specified. The type description of a variable is found in the technique symbol table. A matrix element refence is specified to be of type scalar; a submatrix of type matrix. The number of dimensions in matrix refferences must be equal to the number of dimensions in the matrix declaration.

2.b. Specification.

```
module Stat-check-variables
begin
     exports
       begin
         functions
         var-type: VARIABLE # TECH-SYMTAB -> TECH-TYPE
       end

     imports Variable-abstr-syntax, Techn-symtabs

     variables
       tst                    :-> TECH-SYMTAB
       id                     :-> ID
       inds                   :-> IND-EXPR-SEQ
       rngs                   :-> RANGE-SEQ

     equations

     [146] var-type(abs-var(id),tst)       = type(id^tst)

     [147] var-type(abs-var(id,inds),tst) = tech-type(scalar-type)

       when eq(n-of-dims(type(id^tst)),n-of-dims(inds)) = true

     [148] var-type(abs-var(id,rngs),tst) = matrix-type(rngs)

       when eq-skelet(matrix-type(rngs),type(id^tst))   = true

 -- the error cases in the above equations are not specified --

end Stat-check-variables
```

```
module Stat-check-expressions
begin
  exports
   begin
     functions
      expr-type: EXPR  # TECH-SYMTAB                   -> TECH-TYPE
      arg-types: ARG-LIST # TYPE-LIST # TECH-SYMTAB   -> TYPE-LIST
   end

  imports Expr-abstr-syntax, Type-lists,
          Func-type-tables, Stat-check-variables

  variables
     exp              :-> EXPR
     tl               :-> TYPE-LIST
     tst              :-> TECH-SYMTAB
     var              :-> VARIABLE
     sca              :-> SCALAR
     ind              :-> INDEX
     b                :-> BOOL
     f-id             :-> ID
     argl             :-> ARG-LIST

   equations

   -- simple expressions --

   [149] expr-type(abs-expr(var),tst)          = var-type(var,tst)
   [150] expr-type(abs-expr(t-data(sca)),tst) = tech-type(scalar-type)
   [151] expr-type(abs-expr(t-data(ind)),tst) = tech-type(scalar-type)
   [152] expr-type(abs-expr(t-data(b)),tst)   = tech-type(bool-type)

    -- function calls --

   [153] expr-type(abs-f-call(f-id,argl),tst)
            =  result-f(f-id,arg-types(argl,null,tst))

     when check-f-restr(f-id,arg-types(argl,null,tst)) = true

 -- the error case in the above equation is not specified --

   [154] arg-types(abs-arg-l(argl,exp),tl,tst)
            = add-item(expr-type(exp,tst),arg-types(argl,tl,tst))

   [155] arg-types(abs-arg-l(exp),tl,tst)
            = add-item(expr-type(exp,tst),null)

end Stat-check-expressions


module Stat-check-statements
begin
   exports
     begin
      functions
        type-check-stmnts: STATEMENTS # TECH-SYMTAB    -> BOOL
        type-check-stmnt:  STATEMENT # TECH-SYMTAB     -> BOOL
     end

   imports Statements-abstr-syntax, Stat-check-expressions

   variables
    var          :-> VARIABLE
    expr         :-> EXPR
    tst          :-> TECH-SYMTAB
    id           :-> ID
    ie1,ie2      :-> IND-EXPR
    m            :-> STRING
    stmnts       :-> STATEMENTS
    stmnt        :-> STATEMENT

   equations

   [156] type-check-stmnt(abs-assgn(var,expr),tst)
           = eq-type(var-type(var,tst),expr-type(expr,tst))

   [157] type-check-stmnt(abs-for(id,ie1,ie2,stmnts),tst)
           = type-check-stmnts(stmnts,tst)

   [158] type-check-stmnt(abs-messtmt(m),tst)
           = true

   [159] type-check-stmnt(abs-compound(stmnts),tst)
           = type-check-stmnts(stmnts,tst)

   [160] type-check-stmnt(abs-ind-assgn(id,ie1),tst)
           = true

end Stat-check-statements
```

3. STATIC TYPE CHECKING OF SECTIONS OF A STATISTICAL PROGRAM.

3.1.a. Global description.

The static type checking of the sections gives an updated technique symbol table. The internal declarations in each section are stored in the technique symbol table, and the type restrictions in the statements are checked.

3.1.b. Specification

```
module Stat-check-impl
begin
   exports
     begin
      functions
        type-check-impl-s: IMPL-SEC # TECH-SYMTAB -> TECH-SYMTAB
     end

   imports Stat-check-statements, Store-declarations,
           Impl-abstr-syntax
```

```
  variables
    decl            :-> INTERN-DECLS
    stmnts          :-> STATEMENTS
    tst             :-> TECH-SYMTAB

  equations

  [161] type-check-impl-s(abs-impl-sec(decl,stmnts),tst)
          = store-intern-decls(tst,decl)

       when type-check-stmnts(stmnts,store-intern-decls(tst,decl))
            = true

-- the error case in the above equation is not specified --

end Stat-check-impl


module Stat-check-tests
begin
  exports
    begin
      functions
        type-check-test-s: TEST-SEC # TECH-SYMTAB  -> TECH-SYMTAB
    end

  imports Stat-check-statements, Store-declarations,
          Test-abstr-syntax

  variables
   decl          :-> INTERN-DECLS
   stmnts        :-> STATEMENTS
   rss           :-> RAISES
   tst           :-> TECH-SYMTAB

  equations

  [162] type-check-test-s(abs-test-sec(decl,stmnts,rss),tst)
         = store-intern-decls(tst,decl)

      when type-check-stmnts(stmnts,store-intern-decls(tst,decl))
           = true

-- the error case in the above equation is not specified --

end Stat-check-tests


module Stat-check-handlers
begin
  exports
    begin
     functions
       type-check-handl-s: HANDL-SEC # TECH-SYMTAB -> TECH-SYMTAB
       type-check-handler: HANDLER # TECH-SYMTAB   -> TECH-SYMTAB
    end
```

```
imports Stat-check-statements, Store-declarations,
        Handler-abstr-syntax

variables
   h-sec           :-> HANDL-SEC
   hndl            :-> HANDLER
   decl            :-> INTERN-DECLS
   stmnts          :-> STATEMENTS
   tst             :-> TECH-SYMTAB
   id              :-> ID

equations

[163]   type-check-handl-s(abs-handl-sec(h-sec,hndl),tst)
              = type-check-handl-s(abs-handl-sec(hndl),
                                   type-check-handl-s(h-sec,tst))

[164]   type-check-handl-s(abs-handl-sec(hndl),tst)
              = type-check-handler(hndl,tst)

[165]   type-check-handler(abs-handler(id,decl,stmnts),tst)
              = store-intern-decls(tst,decl)

        when type-check-stmnts(stmnts,store-intern-decls(tst,decl))
              = true

        -- the error case in the above equation is not specified --

end Stat-check-handlers


module Static-type-checking
begin
   exports
     begin
       functions
          type-check-sct: SECTION # TECH-SYMTAB  -> TECH-SYMTAB
          type-check-pro: STAT-PRO               -> TECH-SYMTAB
     end

   imports Statistical-programs, Store-declarations,
           Stat-check-handlers, Stat-check-tests,
           Stat-check-impl

   variables
       id        :-> ID
       tst       :-> TECH-SYMTAB
       io-sec    :-> IO-SEC
       impl-sec  :-> IMPL-SEC
       test-sec  :-> TEST-SEC
       handl-sec :-> HANDL-SEC
       sect      :-> SECTION
       pro       :-> STAT-PRO
```

```
equations

[166]  type-check-sct(abs-name(id),tst)
         = tst

[167]  type-check-sct(abs-sect(io-sec),tst)
         = store-io-sect(tst,io-sec)

[168]  type-check-sct(abs-sect(impl-sec),tst)
         = type-check-impl-s(impl-sec,tst)

[169]  type-check-sct(abs-sect(test-sec),tst)
         = type-check-test-s(test-sec,tst)

[170]  type-check-sct(abs-sect(handl-sec),tst)
         = type-check-handl-s(handl-sec,tst)

 -- the statistical program --

[171]  type-check-pro(abs-prog(pro,sect))
         = type-check-sct(sect,type-check-pro(pro))

[172] type-check-pro(abs-prog(sect))
         = type-check-sct(sect,empty-mem)

end Static-type-checking
```

APPENDIX I. ASF SPECIFICATION OF THE GENERATION OF THE INPUT RESTRICTIONS

This appendix contains the formal specification of the input restriction generation mechanism in CONDUCTOR. Input restrictions are generated in order to guarantee that no type or dimension bound conflicts will occur during the execution of a statistical technique. Modules are given to specify: (1) input restrictions, (2) the calculation of a symbolic range for an index expression, (3) the assignment of a symbolic range to index variables, (4) the generation of input restrictions for variables, expressions and statements, (5) the generation of input restrictions for sections of a statistical program, and (6) the generation of input restrictions for a statistical program.

1. INPUT RESTRICTIONS.

1.a. Global description.

An input restriction is defined as a restriction that an index expression is greater or equal zero. The index expressions in the input restrictions may only consist of input variables and constants. This is specified in module *Input-var-restrictions.*
In module *Ind-expr-order* the order relation on index expressions is specified. If an index expression always yields a greater (equal or smaller) value than an other index expression, for all possible values of the variables in the expressions, the order relation is greater (equal or smaller). For pairs of index expressions where this problem is undecidable, the relation is undecidable.
In module *Input-restr-sequence* it is specified how an input restriction is added to a sequence of these restrictions, if the order relation of two index expressions is undecidable.
In module *Range-restrictions* the following restrictions on ranges are defined:

- positive restrictions,the upper bound of a dimension range must be larger than the lower bound, and
- subrange restriction, the upper bound of the subrange must be smaller than the upper bound of the range, and the lower bound greater than the lower bound of the range.

1.b.Specification.

```
module Input-var-restrictions
begin
   exports
      begin
        functions
          are-input-vars: IND-EXPR # TECH-SYMTAB      -> BOOL
      end

   imports Ind-expr-abstr-syntax, Techn-symtabs, User-visibility

   variables
     ie,ie1,ie2          :-> IND-EXPR
     ts                  :-> TECH-SYMTAB
     c                   :-> INDEX
     id                  :-> ID
   equations

    [174] are-input-vars(abs-plus(ie1,ie2),ts)
           = and(are-input-vars(ie1,ts),are-input-vars(ie2,ts))

    [175] are-input-vars(abs-minus(ie1,ie2),ts)
           = and(are-input-vars(ie1,ts),are-input-vars(ie2,ts))

    [176] are-input-vars(abs-mul(ie1,ie2),ts)
           = and(are-input-vars(ie1,ts),are-input-vars(ie2,ts))

    [177] are-input-vars(abs-ind(id),ts)
           = if(eq(view(id^ts),input),true,false)

    [178] are-input-vars(abs-const(c),ts)
           = true

end Input-var-restrictions


module Input-restrictions
begin
     exports
       begin
         sorts INP-RESTR
         functions
           pos-restr:    IND-EXPR                 -> INP-RESTR
           inp-restr:    IND-EXPR # TECH-SYMTAB -> INP-RESTR
           eq-inp-restr: INP-RESTR # INP-RESTR  -> BOOL
       end

     imports Ind-expr-abstr-syntax, Input-var-restrictions

     functions
           error-inp-restr:       -> INP-RESTR

     variables
          ie,ie1,ie2    :-> IND-EXPR
          ts            :-> TECH-SYMTAB

     equations

     [179] inp-restr(ie,ts) = pos-restr(ie)

             when are-input-vars(ie,ts) = true

     [180] inp-restr(ie,ts) = error-inp-restr

             when are-input-vars(ie,ts) = false

     [181] eq-inp-restr(pos-restr(ie1),pos-restr(ie2))
             = eq-result(ie1,ie2)

         -- the error case is left unspecified --

end Input-restrictions
```

```
module Order
begin
   exports
     begin
       sorts ORDER
       functions
         greater:                            -> ORDER
         less:                               -> ORDER
         equal:                              -> ORDER
         undecided:                          -> ORDER
         eq:                 ORDER # ORDER -> BOOL
     end

   imports  Booleans

   variables
       o,o1,o2          :-> ORDER

   equations
     [182] eq(o,o)                   = true
     [183] eq(o1,o2)                 = eq(o2,o1)
     [184] eq(equal,less)            = false
     [185] eq(equal,greater)         = false
     [186] eq(equal,undecided)       = false
     [187] eq(less,greater)          = false
     [188] eq(less,undecided)        = false
     [189] eq(greater,undecided)     = false

end Order


module Ind-expr-order
begin
   exports
     begin
       functions
         order:     IND-EXPR # IND-EXPR -> ORDER
     end

   imports Order, Ind-expr-abstr-syntax

   variables
       ie1,ie2          :-> IND-EXPR
       c                :-> INDEX

   equations

   [190] order(ie1,ie2) = if (eq(c,0), equal,
                                    if (ge(c,0), greater, less))

       when eq-result(abs-minus(ie1,ie2),abs-const(c)) = true

   [191] order(ie1,ie2) = undecided

       when eq-result(abs-minus(ie1,ie2),abs-const(c)) = false

end Ind-expr-order
```

```
module Input-restr-sequences
begin
  exports
    begin
      functions
        add-inp-restr: IND-EXPR # IND-EXPR #
                       TECH-SYMTAB # INP-RESTR-SEQ -> INP-RESTR-SEQ
      end

      imports Ind-expr-order, Techn-symtabs,
              Sequences
                  {renamed by
                    [ SEQ     -> INP-RESTR-SEQ,
                      eq-seq  -> eq-inp-restr-seq,
                      null    -> no-restrictions ]
                   Items bound by
                    [ ITEM    -> INP-RESTR,
                      eq-item -> eq-inp-restr ]
                   to Input-restrictions }

   functions
        error-inp-restr-seq:    -> INP-RESTR-SEQ

   variables
        ts              :-> TECH-SYMTAB
        ie1,ie2         :-> IND-EXPR
        irs             :-> INP-RESTR-SEQ

   equations

    [192] add-inp-restr(ie1,ie2,ts,irs)
          = if(eq(order(ie1,ie2),undecided),
               add-item(inp-restr(abs-minus(ie1,ie2),ts),irs),
     if(eq(order(ie1,ie2),less),
                  error-inp-restr-seq,
                  irs))

end Input-restr-sequences


module Range-restrictions
begin
  exports
    begin
      functions
        pos-range-restr: RANGE # TECH-SYMTAB       -> INP-RESTR-SEQ
        pos-range-restr: RANGE-SEQ # TECH-SYMTAB  -> INP-RESTR-SEQ

        range-within-range: RANGE # RANGE # TECH-SYMTAB -> INP-RESTR-SEQ
        range-within-range: RANGE-SEQ # RANGE-SEQ
                            # TECH-SYMTAB               -> INP-RESTR-SEQ
      end

   imports Input-restr-sequences, Ranges,
           Range-sequences, Techn-symtabs
```

```
   variables
        ts                      :-> TECH-SYMTAB
        ie1,ie2,ie3,ie4         :-> IND-EXPR
        rs,rs1,rs2              :-> RANGE-SEQ
        r,r1,r2                 :-> RANGE

   equations

   [193] pos-range-restr(range(ie1,ie2),ts)
         = add-inp-restr(ie1,abs-const(1),ts,
               add-inp-restr(ie2,ie1,ts,no-restrictions))

   [194] range-within-range(range(ie1,ie2),range(ie3,ie4),ts)
         = add-inp-restr(ie1,ie3,ts,
             add-inp-restr(ie4,ie2,ts,
               add-inp-restr(ie2,ie1,ts,no-restrictions)))

        -- a sequence of positive restrictions --

   [195] pos-range-restr(add-item(r,rs),ts)
         = conc(pos-range-restr(r,ts),
                pos-range-restr(rs,ts))

   [196] pos-range-restr(add-item(r,null-range),ts)
          = pos-range-restr(r,ts)

         -- a sequence of range within range restrictions --

   [197] range-within-range(add-item(r1,rs1),add-item(r2,rs2),ts)
              = conc(range-within-range(r1,r2,ts),
                     range-within-range(rs1,rs2,ts))

           when eq(n-of-rngs(rs1),n-of-rngs(rs2)) = true

        -- the error case is left unspecified --

   [198] range-within-range(add-item(r1,null-range),
                            add-item(r2,null-range),ts)
              = range-within-range(r1,r2,ts)

end Range-restrictions
```

2. RANGE CALCULATION FOR INDEX EXPRESSIONS.

2. Global description.

For each index variable a range is stored in the range table. These ranges can be used to calculate symbolic ranges for index expressions. Ranges can only be calculated for symbolic index expressions if the index expressions are monotone increasing or decreasing, as specified in module *Monotone-restrictions*. The specification of monotony amounts to the construction of two lists of variables. One list contains the variables in the negative terms of the expression, the other the variables in the positive terms. If these two list are disjoint the index expression is monotone. The calculation of ranges for index expressions leads to input restrictions as specified in module *Range-calc-restr*. These input restrictions assure that all ranges involved in the calculations are positive.

