WORKING PAPER

PROBLEM INTERFACE FOR NONLINEAR DIDAS
Part 1: Static Systems

Andrzej Lewandowski

September 1986
WP-86-50

IIASA
International Institute
for Applied Systems Analysis

PROBLEM INTERFACE FOR NONLINEAR DIDAS

Part 1: Static Systems

Andrzej Lewandowski

September 1986
WP-86-50

INTERNATIONAL INSTITUTE FOR APPLIED SYSTEMS ANALYSIS
A-2361 Laxenburg, Austria

Foreword

One of the important problems of designing and implementing Decision Support System relates to the user friendliness and simplicity of problem definition. This is especially important in the case when a model of the system, constraints and objectives are described in terms of nonlinear equations. In all existing implementations of decision support systems this definition must be performed on the level of FORTRAN or other high-level language, which requires a rather deep knowledge of computer programming. Preparation of the problem, especially analytical computation of derivatives can also be the source of errors.

In the paper the principles of implementation of user-friendly interface to DIDAS system is presented. This interface utilizes the small subset of the PASCAL language for defining the problem; the compiler of the language performs all the algebraic manipulations necessary to analytical calculation and computation of the derivatives. This concept simplifies essentially the utilization of the DIDAS system and it can be extended for many other applications.

Alexander Kurzhanski
Chairman
System and Decision Sciences Program

Table of Contents

PROBLEM INTERFACE FOR NONLINEAR DIDAS
Part 1: Static systems

Andrzej Lewandowski

1. INTRODUCTION

The existing experience with various implementations of interactive decision support system - for example, of DIDAS type (Dynamic Interactive Decision Analysis and Support, see Grauer at all., 1984) indicates, that one of the important features, deciding about real applicability of the software is simplicity and user friendliness of the man machine interface (Lewandowski, 1986). This problem is especially important when constructing decision support systems for problems described by nonlinear mathematical models.

Problems described by mathematical models of linear structure were investigated by many authors and several approaches for defining such problems interactively were proposed. Such problems are somehow easier to define because the entire model can be specified as a collection of vectors and matrices. Therefore, the task of defining the problem - i.e. converting initial data and knowledge about the structure of the problem into a linear programming model can be performed without essential conceptual problems. Several software tools supporting this task were proposed recently (Orchard-Hays, 1978, Fourer, 1983). Another approach, recently gaining on popularity, is the use of concept of a spreadsheet program.

When dealing with nonlinear models, the situation is much more complicated. One of the reasons is the negative character of the definition of nonlinear problems: "a nonlinear problem is such a problem, which is not linear". Another source of difficulties is connected with the requirement of calculating the derivatives of the objective and constraints functions. These derivatives are necessary when applying nonlinear programming methods, since practically only differentiable optimization methods are sufficiently efficient and robust to be applied in interactive decision support systems. Usually the derivatives must be calculated analytically by the user of the system and properly interfaced to the rest of software. This task is time consuming and can be a source of errors that are typically difficult to detect. Another problem relates to the nature of problem interface - the user usually is forced to define his problems in FORTRAN or other high level language according to the specification of the interface provided by the implementator of the system. This can be a rather complicated task, requiring certain skills in computing.

All these difficulties must be overcome when we want to design a

user-friendly man machine interface for a decision support system. The user friendliness of the man machine and problem interface is especially important for microcomputer implementations of the DIDAS systems. These implementations are especially dedicated for users not being computer specialists.

The aim of this paper is to present a model implementation of a flexible problem interface based on the PASCAL language. In order to make the implementation sufficiently simple, a very small subset of PASCAL was implemented. It is a straightforward task to implement the same features in full sized PASCAL, particularly since a source code of several PASCAL compilers is available (PASCAL-S, see Wirth, 1981, and PASCAL-P, see Nori at all, 1981 and Pemberton, 1982). Despite the simplicity of the implemented language, the problem interface described in the paper is a rather powerful tool, which can be used for defining quite complicated, practical problems.

2. AUTOMATIC COMPUTATION OF GRADIENT

The problem of gradient calculation was investigated by many scientists working in the field of mathematical programming and sensitivity analysis. The algorithmic approach for calculating derivatives, was proposed for application in sensitivity analysis by Tomovic and Vukobratovic (1972). One of the early works was done at IIASA by Orchard-Hays (1978), but his system was oriented only to automatic differentiation of polynomials. Deep investigation of the problem of differentiation of mathematical models, especially of implicit type and dynamic models described by differential and difference equations were performed by Wierzbicki (1977, 1985). The most complete presentation of various techniques of gradient computation and possible applications of these techniques can be found in the monography by Rall (1981) and his revue paper (Rall, 1980).

Let us analyze several existing approaches supporting the problem of automatic calculation of derivatives.

2.1 Application of general purpose languages.

The first approach was proposed by Wegnert (1964) and later exploited by Kalaba and others (Kalaba, 1965, 1983, 1984). They utilize FORTRAN to write programs for automatic calculation of derivatives. The principles of the method (known as "table method") are as follows:

- every variable is represented by 2 (or more) dimensional array; the first element of the array contains value of the variable, the next - values of first and possibly higher derivatives,

- all standard functions and operators are emulated by special subroutines; these subroutines calculate values of the result as well as values of corresponding derivatives,

- mathematical expressions are composed from the emulating subroutines in such a way, that the resulting program reflects the tree structure of computed expression.

The following are the examples of subroutines which emulate addition

and multiplication:

```
SUBROUTINE ADD(X,Y,R)
DIMENSION X(2),Y(2),R(2)
R(1)=X(1)+Y(1)
R(2)=X(2)+Y(2)
RETURN

SUBROUTINE MUL(X,Y,R)
DIMENSION X(2),Y(2),R(2)
R(1)=X(1)+Y(1)
R(2)=X(1)*Y(2)+Y(1)*X(2)
RETURN
```

If the user wants to calculate the value and the derivative of the expression like

```
x1*x2+x3
```

he should prepare the following program

```
DIMENSION X1(2),X2(2),X3(2),R1(2),R2(2)
...
...

...
CALL MUL(X1,X2,R1)
CALL ADD(R1,X3,R2)
```

Evidently, some "initial conditions" must be set for values of derivatives; if the user wants to calculate the derivative of the above expression with respect to x1, the following assignments must precede the subroutine calls:

```
X1(2)=1
X2(2)=0
X3(2)=0
```

In the similar way, higher order derivatives