2.b. Specification.

```
module Monotone-restrictions
begin
  exports
    begin
     functions
      is-monotone: IND-EXPR                            -> BOOL
      var-list:    BOOL # BOOL # IND-EXPR # ID-SEQ  -> ID-SEQ
    end

   imports Ind-expr-abstr-syntax, Id-sequences,
           Booleans
              {renamed by [true  -> pos, false -> neg] }

   variables
        ie,ie1,ie2      :-> IND-EXPR
        idl             :-> ID-SEQ
        id              :-> ID
        sgn1,sgn2       :-> BOOL
        c               :-> INDEX

   equations

  [199] var-list(sgn1,sgn2,abs-plus(ie1,ie2),idl)
          = var-list(sgn1,sgn2,ie1,
                    var-list(sgn1,sgn2,ie2,idl))

  [200] var-list(sgn1,sgn2,abs-mul(ie1,ie2),idl)
          = var-list(sgn1,sgn2,ie1,
                    var-list(sgn1,sgn2,ie2,idl))

  [201] var-list(sgn1,sgn2,abs-minus(ie1,ie2),idl)
          = var-list(sgn1,sgn2,ie1,
                    var-list(sgn1,not(sgn2),ie2,idl))

  [202] var-list(sgn1,sgn2,abs-ind(id),idl)
          = if(eq(sgn1,sgn2),add-item(id,idl),idl)

  [203] var-list(sgn1,sgn2,abs-const(c),idl)
          = idl

          when ge(c,0) = true

      -- the error case is left unspecified --

      -- monotone restricition --

  [204] is-monotone(ie)
             = disjoint(var-list(pos,pos,ie,null-id-seq),
                        var-list(neg,pos,ie,null-id-seq))

end Monotone-restrictions
```

```
module Range-tables
begin
     imports Tables
        { renamed by
            [ TABLE              -> RANGE-TABLE ]
          Entries bound by
            [ ENTRY              -> RANGE,
              eq-entry           -> eq-range ]
            to Ranges
          Addresses bound by
            [ ADDRESS            -> ID,
              eq-addr            -> eq-id]
            to Identifiers
        }
end Range-tables

module Range-calculations
begin
  exports
    begin
     functions
     calc-range:    IND-EXPR # RANGE-TABLE     -> RANGE
     calc-ranges:   IND-EXPR-SEQ # RANGE-TABLE -> RANGE-SEQ

     range-calc-range: RANGE # RANGE-TABLE      -> RANGE
     range-calc-ranges:RANGE-SEQ # RANGE-TABLE  -> RANGE-SEQ
    end

   imports Ind-expr-sequences, Range-sequences,
           Range-tables, Monotone-restrictions

   variables
     ie,ie1,ie2        :-> IND-EXPR
     ies               :-> IND-EXPR-SEQ
     id                :-> ID
     c                 :-> INDEX
     rt                :-> RANGE-TABLE
     r                 :-> RANGE
     rs                :-> RANGE-SEQ

    equations

   [205] calc-range(abs-plus(ie1,ie2),rt)
           = range-plus(calc-range(ie1,rt),
                        calc-range(ie2,rt))

           when is-monotone(abs-plus(ie1,ie2)) = true

        -- the error case is left unspecified --

   [206] calc-range(abs-minus(ie1,ie2),rt)
           = range-min(calc-range(ie1,rt),
                       calc-range(ie2,rt))

           when is-monotone(abs-minus(ie1,ie2)) = true

       -- the error case is left unspecified --
    [207] calc-range(abs-mul(ie1,ie2),rt)
             = range-mul(calc-range(ie1,rt),
                         calc-range(ie2,rt))

           when is-monotone(abs-mul(ie1,ie2)) = true
       -- the error case is left unspecified --

    [208] calc-range(abs-ind(id),rt)  = id^rt

    [209] calc-range(abs-const(c),rt)
             = range(abs-const(c),abs-const(c))

  -- calculations of ranges for a sequence of index
     expressions --

    [210] calc-ranges(add-item(ie,ies),rt)
            = add-item(calc-range(ie,rt),
                       calc-ranges(ies,rt))

    [211] calc-ranges(add-item(ie,null-ind-expr-seq),rt)
            = add-item(calc-range(ie,rt),null-range)

     -- calculation of a range of a range --

    [212] range-calc-range(range(ie1,ie2),rt)
            = range(lower(calc-range(ie1,rt)),
                    upper(calc-range(ie2,rt)))

          when is-monotone(abs-plus(ie1,ie2)) = true

        -- the error case is left unspecified --

     -- calculations of ranges for a sequence of ranges --

    [213] range-calc-ranges(add-item(r,rs),rt)
            = add-item(range-calc-range(r,rt),
                   range-calc-ranges(rs,rt))

    [214] range-calc-ranges(add-item(r,null-range),rt)
            = add-item(range-calc-range(r,rt),null-range)

  end Range-calculations


  module Range-calc-restr
  begin
    exports
     begin
      functions
       restr-calc-range:    IND-EXPR #  RANGE-TABLE #
                            TECH-SYMTAB # INP-RESTR-SEQ  -> INP-RESTR-SEQ
       restr-calc-ranges:   IND-EXPR-SEQ #  RANGE-TABLE #
                            TECH-SYMTAB # INP-RESTR-SEQ  -> INP-RESTR-SEQ

       restr-range-calc-range: RANGE #  RANGE-TABLE #
                               TECH-SYMTAB # INP-RESTR-SEQ -> INP-RESTR-SEQ
```

```
   restr-range-calc-ranges: RANGE-SEQ #  RANGE-TABLE #
                            TECH-SYMTAB # INP-RESTR-SEQ -> INP-RESTR-SEQ
  end

imports Ind-expr-sequences, Input-restr-sequences,
        Range-tables, Range-calculations, Range-restrictions

variables
    ie,ie1,ie2       :-> IND-EXPR
    ies              :-> IND-EXPR-SEQ
    id               :-> ID
    c                :-> INDEX
    irs              :-> INP-RESTR-SEQ
    rt               :-> RANGE-TABLE
    r                :-> RANGE
    rs               :-> RANGE-SEQ
    ts               :-> TECH-SYMTAB

equations

  [215] restr-calc-range(abs-plus(ie1,ie2),rt,ts,irs)
          = restr-calc-range(ie2,rt,ts,
                 restr-calc-range(ie1,rt,ts,irs))

  [216] restr-calc-range(abs-minus(ie1,ie2),rt,ts,irs)
          = conc(
                 pos-range-restr(calc-range(abs-minus(ie1,ie2),rt),ts),
                 restr-calc-range(ie2,rt,ts,
                                  restr-calc-range(ie1,rt,ts,irs)))

  [217] restr-calc-range(abs-mul(ie1,ie2),rt,ts,irs)
          = restr-calc-range(ie2,rt,ts,
                 restr-calc-range(ie1,rt,ts,irs))

  [218] restr-calc-range(abs-ind(id),rt,ts,irs)
          = conc(pos-range-restr(id^rt,ts),irs)

  [219] restr-calc-range(abs-const(c),rt,ts,irs)
          = add-inp-restr(abs-const(c),abs-const(0),ts,irs)

   -- calculations of restrictions for a sequence of index
      expressions --

  [220] restr-calc-ranges(add-item(ie,ies),rt,ts,irs)
          = conc(restr-calc-range(ie,rt,ts,irs),
                 restr-calc-ranges(ies,rt,ts,irs))

  [221] restr-calc-ranges(add-item(ie,null-ind-expr-seq),rt,ts,irs)
          = restr-calc-range(ie,rt,ts,irs)

    -- calculation of a range of a range --

  [222] restr-range-calc-range(range(ie1,ie2),rt,ts,irs)
          = conc(restr-calc-range(ie2,rt,ts,
                        restr-calc-range(ie1,rt,ts,irs)),
                 pos-range-restr(range(ie1,ie2),ts))
```

```
-- calculations of restrictions for a sequence of ranges--

  [223]  restr-range-calc-ranges(add-item(r,rs),rt,ts,irs)
           = conc(restr-range-calc-range(r,rt,ts,irs),
                  restr-range-calc-ranges(rs,rt,ts,irs))

  [224]  restr-range-calc-ranges(add-item(r,null-range),rt,ts,irs)
           = restr-range-calc-range(r,rt,ts,irs)

end Range-calc-restr
```

3. RANGE ASSIGNMENTS FOR INDEX VARIABLES.

3.a. Global description.

Symbolic ranges are assigned to index variables in index assignment statements for statements and index input declarations. The specification is given in module *Range-assignments*. The range assignments are stored in the range table specified in module *Range-tables*.

3.b. Specification.

```
module Range-assignments
begin
  exports
    begin
      functions
        assgn-rngs: STATEMENTS # RANGE-TABLE -> RANGE-TABLE
        assgn-rng:  STATEMENT  # RANGE-TABLE -> RANGE-TABLE
        assgn-rng:  ID # RANGE-TABLE         -> RANGE-TABLE
        assgn-rngs: ID-SEQ # RANGE-TABLE     -> RANGE-TABLE
    end

  imports Statements-abstr-syntax, Range-tables,
          Range-calculations, Id-sequences

  variables
    stmts          :-> STATEMENTS
    stmt           :-> STATEMENT
    var            :-> VARIABLE
    expr           :-> EXPR
    str            :-> STRING
    id             :-> ID
    ie,ie1,ie2     :-> IND-EXPR
    rt             :-> RANGE-TABLE
   equations

  [225] assgn-rngs(abs-statmts(stmts,stmt),rt)
          = assgn-rng(stmt,assgn-rngs(stmts,rt))

  [226] assgn-rngs(abs-statmts(stmt),rt)
          = assgn-rng(stmt,rt)

    -- assigment for individual statements --

  [227] assgn-rng(abs-assgn(var,expr),rt) = rt

  [228] assgn-rng(abs-messtmt(str),rt)    = rt

  [229] assgn-rng(abs-for(id,ie1,ie2,stmts),rt)
          = assgn-range(stmts,
                        insert(id,
                               range-calc-range(range(ie1,ie2),
                               rt)))

  [230] assgn-rng(abs-compound(stmts),rt)
          = assgn-rngs(stmts,rt)

  [231] assgn-rng(abs-ind-assgn(id,ie),rt)
          = insert(id,calc-range(ie,rt),rt)

    -- input declaration --

  [232] assgn-rng(id,rt) = insert(id,range(abs-ind(id),abs-ind(id)),rt)

end Range-assignments
```

4. GENERATION OF INPUT RESTRICTIONS FOR THE BASIC ELEMENTS OF THE STATISTICAL PROGRAM.

4.a. Global description.

In this section it is shown how input restrictions are generated for variables, expressions and statements in the statistical language. The input restrictions generated for matrix element references and consist of two groups of restrictions:
- the restrictions generated during the calculation of symbolic ranges for index expressions used as matrix element or submatrix reference.
- the dimension bound restrictions, checked with the use of the calculated symbolic ranges

4.b. Specification.

```
module Gen-restr-variables
begin
exports
  begin
    functions
      gen-restr-var: INP-RESTR-SEQ # RANGE-TABLE #
                     TECH-SYMTAB # VARIABLE -> INP-RESTR-SEQ
  end
 imports Variable-abstr-syntax, Input-restr-sequences,
         Range-tables, Techn-symtabs, Range-restrictions,
         Range-calculations, Range-calc-restr

 variables
       irs                  :-> INP-RESTR-SEQ
       id                   :-> ID
       inds                 :-> IND-EXPR-SEQ
       rngs,rngs1,rngs2     :-> RANGE-SEQ
       rt                   :-> RANGE-TABLE
       ts                   :-> TECH-SYMTAB
```

```
equations
             -- simple variables --

  [233] gen-restr-var(irs,rt,ts,abs-var(id))  = irs

              -- matrix element --

  [234] gen-restr-var(irs,rt,ts,abs-var(id,inds))
      = conc(restr-calc-ranges(inds,rt,ts,irs),
             range-within-range(calc-ranges(inds,rt),
                                rngs,ts))

        when eq-type(type(id^ts),matrix-type(rngs)) = true

              -- submatrix --

  [235] gen-restr-var(irs,rt,ts,abs-var(id,rngs1))
          = conc(restr-range-calc-ranges(rngs1,rt,ts,irs),
                 range-within-range(
                      range-calc-ranges(rngs1,rt),rngs2,ts))

        when eq-type(type(id^ts),matrix-type(rngs2)) = true

end Gen-restr-variables


module Gen-restr-expressions
begin
  exports
   begin
     functions
     gen-restr-expr: INP-RESTR-SEQ # RANGE-TABLE #
                     TECH-SYMTAB # EXPR          -> INP-RESTR-SEQ
     gen-restr-argl: INP-RESTR-SEQ # RANGE-TABLE #
                     TECH-SYMTAB # ARG-LIST      -> INP-RESTR-SEQ
   end

  imports Expr-abstr-syntax, Gen-restr-variables

  variables
          irs                   :-> INP-RESTR-SEQ
          ts                    :-> TECH-SYMTAB
          exp,exp1,exp2         :-> EXPR
          var                   :-> VARIABLE
          td                    :-> TECH-DATA
          f-id                  :-> ID
          argl                  :-> ARG-LIST
          rt                    :-> RANGE-TABLE

  equations

  [236] gen-restr-expr(irs,rt,ts,abs-expr(var))
          = gen-restr-var(irs,rt,ts,var)

  [237] gen-restr-expr(irs,rt,ts,abs-expr(td))
          = irs
```

```
          -- function calls --

  [238] gen-restr-expr(irs,rt,ts,abs-f-call(f-id,argl))
          = gen-restr-argl(irs,rt,ts,argl)

  [239] gen-restr-argl(irs,rt,ts,abs-arg-l(argl,exp))
          = gen-restr-expr(
               gen-restr-argl(irs,rt,ts,argl),rt,ts,exp)

  [240] gen-restr-argl(irs,rt,ts,abs-arg-l(exp))
          = gen-restr-expr(irs,rt,ts,exp)

end Gen-restr-expressions


module Input-restr-info
begin
  exports
   begin
    sorts INP-RESTR-INFO
    functions
     inp-res-info: INP-RESTR-SEQ # RANGE-TABLE -> INP-RESTR-INFO
   end

  imports Range-tables, Input-restr-sequences

end Input-restr-info


module Gen-restr-statements
begin
  exports
    begin
      functions
        gen-restr-stmts: INP-RESTR-INFO #  TECH-SYMTAB #
                         STATEMENTS    -> INP-RESTR-INFO
        gen-restr-stm:   INP-RESTR-INFO #  TECH-SYMTAB #
                         STATEMENT    ->  INP-RESTR-INFO
     end

  imports Gen-restr-expressions, Gen-restr-variables,
          Statements-abstr-syntax, Input-restr-info,
          Range-assignments

  variables
        iri             :-> INP-RESTR-INFO
        irs             :-> INP-RESTR-SEQ
        stmts           :-> STATEMENTS
        stm             :-> STATEMENT
        var             :-> VARIABLE
        exp             :-> EXPR
        m               :-> STRING
        rt              :-> RANGE-TABLE
        tst             :-> TECH-SYMTAB
        id              :-> ID
        ie,ie1,ie2      :-> IND-EXPR
  equations

  [241] gen-restr-stmts(iri,tst,abs-statmts(stmts,stm))
        = gen-restr-stm(
               gen-restr-stmts(iri,tst,stmts),tst,stm)

  [242] gen-restr-stmts(iri,tst,abs-statmts(stm))
        = gen-restr-stm(iri,tst,stm)

        -- assignment statement --

  [243] gen-restr-stm(inp-res-info(irs,rt),
                  tst,abs-assgn(var,exp))
        =inp-res-info(gen-restr-var(gen-restr-expr(irs,rt,tst,exp),
                                    rt,tst,var),
                   rt)

   -- message statement --

  [244] gen-restr-stm(iri,tst,abs-messtmt(m))
        = iri

   -- compound statement --

  [245] gen-restr-stm(iri,tst,abs-compound(stmts))
        = gen-restr-stmts(iri,tst,stmts)

   -- for statement --

  [246] gen-restr-stm(inp-res-info(irs,rt),
                   tst,abs-for(id,ie1,ie2,stmts))
          = gen-restr-stmts(
                inp-res-info(
                   restr-range-calc-range(range(ie1,ie2),
                                 rt,tst,irs),
                   assgn-rng(abs-for(id,ie1,ie2,stmts),rt)),
                tst,stmts)

    -- index assignment statement --

   [247] gen-restr-stm(inp-res-info(irs,rt),
                   tst,abs-ind-assgn(id,ie))
          = inp-res-info(restr-calc-range(ie,rt,tst,irs),
                     assgn-rng(abs-ind-assgn(id,ie),rt))

end Gen-restr-statements
```

5. GENERATION OF INPUT RESTRICTIONS FOR SECTIONS OF A STATISTICAL PROGRAM.

5.a. Global description.

In this section it is shown how restrictions are generated for the sections in a statistical program.

5.b. Specification.

```
module Gen-restr-declarations
begin
  exports
    begin
     functions
       gen-restr-io-sect:    INP-RESTR-INFO # TECH-SYMTAB #
                             IO-SEC               -> INP-RESTR-INFO

       gen-restr-int-decls:  INP-RESTR-INFO # TECH-SYMTAB #
                             INTERN-DECLS         -> INP-RESTR-INFO

       gen-restr-decls:      INP-RESTR-INFO # TECH-SYMTAB #
                             DECLS # USER-VIEW    -> INP-RESTR-INFO

       gen-restr-decl:       INP-RESTR-INFO # TECH-SYMTAB #
                             DECL # USER-VIEW     -> INP-RESTR-INFO
    end

  imports Input-restr-info, Decl-abstr-syntax,
          Range-restrictions, Range-assignments

  variables
     ts                             :-> TECH-SYMTAB
     iri                            :-> INP-RESTR-INFO
     rt                             :-> RANGE-TABLE
     irs                            :-> INP-RESTR-SEQ
     decls,inp-decls,out-decls      :-> DECLS
     decl                           :-> DECL
     idl                            :-> ID-SEQ
     rngs                           :-> RANGE-SEQ
     mess                           :-> STRING
     st                             :-> SIMPLE-TYPE
     uv                             :-> USER-VIEW

  equations

  [248] gen-restr-io-sect(iri,ts,
                       abs-io-sect(inp-decls,out-decls))
        = gen-restr-decls(
                 gen-restr-decls(iri,ts,inp-decls,input),
                 ts,out-decls,output)

 [249] gen-restr-int-decls(iri,ts,abs-intern-decl(decls))
       = gen-restr-decls(iri,ts,decls,internal)

 [250] gen-restr-decls(iri,ts,abs-empty-decls,uv)
       = iri

 [251] gen-restr-decls(iri,ts,abs-decls(decls,decl),uv)
       = gen-restr-decl(
               gen-restr-decls(iri,ts,decls,uv),
               ts,decl,uv)

 [252] gen-restr-decls(iri,ts,abs-decls(decl),uv)
       = gen-restr-decl(iri,ts,decl,uv)
 [253] gen-restr-decl(inp-res-info(irs,rt),ts,
                      abs-decl(idl,matrix-type(rngs),mess),uv)
        = inp-res-info(conc(irs,pos-range-restr(rngs,ts)),
                       rt)

 [254] gen-restr-decl(inp-res-info(irs,rt),ts,
                      abs-decl(idl,tech-type(index-type),mess),
                      input)
        = inp-res-info(irs,assgn-rngs(idl,rt))

end Gen-restr-declarations


module Gen-restr-impl
begin
  exports
    begin
      functions
        gen-restr-impl-s: IMPL-SEC # TECH-SYMTAB #
                INP-RESTR-INFO -> INP-RESTR-INFO
     nd

  imports Gen-restr-statements, Gen-restr-declarations,
          Impl-abstr-syntax

  variables
   decl             :-> INTERN-DECLS
   stmnts           :-> STATEMENTS
   iri              :-> INP-RESTR-INFO
   ts               :-> TECH-SYMTAB

  equations

  [255] gen-restr-impl-s(abs-impl-sec(decl,stmnts),ts,iri)
         = gen-restr-stmts(
               gen-restr-int-decls(iri,ts,decl),ts,stmnts)

end Gen-restr-impl


module Gen-restr-tests
begin
  exports
    begin
      functions
        gen-restr-test-s: TEST-SEC # TECH-SYMTAB #
                          INP-RESTR-INFO -> INP-RESTR-INFO
    end

  imports Gen-restr-statements, Gen-restr-declarations,
          Test-abstr-syntax

  variables
   decl             :-> INTERN-DECLS
   stmnts           :-> STATEMENTS
   rss              :-> RAISES
   iri              :-> INP-RESTR-INFO
   ts               :-> TECH-SYMTAB

  equations

  [256] gen-restr-test-s(abs-test-sec(decl,stmnts,rss),ts,iri)
         = gen-restr-stmts(
               gen-restr-int-decls(iri,ts,decl),ts,stmnts)

end Gen-restr-tests


module Gen-restr-handlers
begin
  exports
    begin
      functions
        gen-restr-handl-s: HANDL-SEC # TECH-SYMTAB #
                           INP-RESTR-INFO -> INP-RESTR-INFO

        gen-restr-handler: HANDLER # TECH-SYMTAB #
                           INP-RESTR-INFO -> INP-RESTR-INFO
      end

  imports Gen-restr-statements, Gen-restr-declarations,
          Handler-abstr-syntax

  variables
   h-sec            :-> HANDL-SEC
   hndl             :-> HANDLER
   decl             :-> INTERN-DECLS
   stmnts           :-> STATEMENTS
   iri              :-> INP-RESTR-INFO
   id               :-> ID
   ts               :-> TECH-SYMTAB

  equations

  [257] gen-restr-handl-s(abs-handl-sec(h-sec,hndl),ts,iri)
         = gen-restr-handl-s(abs-handl-sec(hndl),ts,
               gen-restr-handl-s(h-sec,ts,iri))

  [258] gen-restr-handl-s(abs-handl-sec(hndl),ts,iri)
         = gen-restr-handler(hndl,ts,iri)

  [259] gen-restr-handler(abs-handler(id,decl,stmnts),ts,iri)
         = gen-restr-stmts(
               gen-restr-int-decls(iri,ts,decl),ts,stmnts)

end Gen-restr-handlers
```

6. THE GENERATION OF INPUT RESTRICTIONS FOR A STATISTICAL PROGRAM.

6.a. Global description.

In this section it is specified how restrictions are generated for a statistical program.

6.b. Specification.

```
module Input-restr-generator
begin
  exports
    begin
      functions
        gen-restr-sct: SECTION # TECH-SYMTAB #
                       INP-RESTR-INFO # INP-RESTR-SEQ  -> INP-RESTR-INFO
        gen-restr-pro: STAT-PRO # TECH-SYMTAB           -> INP-RESTR-INFO
    end

  imports Statistical-programs, Gen-restr-declarations,
          Gen-restr-handlers, Gen-restr-tests, Gen-restr-impl

  variables
   id        :-> ID
   iri       :-> INP-RESTR-INFO
   io-sec    :-> IO-SEC
   impl-sec  :-> IMPL-SEC
   test-sec  :-> TEST-SEC
   handl-sec :-> HANDL-SEC
   sect      :-> SECTION
   pro       :-> STAT-PRO
   ts        :-> TECH-SYMTAB

  equations
  [260] gen-restr-sct(abs-name(id),ts,iri)
        = iri

  [261] gen-restr-sct(abs-sect(io-sec),ts,iri)
        = gen-restr-io-sect(iri,ts,io-sec)

  [262] gen-restr-sct(abs-sect(impl-sec),ts,iri)
        = gen-restr-impl-s(impl-sec,ts,iri)

  [263] gen-restr-sct(abs-sect(test-sec),ts,iri)
        = gen-restr-test-s(test-sec,ts,iri)

  [264] gen-restr-sct(abs-sect(handl-sec),ts,iri)
        = gen-restr-handl-s(handl-sec,ts,iri)

  [265] gen-restr-pro(abs-prog(pro,sect),ts)
        = gen-restr-sct(sect,ts,gen-restr-pro(pro,ts))

  [266] gen-restr-pro(abs-prog(sect),ts)
        = gen-restr-sct(sect,ts,
              inp-res-info(no-restrictions,null-table))

end Input-restr-generator
```

APPENDIX J. ASF SPECIFICATION OF THE KERNEL

The kernel is a virtual machine that can execute the semantic actions that take place during the execution of a statistical technique. A statistical technique is represented at the kernel level by a kernel program. The specification of the kernel consists of the specification of (1) the evaluation of index expressions, data types and input restrictions, (2) the data memory, (3) the instructions set, (4) the data stack, (5) exception handler tables, (6) the function code table, (7) a kernel program, (8) the processor: execution of the instructions, (9) the kernel.

1. EVALUATION OF INDEX EXPRESSIONS, TYPES AND RESTRICTIONS.

1.a. Global description.

The values of the variables in an index expression are stored in the technique symbol table. The kernel uses these values to evaluate the index expression. The evaluation of a symbolic dimension range requires the evaluation of the two index expressions in this range, and results in a constant range.
The evaluation of a matrix type description in the statistical language requires the evaluation of the symbolic ranges describing the dimension ranges. The evaluation of a matrix type description results in a matrix description with fixed dimensions, as used at the user level.
The evaluation of an input restrictions results in the value *true* or *false*.

1.b. Specification

```
module Evaluate-ind-expr
begin
  exports
    begin
      functions
        evaluate: IND-EXPR # TECH-SYMTAB -> INDEX
    end

  imports Ind-expr-abstr-syntax, Techn-symtabs, Indices

  variables
        iexpr1,iexpr2       :-> IND-EXPR
        ts                  :-> TECH-SYMTAB
        id                  :-> ID
        i                   :-> INDEX
        tt                  :-> TECH-TYPE
        uv                  :-> USER-VIEW
        str                 :-> STRING

  equations

  [267]  evaluate(abs-mul(iexpr1,iexpr2),ts)
           = mul(evaluate(iexpr1,ts),evaluate(iexpr2,ts))

  [268]  evaluate(abs-plus(iexpr1,iexpr2),ts)
           = add(evaluate(iexpr1,ts),evaluate(iexpr2,ts))
  [269]  evaluate(abs-minus(iexpr1,iexpr2),ts)
            = sub(evaluate(iexpr1,ts),evaluate(iexpr2,ts))

  [270]  evaluate(abs-const(i),ts)              = i

  [271]  evaluate(abs-ind(id),insert(id,
            ts-info(tt,uv,t-data(i),str),ts))  = i

        -- the error case is not specified --

end Evaluate-ind-expr


module Evaluate-ranges
begin
  exports
    begin
     functions
        evaluate: RANGE      # TECH-SYMTAB -> CONST-RANGE
        evaluate: RANGE-SEQ # TECH-SYMTAB -> CONST-RANGE-SEQ
    end

  imports  Range-sequences, Const-range-sequences,
           Evaluate-ind-expr

  variables
     iexpr1,iexpr2       :-> IND-EXPR
     ts                  :-> TECH-SYMTAB
     r                   :-> RANGE
     rs                  :-> RANGE-SEQ

  equations

  [272] evaluate(range(iexpr1,iexpr2),ts)
         = c-range(evaluate(iexpr1,ts),evaluate(iexpr2,ts))

  [273] evaluate(add-item(r,rs),ts)
         = add-item(evaluate(r,ts),evaluate(rs,ts))

  [274] evaluate(add-item(r,null-range),ts)
         = add-item(evaluate(r,ts),null-cr-seq)

end Evaluate-ranges


module Evaluate-types
begin
  exports
    begin
      functions
        evaluate: TECH-TYPE  # TECH-SYMTAB -> USER-TYPE
    end
  imports  User-types, Technique-types,
           Evaluate-ranges

  variables
     st          :-> SIMPLE-TYPE
     rngs        :-> RANGE-SEQ
     ts          :-> TECH-SYMTAB

  equations

  [275] evaluate(tech-type(st),ts)        =  user-type(st)

  [276] evaluate(matrix-type(rngs),ts) =  matrix-type(evaluate(rngs,ts))

end Evaluate-types


module Evaluate-restrictions
begin
  exports
    begin
     functions
       check: INP-RESTR # TECH-SYMTAB      -> BOOL
       check: INP-RESTR-SEQ # TECH-SYMTAB  -> BOOL
    end

   imports Evaluate-ind-expr, Input-restr-sequences

   variables
      ie              :-> IND-EXPR
      ts              :-> TECH-SYMTAB
      ir              :-> INP-RESTR
      irs             :-> INP-RESTR-SEQ

   equations

   [277] check(inp-restr(ie,ts),ts) = ge(evaluate(ie,ts),0)
   [278] check(no-restrictions,ts)  = true
   [279] check(add-item(ir,irs),ts) = and(check(ir,ts),check(irs,ts))

end Evaluate-restrictions
```

2. THE DATA MEMORY OF THE KERNEL.

2.a. Global description.

The memory of the kernel consists of two symbol tables: a technique and and a user symbol table. The kernel can transfer data between the technique symbol table and a data stack. How data is retrieved from the technique symbol table is specified in the functions *get-data*, *get-elem* (for matrix elements) and *get-subm* (for submatrices). How the calculated results on the data stack are stored in the technique symbol table in specified by the functions *store-data*, *store-elem* (for matrix elements) and *store-subm* (for submatrices).
The input variables must be intialized by the user of the statistical technique. At the kernel level this is represented by a transfer from data in the user symbol table to the technique symbol table. A special case is the situation where series form the columns of a matrix. This is specified in module *Series-matrix-interface*. No equations are given in this module. The resulting statistics calculated by a statistical

technique are stored in the user symbol table, as specified in module *User-io*.

2.b. Specification.

```
module Memory
begin
   exports
    begin
     sorts MEMORY
      functions
      memory:      TECH-SYMTAB # USER-SYMTAB      -> MEMORY

      get-data:    ID #  MEMORY                   -> TECH-DATA
      get-elem:    ID #  INDEX-SEQ # MEMORY       -> TECH-DATA
      get-subm:    ID #  CONST-RANGE-SEQ # MEMORY -> TECH-DATA

      store-data:  ID #  TECH-DATA # MEMORY       -> MEMORY
      store-elem:  ID #  TECH-DATA #
                   INDEX-SEQ # MEMORY             -> MEMORY
      store-subm:  ID #  TECH-DATA #
                   CONST-RANGE-SEQ # MEMORY       -> MEMORY
    end

   imports Techn-symtabs, User-symtabs,
           Index-sequences, Const-range-sequences

   variables
     id          :-> ID
     ts          :-> TECH-SYMTAB
     us          :-> USER-SYMTAB
     is          :-> INDEX-SEQ
     td          :-> TECH-DATA
     drs         :-> CONST-RANGE-SEQ

   equations

   -- data retrieval --

   [280] get-data(id,memory(ts,us))      = data(id^ts)
    [281] get-elem(id,is,memory(ts,us))   = element(is,data(id^ts))
[282] get-subm(id,drs,memory(ts,us)) = submat(drs,data(id^ts))

    -- data storage --

   [283]  store-data(id,td,memory(ts,us))
            = memory(insert-data(id,td,ts),us)

end Memory
```

```
module Series-matrix-interface
begin
   exports
     begin
      functions
        data-matrix:     ID-SEQ # USER-SYMTAB -> MATRIX
        load-mat-excep:  ID-SEQ # USER-SYMTAB -> BOOL
        mat-excep-id:    ID-SEQ # USER-SYMTAB -> ID
     end

     imports Id-sequences, User-symtabs, Matrices

end Series-matrix-interface


module User-io
begin
   exports
     begin
      functions
        user-insert:      ID # MEMORY -> MEMORY
        user-load-excep:  ID # MEMORY -> BOOL
        load-excep-id:    ID # MEMORY -> ID
        store-result:     ID # MEMORY -> MEMORY
     end

    imports Identifiers, Id-sequences, Evaluate-types,
            Memory, Series-matrix-interface, User-data,
            User-types, Scalars, Indices, Booleans

   functions
     -- possible user initializations --
        user-input-id:       -> ID
        user-input-id-seq:   -> ID-SEQ
        user-input-scalar:   -> SCALAR
        user-input-index:    -> INDEX
        user-input-bool:     -> BOOL

  variables
        id                   :-> ID
        us                   :-> USER-SYMTAB
        ts                   :-> TECH-SYMTAB
        rng1,rng2            :-> RANGE

  equations
        -- intializing input variables --

  [284] user-insert(id,memory(ts,us))
        = memory(insert-data(id,
                             us-data(user-input-scalar),
                             ts),
                 us)

        when eq-type(type(id^ts),tech-type(scalar-type)) = true
```

```
[285] user-insert(id,memory(ts,us))
        = memory(insert-data(id,
                             t-data(user-input-index),
                             ts),
                 us)

      when eq-type(type(id^ts),tech-type(index-type)) = true

[286] user-insert(id,memory(ts,us))
          = memory(insert-data(id,
                               t-data(user-input-bool),
                               ts),
                   us)

      when eq-type(type(id^ts),tech-type(bool-type)) = true

[287] user-insert(id,memory(ts,us))
        = memory(insert-data(id,
                             t-data(data(user-input-id^us)),
                            ts),
                 us)

      when eq(evaluate(type(id^ts),ts),
              type(user-input-id^us)) = true

[288]  user-insert(id,memory(ts,us))
         = memory(insert-data(id,
                              t-data(data-matrix(user-input-id-seq,us)),
                              ts),
                  us)

      when 2-dim-matrix(type(id^ts)) = true

   -- error cases of user insert are left unspecified --

[289] user-load-excep(id,memory(ts,us))
        = if (2-dim-matrix(type(id^ts)),
              load-mat-excep(user-input-id-seq,us),
              false)

[290] load-excep-id(id,memory(ts,us))
         = if (2-dim-matrix(type(id^ts)),
               mat-excep-id(user-input-id-seq,us),
               no-id)

     -- store result calculations --

[291]  store-result(id,memory(ts,us))
         = memory(ts,
                  insert(id,
                         us-info(evaluate(type(id^ts),ts),
                                 u-data(data(id^ts))),
                         us))

end User-io
```

3. THE KERNEL INSTRUCTIONS.

3.1. Global description.

The kernel has instructions to

- transfer data between the user and statistical level,
- pop data from and push data on the data stack,
- evaluate index expressions and check index expression conditions,
- call functions that operate on the data stack,
- alter the sequential control of kernel,
- control the exception handling,
- display a message,
- stop the execution of the kernei.

The abstract syntax of these instructions is specified in module *Instructions*. The semantics of the instructions can be found in module *Processor* in section 8 of this appendix. An address of an instruction is specified as an instruction with an index indicating the position in the sequence.

3.2. Specification.

```
module Instructions
begin
  exports
    begin
      sorts INSTR
      functions
           -- interface with user level --

         user-load:     ID                  -> INSTR
         user-store:    ID                  -> INSTR

                -- load instructions --

         load:          TECH-DATA           -> INSTR
         load:          ID                  -> INSTR

         load-elem:     ID # INDEX          -> INSTR
         load-subm:     ID # INDEX          -> INSTR

                -- store instructions --

         store:         ID                  -> INSTR
         store-elem:    ID # INDEX          -> INSTR
         store-subm:    ID # INDEX          -> INSTR

             -- ind-expression instructions --

         eval-expr:     IND-EXPR            -> INSTR
         check-ge-restr:                    -> INSTR

                -- other instructions --

         jump:          INDEX               -> INSTR
         jump-true:     INDEX               -> INSTR
```

```
        jump-false:     INDEX                   -> INSTR

        increm:         ID                      -> INSTR
        fcall:          ID # INDEX              -> INSTR
        display:        STRING                  -> INSTR
        raise:          ID                      -> INSTR
        unraise:                                -> INSTR
        halt:                                   -> INSTR

        eq:             INSTR # INSTR           -> BOOL
      end

    imports Technique-data, Identifiers,
            Ind-expr-abstr-syntax, Strings

end Instructions


module Instruction-sequences
begin
    exports
      begin
        sorts INSTR-ADDR
        functions
         addr:      INSTR-SEQ # INDEX           -> INSTR-ADDR
         next:      INSTR-ADDR                  -> INSTR-ADDR
         null-addr:                             -> INSTR-ADDR
         fetch:     INSTR-ADDR                  -> INSTR
         halt-sequence:                         -> INSTR-SEQ
         eq-addr: INSTR-ADDR # INSTR-ADDR       -> BOOL
      end

    imports Sequences
            { renamed by
               [ SEQ             -> INSTR-SEQ,
                 null            -> null-instr-seq,
                 eq-seq          -> eq-instr-seq ]
              Items bound by
               [ ITEM            -> INSTR,
                 eq-item         -> eq ]
                to Instructions
             }

    variables
      is,is1,is2  :-> INSTR-SEQ
      i,i1,i2     :-> INDEX

    equations

    [292] next(addr(is,i))               = addr(is,increm(i))
    [293] fetch(addr(is,i))              = item-no(is,i)
    [294] add-item(halt,null-instr-seq) = halt-sequence

    [295] eq-addr(addr(is1,i1),addr(is2,i2))
           = and (eq-instr-seq(is1,is2), eq(i1,i2))

end Instruction-sequences
```

4. THE DATA STACK

4.1. Global description.

The kernel evaluates an expression on a data stack. The functions *index-s* and *range-s* are special functions that can retrieve, respectivily, a sequence of indices (matrix element reference) or a sequence of constant ranges (submatrix reference) from the data stack.

4.2. Specification.

```
module Data-stacks
begin
   exports
     begin
       functions
         index-s: DATA-STACK -> INDEX-SEQ
         range-s: DATA-STACK -> CONST-RANGE-SEQ
     end

   imports Index-sequences, Const-range-sequences,
           Sequences
              { renamed by
                  [ SEQ             -> DATA-STACK,
                    add-item        -> push,
                    del-item        -> pop,
                    last            -> top,
                    null            -> empty-stack ]
                 Items bound by
                  [ ITEM            -> TECH-DATA,
                    eq-item         -> eq ]
                 to Technique-data
              }

   variables
     i,i1,i2  :-> INDEX
     is       :-> INDEX-SEQ
     rs       :-> CONST-RANGE-SEQ
     rst      :-> DATA-STACK

   equations

   [296] index-s(push(t-data(i),rst)) = add-item(i,index-s(rst))
   [297] index-s(empty-stack)         = null-ind-seq

   [298]  range-s(push(t-data(i1),push(t-data(i2),rst)))
            = add-item(c-range(i1,i2),range-s(rst))

   [299]  range-s(empty-stack)          = null-cr-seq

end Data-stacks
```

5. HANDLER TABLES.

5.a. Global descpription.

Instruction sequences of exception handlers declared in a statistical program are stored in a technique handler table. External exception handlers consists of a technique symbol table and instructions sequence. The external handlers are stored in the external handler table. Both the handlers stored in both the technique and the external handler table are available during the execution of a statistical technique.

5.b. Specification.

```
module Tech-handler-tables
begin
     imports
          Tables
        { renamed by
            [ TABLE             -> TECH-HANDLER-TABLE,
              null-table        -> empty-tech-hand-tab ]
          Entries bound by
            [ ENTRY             -> INSTR-SEQ,
              eq-entry          -> eq-instr-seq ]
            to Instruction-sequences
          Addresses bound by
            [ ADDRESS           -> ID,
              eq-addr           -> eq-id ]
            to Identifiers
        }
end Tech-handler-tables


module External-handlers
begin
  exports
    begin
      sorts EXT-HANDL
      functions
       ext-handl:    INSTR-SEQ # TECH-SYMTAB  -> EXT-HANDL
       han-symtab:   EXT-HANDL                -> TECH-SYMTAB
       han-instr-s:  EXT-HANDL                -> INSTR-SEQ
       eq-ext-handl: EXT-HANDL # EXT-HANDL    -> BOOL
    end

   imports Techn-symtabs, Instruction-sequences

end External-handlers


module Ext-handler-tables
begin
    imports Tables
       { renamed by
           [ TABLE             -> EXT-HANDLER-TABLE,
             null-table        -> empty-ext-hand-tab ]
         Entries bound by
           [ ENTRY             -> EXT-HANDL,
             eq-entry          -> eq-ext-handl ]
         to External-handlers
         Addresses bound by
           [ ADDRESS           -> ID,
             eq-addr           -> eq ]
         to Identifiers
         }
end Ext-handler-tables


module Handler-tables
begin
   exports
     begin
       sorts HANDLER-TABLES
       functions
         handlers:   TECH-HANDLER-TABLE # EXT-HANDLER-TABLE
                          -> HANDLER-TABLES
     end

   imports Tech-handler-tables, Ext-handler-tables

end Handler-tables
```

6. FUNCTION CODE.

6.a. Global description.

The functions in the statistical language are left unspecified. Only an abstract notion of such a function is defined. Given the current values on the data stack a function returns a value. Also such a function may return an exception identifier. The fact that the function table still has to be intialized is emphasized by a parameter *Current-function-code* in the module *Func-code-table*.

6.b. Specification.

```
module Function-code
begin
   exports
     begin
       sorts  FUNC-CODE
        functions
         execute:       FUNC-CODE # DATA-STACK -> TECH-DATA
         excep-f:       FUNC-CODE # DATA-STACK -> ID
         excep-raised:  FUNC-CODE # DATA-STACK -> BOOL
         eq-f-code:     FUNC-CODE # FUNC-CODE  -> BOOL
     end

   imports  Data-stacks , Identifiers

end Function-code
```

```
module Func-code-tables
begin
     parameters Current-func-code
       begin
        functions
           ini-fcode-tab:                    -> FUNC-CODE-TABLE
       end Current-func-code
     exports
       begin
        functions
           execute-f:    ID # DATA-STACK -> TECH-DATA
           exception-f:  ID # DATA-STACK -> ID
           excep-raised: ID # DATA-STACK -> BOOL
       end
     imports Tables
        { renamed by
            [ TABLE              -> FUNC-CODE-TABLE ]
          Entries bound by
            [ ENTRY              -> FUNC-CODE,
              eq-entry           -> eq-f-code ]
            to Function-code
          Addresses bound by
            [ ADDRESS            -> ID,
              eq-addr            -> eq-id ]
            to Identifiers
        }

     variables
         id    :-> ID
         ds    :-> DATA-STACK

     equations

     [300] execute-f(id,ds)    = execute(id^ini-fcode-tab,ds)
     [301] exception-f(id,ds) = excep-f(id^ini-fcode-tab,ds)
     [302] excep-raised(id,ds)
           = excep-raised(id^ini-fcode-tab,ds)

end Func-code-tables
```

7. KERNEL PROGRAMS.

7.a. Global description.

A statistical program is represented at the kernel level by a kernel program. A kernel program consist of an instruction sequence an input restriction sequence, a technique symbol table and a technique handler table. The collection of all available statistical techniques at the kernel level is stored in the statistical technique table.

7.b. Specification.

```
module Kernel-programs
begin
   exports
     begin
       sorts KERNEL-PRO
        functions
        kern-pro: INSTR-SEQ #
                  INP-RESTR-SEQ #
                  TECH-SYMTAB #
                  TECH-HANDLER-TABLE -> KERNEL-PRO

        eq-pro:   KERNEL-PRO # KERNEL-PRO -> BOOL
     end

   imports Techn-symtabs, Instruction-sequences,
           Tech-handler-tables, Input-restr-sequences

end Kernel-programs

module Stat-technique-tables
begin
     imports Tables
        { renamed by
            [ TABLE              -> STAT-TECH-TABLE,
              null-table         -> empty-stat-tech-tab ]
          Entries bound by
            [ ENTRY              -> KERNEL-PRO,
              eq-entry           -> eq-pro ]
            to Kernel-programs
          Addresses bound by
            [ ADDRESS            -> ID,
              eq-addr            -> eq ]
            to Identifiers
        }
end Stat-technique-tables
```

8. EXECUTION.

8.a. Global description.

The kernel executes sequences of instructions until a halt instruction is reached. Each instruction modifies the state of the kernel. The instructions for exception handling are treated in a separate module. The state of the kernel is determined by
- the data memrory,
- the data stack,
- the current instruction,
- the address of the next instruction,
- the reset information for an unraise instruction,
- the exception handler tables,
- the input restriction sequence,
- a user display.

8.b. Specification

```
module Kernel-states
begin
   exports
      begin
        sorts STATE, RESET-INFO
         functions
         reset-info: INSTR-ADDR #  TECH-SYMTAB  -> RESET-INFO
         reset-info: INSTR-ADDR                 -> RESET-INFO
         no-reset-info:                         -> RESET-INFO

         state:  MEMORY #
                 DATA-STACK #

                 INSTR #         -- current instruction --
                 INSTR-ADDR #    -- address next instruction --

                 RESET-INFO #
                 HANDLER-TABLES #

                 INP-RESTR-SEQ #
                 STRING          -- display --
                                     -> STATE

            stop:                    -> STATE
            result:   STATE          -> USER-SYMTAB
      end

   imports Func-code-tables, Memory, Data-stacks,
           Handler-tables, Instructions, Instruction-sequences,
           Input-restr-sequences, Strings

   variables
        m        :-> MEMORY
        us       :-> USER-SYMTAB
        ts       :-> TECH-SYMTAB
        ds       :-> DATA-STACK
        ia       :-> INSTR-ADDR
        ri       :-> RESET-INFO
        hts      :-> HANDLER-TABLES
        irs      :-> INP-RESTR-SEQ
        dis      :-> STRING
        i        :-> INSTR

   equations

   [303] state(m,ds,halt,ia,ri,hts,irs,dis)                   = stop
   [304] result(state(memory(ts,us),ds,i,ia,ri,hts,irs,dis)) = us

end Kernel-states
```

```
module Exception-handling
begin
   exports
      begin
        functions
          handle: STATE          -> STATE
          reset:  STATE          -> STATE
      end

   imports Kernel-states

   variables
        m             :-> MEMORY
        us            :-> USER-SYMTAB
        ts,ts1,ts2    :-> TECH-SYMTAB
        ds            :-> DATA-STACK
        ia,ia1,ia2    :-> INSTR-ADDR
        ri            :-> RESET-INFO
        h-id          :-> ID
        tht           :-> TECH-HANDLER-TABLE
        eht           :-> EXT-HANDLER-TABLE
        hts           :-> HANDLER-TABLES
        irs           :-> INP-RESTR-SEQ
        dis           :-> STRING

    equations

    [305] handle(state(memory(ts,us),ds,raise(h-id),ia,
                    no-reset-info,handlers(tht,eht),irs,dis))
          = if (found(h-id,tht),
                 state(memory(ts,us),ds,
                      first(h-id^tht),addr(h-id^tht,1),
                      reset-info(ia),handlers(tht,eht),
                      irs,dis),
                 if(found(h-id,eht),
                    state(memory(han-symtab(h-id^eht),us),ds,
                         first(han-instr-s(h-id^eht)),
                         addr(han-instr-s(h-id^eht),1),
                         reset-info(ia,ts),handlers(tht,eht),
                irs,dis),
                    stop))

     [306] handle(state(memory(ts,us),ds,raise(h-id),ia,
                     reset-info(ia),handlers(tht,eht),
                     irs,dis))
           = stop

     [307] handle(state(memory(ts1,us),ds,raise(h-id),ia1,
                      reset-info(ia2,ts2),handlers(tht,eht),
                      irs,dis))
           = stop

     [308] handle(state(m,ds,raise(h-id),ia1,
                     reset-info(ia2),hts,irs,dis))
           = stop
    [309] handle(state(m,ds,raise(h-id),ial,
                       reset-info(ia2,ts),hts,irs,dis))
               = stop

    [310] reset(state(m,ds,unraise,ial,
                      reset-info(ia2),hts,irs,dis))
               = state(m,ds,fetch(ia2),next(ia2),
                       no-reset-info,hts,irs,dis)

    [311] reset(state(memory(ts1,us),ds,raise(h-id),ial,
                      reset-info(ia2,ts2),hts,irs,dis))
               = state(memory(ts2,us),ds,fetch(ia2),next(ia2),
                       no-reset-info,hts,irs,dis)

end Exception-handling

module Processor
begin
   exports
     begin
      functions
        execute: STATE           -> STATE
     end

   imports Kernel-states, Evaluate-restrictions,
           Evaluate-ind-expr, Exception-handling, User-io

   variables
        m                :-> MEMORY
        ts               :-> TECH-SYMTAB
        us               :-> USER-SYMTAB
        ds               :-> DATA-STACK
        irs              :-> INP-RESTR-SEQ
        ri               :-> RESET-INFO
        id               :-> ID
        ie               :-> IND-EXPR
        i,int1,int2      :-> INDEX
        ia               :-> INSTR-ADDR
        is               :-> INSTR-SEQ
        h-id             :-> ID
        ht               :-> HANDLER-TABLES
        f-id             :-> ID
        td               :-> TECH-DATA
        b                :-> BOOL
        dis,mes          :-> STRING

    equations
     -- user interface instructions --

   [312] execute(state(m,ds,user-load(id),ia,ri,ht,irs,dis))
          = execute(if(user-load-excep(id,m),
                   state(user-insert(id,m),ds,
                         raise(load-excep-id(id,m)),ia,ri,
                         ht,irs,dis),
                   state(user-insert(id,m),ds,
                         fetch(ia),next(ia),
                         ri,ht,irs,dis)))
```

```
[313] execute(state(m,ds,user-store(id),ia,ri,ht,irs,dis))
       = execute(state(store-result(id,m),ds,
                       fetch(ia),next(ia),
                       ri,ht,irs,dis))

    -- load data stack instructions --

[314] execute(state(m,ds,load(id),ia,ri,ht,irs,dis))
       = execute(state(m,push(get-data(id,m),ds),
                       fetch(ia),next(ia),
                       ri,ht,irs,dis))

[315] execute(state(m,ds,load(td),ia,ri,ht,irs,dis))
       = execute(state(m,push(td,ds),
                       fetch(ia),next(ia),
                       ri,ht,irs,dis))

[316] execute(state(m,ds,load-elem(id,i),ia,ri,ht,irs,dis))
       = execute(state(m,
                       push(get-elem(
                              id,index-s(top(ds,i)),m),
                            pop(ds,i)),
                       fetch(ia),next(ia),
                       ri,ht,irs,dis))

[317] execute(state(m,ds,load-subm(id,i),ia,ri,ht,irs,dis))
       = execute(state(m,
                       push(get-subm(
                              id,range-s(top(ds,i)),m),
                            pop(ds,i)),
                       fetch(ia),next(ia),
                       ri,ht,irs,dis))


 -- store from data stack in memory instructions --

[318] execute(state(m,ds,store(id),ia,ri,ht,irs,dis))
       = execute(state(store-data(id,top(ds),m),
                       pop(ds),
                       fetch(ia),next(ia),
                       ri,ht,irs,dis))

[319] execute(state(m,ds,store-elem(id,i),ia,ri,ht,irs,dis))
       = execute(state(store-elem(id,top(ds),
                              index-s(top(pop(ds),i)),m),
                       pop(pop(ds),i),
                       fetch(ia),next(ia),
                       ri,ht,irs,dis))

[320] execute(state(m,ds,store-subm(id,i),ia,ri,ht,irs,dis))
      = execute(state(store-subm(id,top(ds),
                              range-s(top(pop(ds),i)),m),
                     pop(pop(ds),i),
                     fetch(ia),next(ia),
                     ri,ht,irs,dis))

    -- index expression instructions --
```

```
[321] execute(state(memory(ts,us),ds,eval-expr(ie),ia,
                ri,ht,irs,dis))
        = execute(state(m,
                        push(t-data(evaluate(ie,ts)),ds),
                        fetch(ia),next(ia),
                        ri,ht,irs,dis))

[322] execute(state(memory(ts,us),ds,check-ge-restr,ia,
                ri,ht,irs,dis))
        = if(check(irs,ts),
             execute(state(memory(ts,us),ds,
                          fetch(ia),next(ia),
                          ri,ht,irs,dis)),
             stop)

              -- other instructions --

[323] execute(state(m,ds,jump(int1),addr(is,int2),ri,ht,
                irs,dis))
        = execute(state(m,ds,
                        fetch(addr(is,add(int1,int2))),
                        next(addr(is,add(int1,int2))),
                        ri,ht,irs,dis))

[324] execute(state(m,push(t-data(b),ds),
                jump-true(int1),addr(is,int2),
                ri,ht,irs,dis))
        = execute(state(m,ds,
                        if(b,
                           fetch(addr(is,add(int1,int2))),
                           fetch(addr(is,int2))),
                        if(b,
                           next(addr(is,add(int1,int2))),
                           next(addr(is,int2))),
                        ri,ht,irs,dis))

  -- the error case (no boolean on the stack) --
  -- is left unspecified                        --

[325] execute(state(m,push(t-data(b),ds),
                jump-false(int1),addr(is,int2),
                ri,ht,irs,dis))
        = execute(state(m,ds,
                        if(not(b),
                           fetch(addr(is,add(int1,int2))),
                           fetch(addr(is,int2))),
                        if(not(b),
                           next(addr(is,add(int1,int2))),
                           next(addr(is,int2))),
                        ri,ht,irs,dis))

  -- the error case (no boolean on the stack) --
  -- is left unspecified                        --
```

```
[326] execute(state(m,ds,fcall(f-id,i),ia,ri,ht,irs,dis))
        = execute(if(excep-raised(f-id,ds),
                     state(m,
                           push(execute-f(f-id,ds),
                                pop(ds,i)),
                           raise(exception-f(f-id,ds)),ia,
                           ri,ht,irs,dis),
                     state(m,
                           push(execute-f(f-id,ds),
                                pop(ds,i)),
                           fetch(ia),next(ia),
                           ri,ht,irs,dis)))

  [327] execute(state(m,ds,raise(h-id),ia,ri,ht,irs,dis))
          = execute(
                handle(
                    state(m,ds,raise(h-id),ia,ri,ht,irs,dis)))

  [328] execute(state(m,ds,unraise,ia,ri,ht,irs,dis))
          = execute(reset(state(m,ds,
                            unraise,ia,ri,ht,irs,dis)))

  [329] execute(state(m,ds,display(mes),ia,ri,ht,irs,dis))
          = execute(state(m,ds,fetch(ia),next(ia),
                          ri,ht,irs,mes))

  [330] execute(stop) = stop

end Processor
```

9. KERNEL.

9.a. Global description.

The kernel can execute the statistical techniques stored in the statistical technique table. The execution of a statistical technique may invoke exception handlers. The first instruction to be executed is the first instruction in the instruction sequence of the statistical technique.
The results of the calculations after the execution of the statistical technique are stored in the user symbol table.

9.b. Specification.

```
module Kernel
begin
   exports
     begin
       functions
         run-techn: USER-SYMTAB # KERNEL-PRO #
                    EXT-HANDLER-TABLE -> USER-SYMTAB
         run-techn: ID # USER-SYMTAB # STAT-TECH-TABLE #
                    EXT-HANDLER-TABLE -> USER-SYMTAB
     end
```

```
imports Stat-technique-tables,
        Processor
variables
      us            :-> USER-SYMTAB
      ts            :-> TECH-SYMTAB
      is            :-> INSTR-SEQ
      irs           :-> INP-RESTR-SEQ
      eht           :-> EXT-HANDLER-TABLE
      tht           :-> TECH-HANDLER-TABLE
      stt           :-> STAT-TECH-TABLE
      id            :-> ID

equations

[331] run-techn(us,kern-pro(is,irs,ts,tht),eht)
       = result(
             execute(
                state(memory(ts,us),empty-stack,
                      first(is),addr(is,1),
                      no-reset-info,handlers(tht,eht),
                      irs,string(blank))))

[332] run-techn(id,us,stt,eht) = run-techn(us,id^stt,eht)

end Kernel
```

APPENDIX K. ASF SPECIFICATION OF THE GENERATION OF KERNEL INSTRUCTIONS

A statistical program is represented at the kernel level as a kernel program. The generation of a kernel instruction sequence for a statistical program and a technique handler constructed is specified in this appendix. It contains:
- modules that describe the generation of kernel instructions for variables, expressions and statements,
- modules that describe the generation of kernel instructions for statistical program sections; the instruction sequence generated for an exception handler section is stored in the technique symbol table,
- a module that describes the generation of a kernel instruction sequence and a technique handler table for an entire statistical program.

1. GENERATION OF KERNEL INSTRUCTIONS FOR THE BASIC ELEMENTS OF THE STATISTICAL LANGUAGE.

1.a. Global description.

In this section the generation of kernel instructions for variables, expressions and statements is specified. The assigment statements are written in postfix notation, making it possible for the kernel to evaluate the statements on the data stack. The message statement generates a display instruction. The instructions for a compound statement consists of the instructions generated for the individual statements in the compound statement. For a for-statement instruction are generated that initialize the control variable, check the restriction that the control variable is smaller than the upper limit, execute the statements inside the loop and increment the control variable. Also the size of the involved jumps is specified.

1.b. Specification.

```
module Gen-code-variables
begin
   exports
      begin
       functions
         gen-code: BOOL # INSTR-SEQ # VARIABLE     -> INSTR-SEQ
         gen-code: INSTR-SEQ # IND-EXPR-SEQ        -> INSTR-SEQ
         gen-code: INSTR-SEQ # RANGE-SEQ           -> INSTR-SEQ
      end

   imports Variable-abstr-syntax, Instruction-sequences,
           Booleans { renamed by [true -> load, false -> store] }

   variables
         is               :-> INSTR-SEQ
         id               :-> ID
         inds             :-> IND-EXPR-SEQ
         subrs            :-> RANGE-SEQ
         ie,ie1,ie2       :-> IND-EXPR
         flag             :-> BOOL
```

```
equations
            -- simple variables --

     [333] gen-code(flag,is,abs-var(id))
            = if (eq(flag,load), add-item(load(id),is),
                                 add-item(store(id),is))

            -- matrix elememt references --

     [334] gen-code(flag,is,abs-var(id,inds))
            = if ( eq(flag,load),
                   add-item(load-elem(id,n-of-dims(inds)),
                            gen-code(is,inds)),
                   add-item(store-elem(id,n-of-dims(inds)),
                            gen-code(is,inds)))

            -- submatrices --

     [335] gen-code(flag,is,abs-var(id,subrs))
            = if(eq(flag,load),
                 add-item(load-subm(id,n-of-rngs(subrs)),
                          gen-code(is,subrs)),
                 add-item(store-subm(id, n-of-rngs(subrs)),
                          gen-code(is,subrs)))

            -- index expr sequences --

     [336] gen-code(is,add-item(ie,inds))
            = add-item(eval-expr(ie),gen-code(is,inds))

     [337] gen-code(is,add-item(ie,null-ind-expr-seq))
            = add-item(eval-expr(ie),is)

            -- note that eval-expr(ie) is only the instruction  --
            -- that ie must be evaluated, the actual evaluation --
            -- is defined in the kernel.                        --

            -- range sequences --

     [338] gen-code(is,add-item(range(ie1,ie2),subrs))
            = add-item(eval-expr(ie1),
                       add-item(eval-expr(ie2),
                                gen-code(is,inds)))

     [339] gen-code(is,add-item(range(ie1,ie2),null-range))
            = add-item(eval-expr(ie1),
                       add-item(eval-expr(ie2),is))

end Gen-code-variables


module Gen-code-expressions
begin
   exports
      begin
        functions
          gen-code-expr:  INSTR-SEQ # EXPR          -> INSTR-SEQ
          gen-code-argl:  INSTR-SEQ # ARG-LIST      -> INSTR-SEQ
          n-of-args:      ARG-LIST                  -> INDEX
      end

   imports Expr-abstr-syntax, Instruction-sequences,
           Gen-code-variables

        variables
          is                    :-> INSTR-SEQ
          exp                   :-> EXPR
          var                   :-> VARIABLE
          td                    :-> TECH-DATA
          f-id                  :-> ID
          argl                  :-> ARG-LIST

        equations

        [340] gen-code-expr(is,abs-expr(var)) = gen-code(load,is,var)
        [341] gen-code-expr(is,abs-expr(td))  = add-item(load(td),is)

        -- function calls --

        [342] gen-code-expr(is,abs-f-call(f-id,argl))
               = add-item(fcall(f-id,n-of-args(argl)),
                          gen-code-argl(is,argl))

        [343] gen-code-argl(is,abs-arg-l(argl,exp))
               =gen-code-expr(gen-code-argl(is,argl),exp)

        [344] gen-code-argl(is,abs-arg-l(exp))
               =gen-code-expr(is,exp)

end Gen-code-expressions


module Gen-code-statements
begin
   exports
      begin
        functions
          gen-code-stmts: INSTR-SEQ # STATEMENTS -> INSTR-SEQ
          gen-code-stm:   INSTR-SEQ # STATEMENT  -> INSTR-SEQ

          -- function for the for-statement --

          increm-code:     ID # STATEMENTS # INSTR-SEQ     -> INSTR-SEQ
          check-up-limit:  ID # IND-EXPR # STATEMENTS #
                           INSTR-SEQ                       -> INSTR-SEQ
          ini-control-var: ID # IND-EXPR # INSTR-SEQ       -> INSTR-SEQ
          size-of-jump1:   STATEMENTS                      -> INDEX
          size-of-jump2:   STATEMENTS                      -> INDEX
      end

   imports Gen-code-expressions, Gen-code-variables,
           Statements-abstr-syntax, Instruction-sequences
```

```
variables
     is              :-> INSTR-SEQ
     stmts           :-> STATEMENTS
     stm             :-> STATEMENT
     var             :-> VARIABLE
     exp             :-> EXPR
     m               :-> STRING
     id              :-> ID
     ie,ie1,ie2      :-> IND-EXPR

equations

     [345] gen-code-stmts(is,abs-statmts(stmts,stm))
             = gen-code-stm(gen-code-stmts(is,stmts),stm)

     [346] gen-code-stmts(is,abs-statmts(stm))
             = gen-code-stm(is,stm)

     -- assignment statement --

     [347] gen-code-stm(is,abs-assgn(var,exp))
             = gen-code(load,gen-code-expr(is,exp),var)

     -- index assignment statements --

     [348] gen-code-stm(is,abs-ind-assgn(id,ie))
             = add-item(store(id),add-item(eval-expr(ie),is))

     -- message statement --

     [349] gen-code-stm(is,abs-messtmt(m))
             = add-item(display(m),is)

     -- compound statement --

     [350] gen-code-stm(is,abs-compound(stmts))
             = gen-code-stmts(is,stmts)

     -- for statement --

     [351] gen-code-stm(is,abs-for(id,ie1,ie2,stmts))
             = increm-code(id,stmts,
                    gen-code-stmts(
                         check-up-limit(id,ie2,stmts,
                            ini-control-var(id,ie1,is)),stmts))

     [352] ini-control-var(id,ie,is)
             = add-item(store(id),
                  add-item(eval-expr(ie),is))

     [353] check-up-limit(id,ie,stmts,is)
             = add-item(jump-true(size-of-jump1(stmts)),
                  add-item(fcall(greater-int,increm(1)),
                     add-item(load(id),
                        add-item(eval-expr(ie),is))))
```

```
     [354] increm-code(id,stmts,is)
             = add-item(jump(size-of-jump2(stmts)),
                       add-item(increm(id),is))

     [355] size-of-jump1(stmts)
             = add(n-of-items(gen-code-stmts(null-instr-seq,stmts)),
                    increm(1))

               -- number of instructions generated for statements --
               -- plus increment instruction and jump instruction --

     [356] size-of-jump2(stmts)
             = add(n-of-items(gen-code-stmts(null-instr-seq,stmts)),
                    increm(increm(increm(increm(1)))) )

               -- number of instructions generated for statements --
               -- plus increment instruction and instruction in   --
               -- upper limit check                               --

end Gen-code-statements
```

2. GENERATION OF KERNEL INSTRUCTIONS FOR SECTIONS OF A STATISTICAL PROGRAM

2.1. Global description.

Each section of a statistical program generates specific instruction sequences. These instruction sequences are added to instructions sequences generated in previous sections. An input/output section only generates user-load and user-store instructions. The exception handler section generates instruction sequences that are stored in a technique handler table.

2.2. Specification.

```
module Gen-code-declarations
begin
   exports
     begin
       functions
         gen-code-io-sect:   INSTR-SEQ # IO-SEC # BOOL   -> INSTR-SEQ
         gen-code-decls:     INSTR-SEQ # DECLS # BOOL    -> INSTR-SEQ
         gen-code-decl:      INSTR-SEQ # DECL # BOOL     -> INSTR-SEQ
         gen-code-id:        INSTR-SEQ # ID # BOOL       -> INSTR-SEQ
     end

   imports Instruction-sequences, Decl-abstr-syntax

   variables
        is                    :-> INSTR-SEQ
        inp-decls,out-decls   :-> DECLS
        decls                 :-> DECLS
        decl                  :-> DECL
        idl                   :-> ID-SEQ
        id                    :-> ID
        tt                    :-> TECH-TYPE
```

```
      uv                :-> USER-VIEW
      mess              :-> STRING
      flag              :-> BOOL

   equations

  -- the variable flag indicates whether input or output --
  -- instructions must be generated                      --

   [357] gen-code-io-sect(is,abs-io-sect(inp-decls,out-decls),flag)
           = if (eq(flag,load), gen-code-decls(is,inp-decls,flag),
                                gen-code-decls(is,out-decls,flag))

   [358] gen-code-decls(is,abs-decls(decls,decl),flag)
           = gen-code-decl(gen-code-decls(is,decls,flag),decl,flag)

   [359] gen-code-decls(is,abs-decls(decl),flag)
           = gen-code-decl(is,decl,flag)

   [360] gen-code-decl(is,abs-decl(add-item(id,idl),tt,mess),flag)
           = gen-code-id(
                gen-code-decl(is,abs-decl(idl,tt,mess),flag),
               id,flag)

   [361] gen-code-decl(is,abs-decl(null-id-seq,tt,mess),flag)
           = is

   [362] gen-code-id(is,id,load)  = add-item(user-load(id),is)
   [363] gen-code-id(is,id,store) = add-item(user-store(id),is)

 end Gen-code-declarations


 module Gen-code-implement-section
 begin
    exports
     begin
      functions
        gen-code-impl-sec: INSTR-SEQ # IMPL-SEC -> INSTR-SEQ
     end

    imports Gen-code-statements, Instruction-sequences,
            Impl-abstr-syntax

    variables
      is      :-> INSTR-SEQ
      decl    :-> INTERN-DECLS
      stmts   :-> STATEMENTS

    equations

    [364] gen-code-impl-sec(is,abs-impl-sec(decl,stmts))
            = gen-code-stmts(is,stmts)

 end Gen-code-implement-section
```

```
module Gen-code-test-section
   begin
     exports
      begin
       functions
         gen-code-test-sec:  INSTR-SEQ # TEST-SEC -> INSTR-SEQ
         gen-code-raises:    INSTR-SEQ # RAISES   -> INSTR-SEQ
         gen-code-raise:     INSTR-SEQ # RAISE    -> INSTR-SEQ
      end

     imports Gen-code-statements, Instruction-sequences,
             Test-abstr-syntax

     variables
       is            :-> INSTR-SEQ
       rs            :-> RAISES
       r             :-> RAISE
       id            :-> ID
       expr          :-> EXPR
       decl          :-> INTERN-DECLS
       stmts         :-> STATEMENTS

   equations

    [365] gen-code-test-sec(is,abs-test-sec(decl,stmts,rs))
            = gen-code-raises(gen-code-stmts(is,stmts),rs)

    [366] gen-code-raises(is,abs-raises(rs,r))
            = gen-code-raise(gen-code-raises(is,rs),r)

    [367] gen-code-raises(is,abs-raises(r))
            = gen-code-raise(is,r)

    [368] gen-code-raise(is,abs-raise(id,expr))
            = add-item(raise(id),
                       add-item(jump-false(increm(1)),
                                gen-code-expr(is,expr)))
end Gen-code-test-section


module Gen-code-handl-section
begin
   exports
     begin
      functions
        gen-code-handl-sec:  TECH-HANDLER-TABLE # HANDL-SEC  ->
                             TECH-HANDLER-TABLE
        gen-code-hndl:       TECH-HANDLER-TABLE # HANDLER    ->
                             TECH-HANDLER-TABLE
     end

   imports Gen-code-statements, Tech-handler-tables,
           Handler-abstr-syntax

   variables
      handl-s          :-> HANDL-SEC
      hndl             :-> HANDLER
```

```
        is              :-> INSTR-SEQ
        ht              :-> TECH-HANDLER-TABLE
        id              :-> ID
        decl            :-> INTERN-DECLS
        stmts           :-> STATEMENTS

    equations

    [369] gen-code-handl-sec(ht,abs-handl-sec(handl-s,hndl))
          = gen-code-hndl(gen-code-handl-sec(ht,handl-s),hndl)

    [370] gen-code-handl-sec(ht,abs-handl-sec(hndl))
          = gen-code-hndl(ht,hndl)

    [371] gen-code-hndl(ht,abs-handler(id,decl,stmts))
          = insert(id,
                   gen-code-stmts(null-instr-seq,stmts),
                   ht)

end Gen-code-handl-section
```

3. THE GENERATION OF KERNEL INSTRUCTION FOR A STATISTICAL PROGRAM

3.a. Global description.

An instruction sequence in a kernel program consists of a sequence of input instrictions, an instruction to check the input restrictions, a sequence that describes the calculations of the statistics, a sequence of output instructions, and a halt instruction. How input and output instruction sequence are generated for a statistical program is specified in module *Gen-io-instructions*. The generation of calculation instructions is specified in module *Gen-calc-instructions*. The instruction sequence are linked in module *Gen-instructions*. The technique handler table of a statistical technique contains the instruction sequences of the exception handlers in the statistical technique, as specified in module *Gen-handlers*.

3.b. Specification.

```
module Gen-io-instructions
begin
    exports
      begin
        functions
          gen-inp-instr: INSTR-SEQ # STAT-PRO ->INSTR-SEQ
          gen-inp-instr: INSTR-SEQ # SECTION  ->INSTR-SEQ
          gen-out-instr: INSTR-SEQ # STAT-PRO ->INSTR-SEQ
          gen-out-instr: INSTR-SEQ # SECTION  ->INSTR-SEQ
      end

    imports Gen-code-declarations, Statistical-programs

    variables
        sp              :-> STAT-PRO
        is              :-> INSTR-SEQ
        sct             :-> SECTION
        io-s            :-> IO-SEC
        impl-s          :-> IMPL-SEC
        test-s          :-> TEST-SEC
        handl-s         :-> HANDL-SEC
        id              :-> ID

    equations
        -- input instructions --

    [372] gen-inp-instr(is,abs-prog(sp,sct))
          = gen-inp-instr(gen-inp-instr(is,sp),sct)

    [373] gen-inp-instr(is,abs-prog(sct))
          = gen-inp-instr(is,sct)

    [374] gen-inp-instr(is,abs-sect(io-s))
          = gen-code-io-sect(is,io-s,load)

    [375] gen-inp-instr(is,abs-sect(impl-s))  = is
    [376] gen-inp-instr(is,abs-sect(handl-s)) = is
    [377] gen-inp-instr(is,abs-sect(test-s))  = is
    [378] gen-inp-instr(is,abs-name(id))      = is

        -- output instructions --

    [379] gen-out-instr(is,abs-prog(sp,sct))
          = gen-out-instr(gen-out-instr(is,sp),sct)

    [380] gen-out-instr(is,abs-prog(sct))
          = gen-out-instr(is,sct)

    [381] gen-out-instr(is,abs-sect(io-s))
          = gen-code-io-sect(is,io-s,store)

    [382] gen-out-instr(is,abs-sect(impl-s))  = is
    [383] gen-out-instr(is,abs-sect(handl-s)) = is
    [384] gen-out-instr(is,abs-sect(test-s))  = is
    [385] gen-out-instr(is,abs-name(id))      = is

end Gen-io-instructions


module Gen-calc-instructions
begin
    exports
      begin
        functions
          gen-calc-instr: INSTR-SEQ # STAT-PRO ->INSTR-SEQ
          gen-calc-instr: INSTR-SEQ # SECTION  ->INSTR-SEQ
      end
    imports Gen-code-implement-section, Gen-code-test-section,
            Statistical-programs

    variables
        sp              :-> STAT-PRO
        is              :-> INSTR-SEQ
        sct             :-> SECTION
        io-s            :-> IO-SEC
        impl-s          :-> IMPL-SEC
        test-s          :-> TEST-SEC
        handl-s         :-> HANDL-SEC
        id              :-> ID

    equations

    [386] gen-calc-instr(is,abs-prog(sp,sct))
            = gen-calc-instr(gen-calc-instr(is,sp),sct)

    [387] gen-calc-instr(is,abs-prog(sct))
            = gen-calc-instr(is,sct)

    [388] gen-calc-instr(is,abs-sect(impl-s))
            = gen-code-impl-sec(is,impl-s)

    [389] gen-calc-instr(is,abs-sect(test-s))
            = gen-code-test-sec(is,test-s)

    [390] gen-calc-instr(is,abs-sect(handl-s)) = is
    [391] gen-calc-instr(is,abs-sect(io-s))    = is
    [392] gen-calc-instr(is,abs-name(id))      = is

end Gen-calc-instructions


module Gen-handlers
begin
   exports
     begin
       functions
         gen-handlers: STAT-PRO                     -> TECH-HANDLER-TABLE
         gen-handlers: TECH-HANDLER-TABLE # SECTION -> TECH-HANDLER-TABLE
     end

   imports Gen-code-handl-section, Statistical-programs,
           Tech-handler-tables

   variables
        sp              :-> STAT-PRO
        ht              :-> TECH-HANDLER-TABLE
        sct             :-> SECTION
        io-s            :-> IO-SEC
        impl-s          :-> IMPL-SEC
        test-s          :-> TEST-SEC
        handl-s         :-> HANDL-SEC
        id              :-> ID

   equations

   [393] gen-handlers(abs-prog(sp,sct))
           = gen-handlers(gen-handlers(sp),sct)

   [394] gen-handlers(abs-prog(sct))
           = gen-handlers(empty-tech-hand-tab,sct)
```

```
   [395] gen-handlers(ht,abs-sect(handl-s))
           = gen-code-handl-sec(ht,handl-s)

   [396] gen-handlers(ht,abs-sect(impl-s)) = ht
   [397] gen-handlers(ht,abs-sect(test-s)) = ht
   [398] gen-handlers(ht,abs-sect(io-s))   = ht
   [399] gen-handlers(ht,abs-name(id))     = ht

end Gen-handlers


module  Gen-instructions
begin
   exports
     begin
      functions
        gen-check-instr: INSTR-SEQ # STAT-PRO   -> INSTR-SEQ
        gen-instr-seq:   STAT-PRO               -> INSTR-SEQ
     end

   imports Instruction-sequences, Gen-calc-instructions,
           Gen-io-instructions

   variables
      sp           :-> STAT-PRO
      is           :-> INSTR-SEQ

   equations

   [400]  gen-check-instr(is,sp) =  add-item(check-ge-restr,is)

   [401] gen-instr-seq(sp)
           = add-item(halt,
               gen-out-instr(
                 gen-calc-instr(
                   gen-check-instr(
                     gen-inp-instr(null-instr-seq,sp),sp),sp),sp))

end Gen-instructions
```

APPENDIX L. ASF SPECIFICATION OF THE OVERALL CONDUCTOR SYSTEM

In this appendix the overall CONDUCTOR system is specified.
In the CONDUCTOR SYSTEM:

- a technical statistician can implement statistical techniques, these techniques are stored in the statistical technique table,
- a data expert or computer scientist can implement external exception handlers,
- an applied statistician can use the implemented statistical techniques,
- a data expert can initialize the database system,
- a computer scientist can implement the functions that are available in the statistical language.

In this appendix we specify the compiler that generates kernel programs for statistical techniques (module *Compiler*), and the generation of external handlers (module *Gen-ext-handlers*). The statistical technique table and the external handler table together form the collection of available techniques for the applied statistician (module *Implemented-techniques*). An applied statistician may call the implemented techniques in a simple user language. The definition of the user language, in section 2 of this appendix, includes the definition of a database interface. Finally, in section 3 the overall CONDUCTOR system is specified.

1. COMPILERS.

1.a. Global description.

A statistical program is compiled into a kernel program as specified in module *Compiler*. External handlers are compiled as specified in module *Gen-ext-handlers*.

1.b. Specification.

```
module Compiler
begin
  exports
    begin
      functions
        compile:        STAT-PRO                        -> KERNEL-PRO
        store-stat-tech: STAT-PRO # STAT-TECH-TABLE -> STAT-TECH-TABLE
     end

  imports Statistical-programs, Kernel-programs,
          Input-restr-generator, Static-type-checking,
          Gen-instructions, Gen-handlers,
          Stat-technique-tables

  functions
     name:     STAT-PRO  -> ID

  variables
       sp       :-> STAT-PRO
       sst      :-> STAT-TECH-TABLE

  equations

  [402]  compile(sp) = kern-pro(gen-instr-seq(sp),
                                gen-restr-pro(sp,type-check-pro(sp)),
                                type-check-pro(sp),
                                gen-handlers(sp))

  [403]  store-stat-tech(sp,sst) = insert(name(sp),compile(sp),sst)

end Compiler


module Gen-ext-handlers
begin
  exports
    begin
      functions
        gen-ext-handler:   HANDLER -> EXT-HANDL
        store-ext-handler: HANDLER # EXT-HANDLER-TABLE ->
                                     EXT-HANDLER-TABLE
    end

  imports Handler-abstr-syntax, External-handlers,
          Stat-check-handlers, Gen-code-statements,
          Gen-restr-handlers, Ext-handler-tables

  variables
        id          :-> ID
        decl        :-> INTERN-DECLS
        stmts       :-> STATEMENTS
        ts          :-> TECH-SYMTAB
        eht         :-> EXT-HANDLER-TABLE
        hnd         :-> HANDLER

  equations

 [404]  gen-ext-handler(abs-handler(id,decl,stmts))
        = ext-handl(gen-code-stmts(null-instr-seq,stmts),
                    typcheck(abs-handler(id,ld,stmts),empty-mem))

        when
          eq-inp-restr-seq(gen-restr-handler(hnd,ts,no-restrictions),
                           no-restrictions)
           = true

        -- the error cases are left unspecified --

 [405] store-ext-handler(hnd,eht)
       = insert(id,
                gen-ext-handler(hnd),
                eht)

end Gen-ext-handlers
```

2. THE USER LANGUAGE.

2.a. Global description.

The user language allows the user to declare variables, initiliaze variables, retrieve series from a database, and call statistical techniques. The user can also determine the current data sample

2.b. Specification.

```
module User-programs
begin
    exports
       begin
         sorts USER-PRO, USER-COMMAND
         functions
         user-pro: USER-PRO # USER-COMMAND        -> USER-PRO
         user-pro: USER-COMMAND                   -> USER-PRO

         user-decl:  USER-TYPE # ID               -> USER-COMMAND
         set:        ID # USER-DATA               -> USER-COMMAND
         set-sample: CONST-RANGE-SEQ              -> USER-COMMAND
         retrieve:   ID                           -> USER-COMMAND
         call-tech:  ID                           -> USER-COMMAND
       end

   imports Identifiers, User-types, User-data,
           Const-range-sequences

end User-programs
```

3. THE RESULTING SOFTWARE.

3.a. Global description.

The implemented statistical techniques and exception handlers together form the implemented techniques. The statistical techniques can be called by the user in the resulting software. The resulting software also has a database interface. The user may retrieve data from the database, when data is retrieved also a background-query is started to check the consistency of the retrieved data.

3.b. Specification.

```
module Implemented-techniques
 begin
  exports
    begin
     sorts IMPL-TECH
     functions
      available: STAT-TECH-TABLE # EXT-HANDLER-TABLE -> IMPL-TECH
    end

 imports Stat-technique-tables, Ext-handler-tables

end Implemented-techniques
```

```
module Database-interface
begin
   parameters Current-database
     begin
       sorts DATA-BASE
       functions
           current-db:                                    -> DATA-BASE
           data-query:  DATA-BASE # ID # CONST-RANGE-SEQ -> USER-DATA
           retrv-data:  DATA-BASE # ID # CONST-RANGE-SEQ -> SCALAR-SEQ
           bg-query:    DATA-BASE # ID # CONST-RANGE-SEQ -> ID-SEQ
     end  Current-database
   exports
     begin
       sorts DATA-BASE
       functions
           query:  ID # CONST-RANGE-SEQ  -> USER-INFO
     end

   imports Identifiers, Const-range-sequences,
           User-symtab-info, Series, Id-sequences

   variables
       db     :-> DATA-BASE
       id     :-> ID
       crs    :-> CONST-RANGE-SEQ

  equations

     [406] data-query(db,id,crs) = u-data(ser(retrv-data(db,id,crs),
                                              crs,
                                              bg-query(db,id,crs)))

     [407] query(id,crs)         = us-info(series-type(crs),
                                           data-query(current-db,id,crs))

end Database-interface
```

```
module Resulting-software
begin
   exports
    begin
     sorts USER-STATE
     functions
      user-state:     USER-SYMTAB # CONST-RANGE-SEQ          -> USER-STATE
      exec-user-pro: USER-STATE # IMPL-TECH # USER-PRO      -> USER-STATE
      exec-user-com: USER-STATE # IMPL-TECH # USER-COMMAND -> USER-STATE
     end

   imports User-symtabs, Const-range-sequences, Kernel,
           Database-interface, User-programs, Implemented-techniques

   variables
     us                     :-> USER-SYMTAB
     crs,crs1,crs2          :-> CONST-RANGE-SEQ
     id                     :-> ID
     ut                     :-> USER-TYPE
    ud                    :-> USER-DATA
    stt                   :-> STAT-TECH-TABLE
    eht                   :-> EXT-HANDLER-TABLE
    it                    :-> IMPL-TECH

  equations

  [408] exec-user-com(user-state(us,crs),it,user-decl(ut,id))
        = user-state(insert-type(id,ut,us),crs)

  [409] exec-user-com(user-state(us,crs),it,set(id,ud))
        = user-state(insert-data(id,ud,us),crs)

  [410] exec-user-com(user-state(us,crs1),it,set-sample(crs2))
        = user-state(us,crs2)

  [411] exec-user-com(user-state(us,crs),it,retrieve(id))
        = user-state(insert(id,query(id,crs),us),crs)

  [412] exec-user-com(user-state(us,crs),
                      available(stt,eht),call-tech(id))
        = user-state(run-techn(id,us,stt,eht),crs)

end Resulting-software
```

4. THE ENTIRE CONDUCTOR SYSTEM

4.a Global description.

In the CONDUCTOR specification environment a technical statistician can implement statistical techniques, other experts can implement exception handlers and a user can apply the implemented statistical techniques and exception handlers.

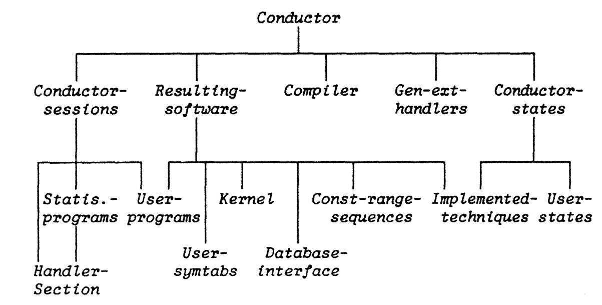
```
module Conductor-sessions
begin
  exports
    begin
      sorts SESSIONS, SESSION
      functions
        abs-sessions: SESSIONS # SESSION   -> SESSIONS
        abs-sessions: SESSION              -> SESSIONS
        user-session: USER-PRO             -> SESSION
        stat-session: STAT-PRO             -> SESSION
        hand-session: HANDLER              -> SESSION
    end

  imports User-programs, Statistical-programs, Handler-abstr-syntax

end Conductor-sessions
```

```
module Conductor-states
begin
  exports
    begin
      sorts CDT-STATE
      functions
        state: USER-STATE # STAT-TECH-TABLE #
               EXT-HANDLER-TABLE                    -> CDT-STATE
    end
  imports  Resulting-software, Stat-technique-tables,
           Ext-handler-tables
end Conductor-states

module Conductor
begin
  exports
    begin
      functions
          execute: SESSIONS # CDT-STATE   -> CDT-STATE
          execute: SESSION  # CDT-STATE   -> CDT-STATE
    end

  imports Conductor-sessions, Conductor-states,
          Compiler, Gen-ext-handlers, Resulting-software

   variables
      ss          :-> SESSIONS
      s           :-> SESSION
      cs          :-> CDT-STATE
      ust         :-> USER-STATE
      stt         :-> STAT-TECH-TABLE
      eht         :-> EXT-HANDLER-TABLE
      sp          :-> STAT-PRO
      up          :-> USER-PRO
      hnd         :-> HANDLER

   equations

   [413] execute(abs-sessions(ss,s),cs)
         = execute(s,execute(ss,cs))

   [414] execute(abs-sessions(s),cs)
         = execute(s,
                   state(user-state(empty-u-symtab,null-cr-seq),
                         empty-stat-tech-tab,
                         empty-ext-hand-tab))

   [415] execute(user-session(up),state(ust,stt,eht))
         = state(exec-user-pro(ust,available(stt,eht),up),stt,eht)

   [416] execute(stat-session(sp),state(ust,stt,eht))
         = state(ust,store-stat-tech(sp,stt),eht)

   [417] execute(hand-session(hnd),state(ust,stt,eht))
         = state(ust,stt,store-ext-handler(hnd,eht))

end Conductor
```

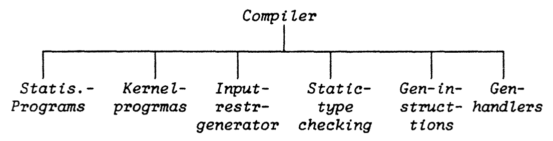
APPENDIX M: TREE REPRESENTATION OF CONDUCTOR

Trees appendix L.

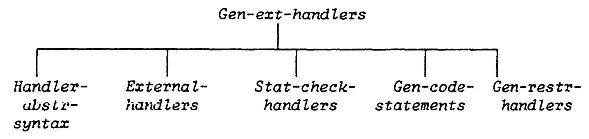
Tree L1. The overall CONDUCTOR system.



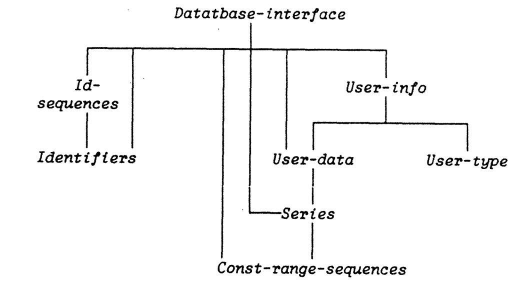
Tree L2. The Compiler.



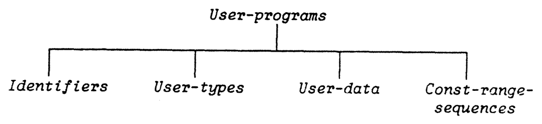
Tree L3. Generation of external handlers.



Tree L4. The Database Interface

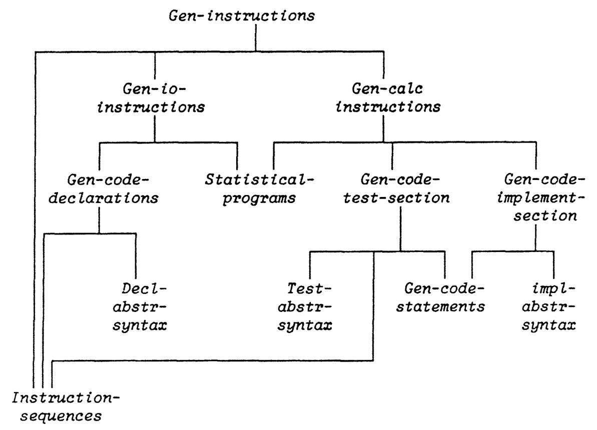


Tree L5. The user-program.

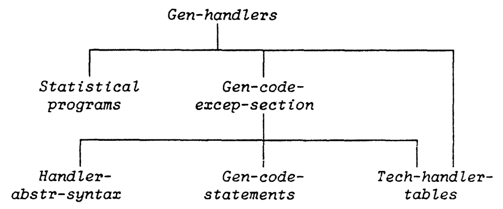


Trees appendix K.

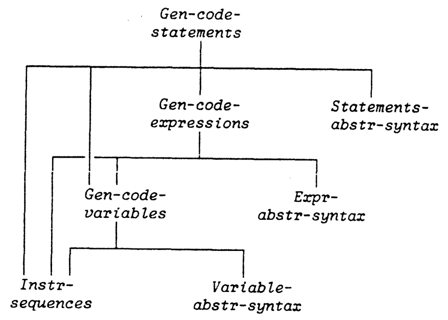
Tree K1. The generation of kernel instructions.



Tree K2. The generation of an exception handler in statistical technique.

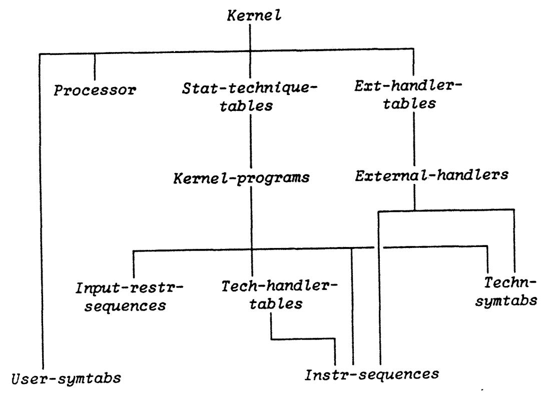


Tree K3. The generation of kernel instructions for statements.

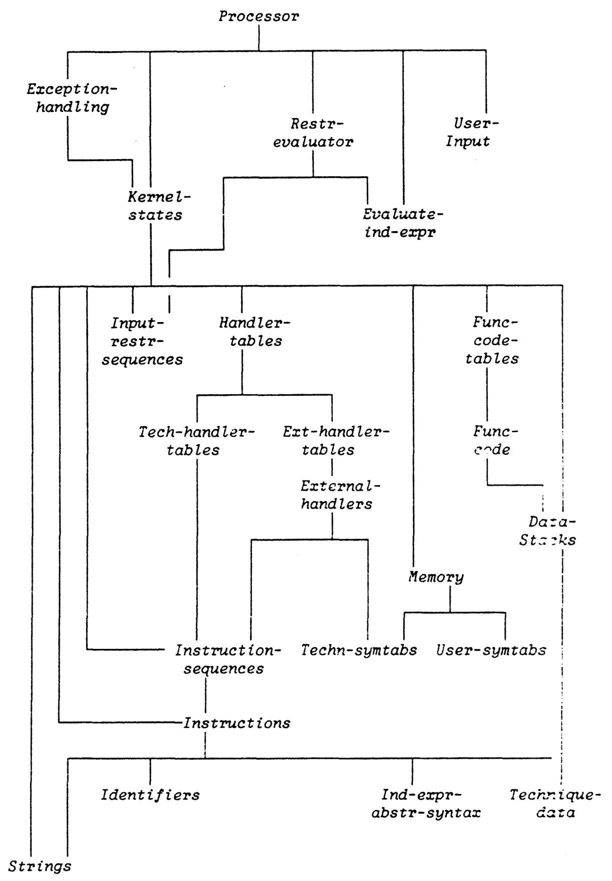


Trees Appendix J.

Tree J1. The kernel.

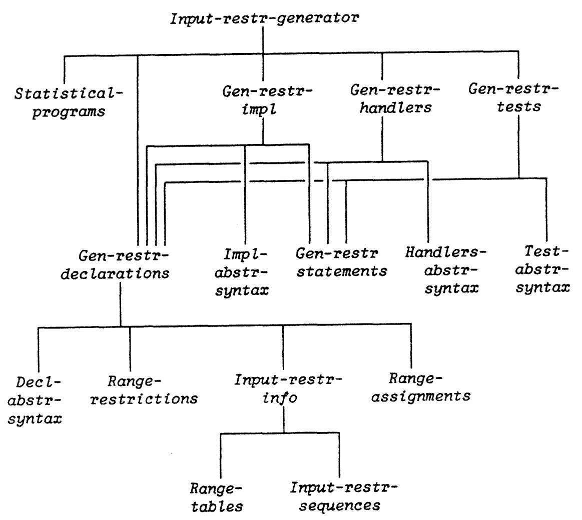


Tree J2. The Processor.

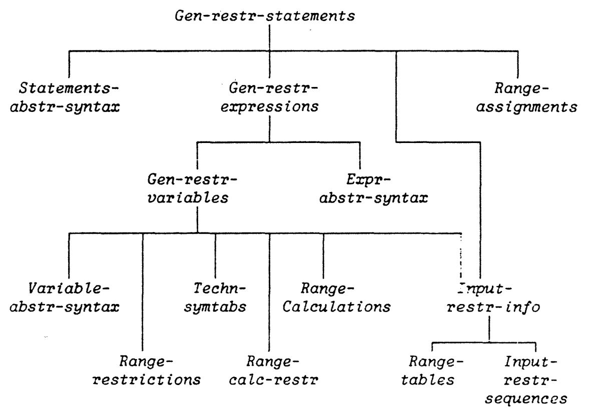


Trees appendix I.

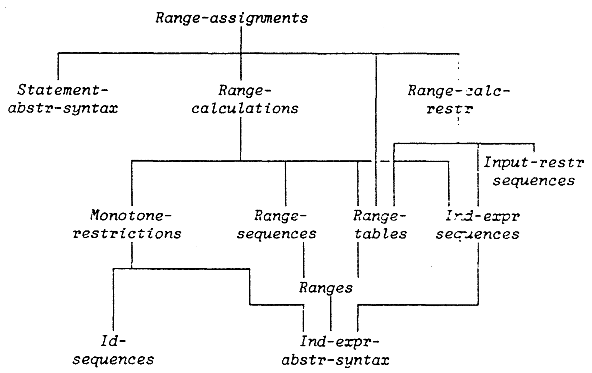
Tree I1. Input restriction generator.



Tree I2. Generation fo input restrictions for statements, expressions and variables.

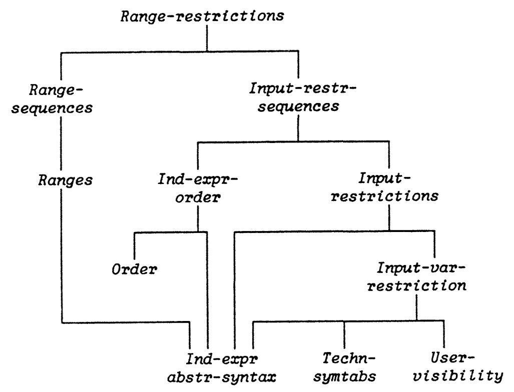


Tree I3. Range assignments and calculations.



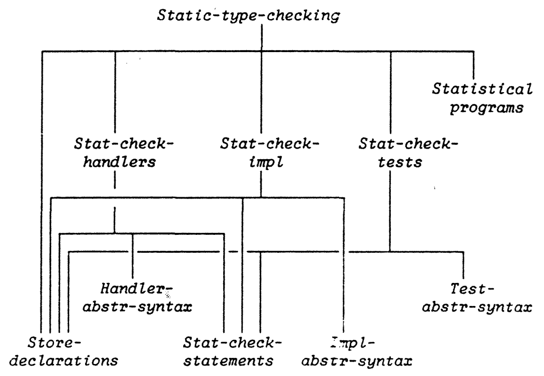
Tree I4. Range restrictions.

Tree I4. Range restrictions.

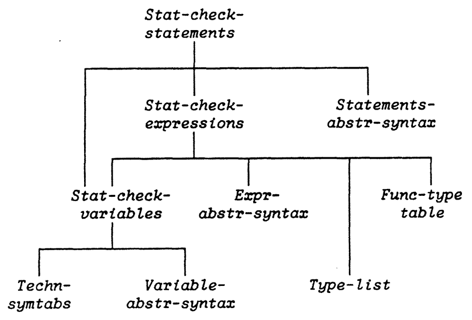


<u>Trees appendix H</u>.

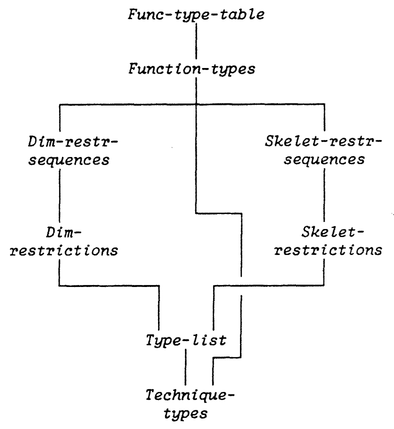
Tree H1. Static type checking of a statistical program.



Tree H2. Static type checking of statements.

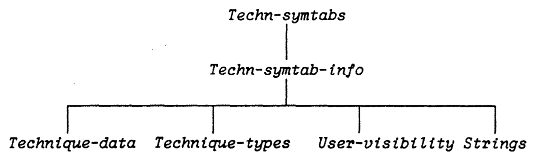


Tree H3. Function-type restrictions.

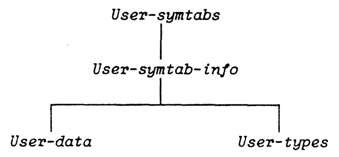


Trees appendix G.

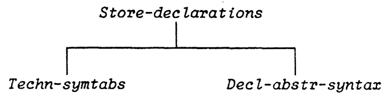
Tree G1. The technique symbol table.



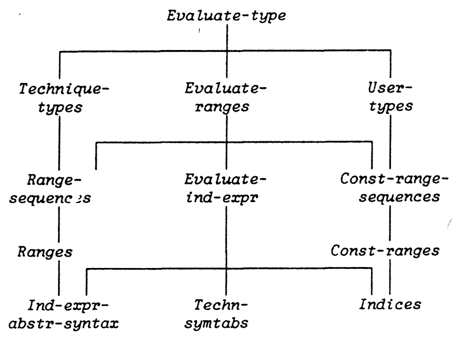
Tree G2. The user symbol table.



Tree G3. Storing declarations in the technique symbol table.

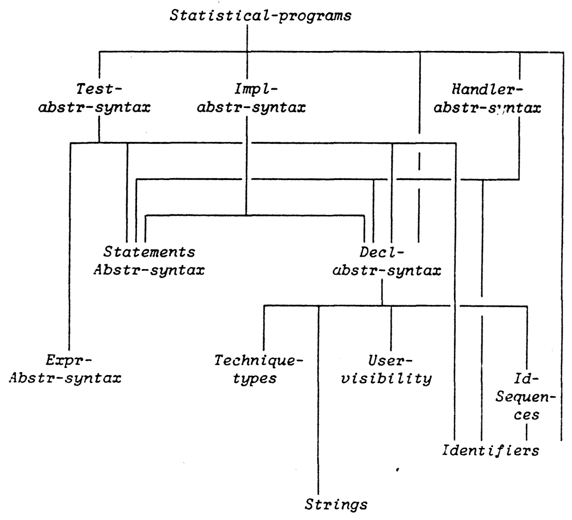


Tree G4. Evaluation of symbolic types, ranges and index expressions.

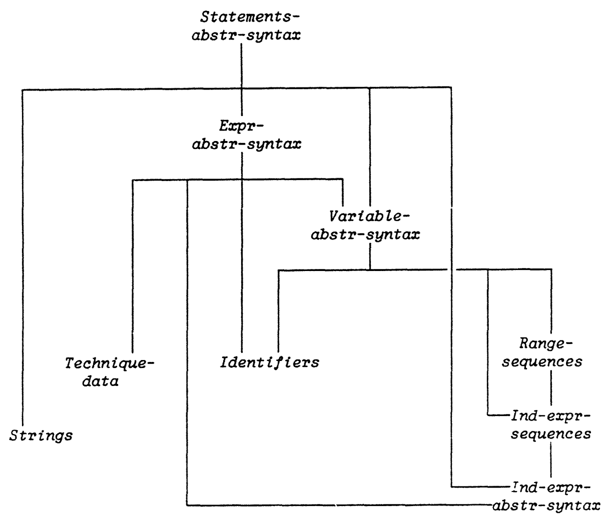


Trees appendix F.

Tree F1. Statistical program.

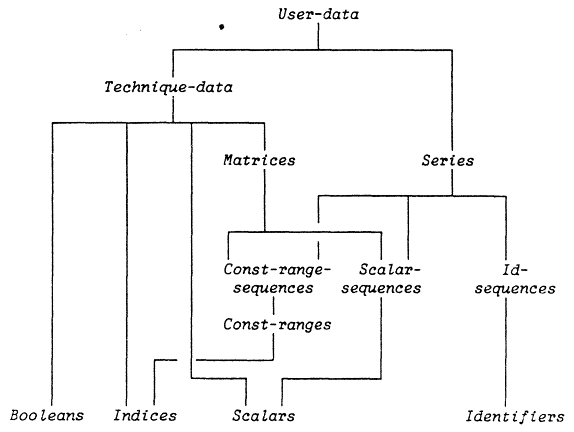


Tree F2. Statements.

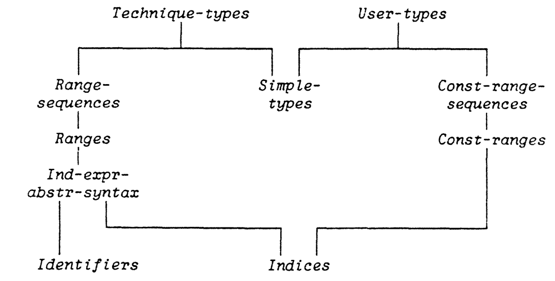


Trees appendices D and E.

Tree DE1. Datatypes.



Tree DE2. Type descriptions


Technique-types
User-types
Range-sequences
Simple-types
Const-range-sequences
Ranges
Const-ranges
Ind-expr-abstr-syntax
Identifiers
Indices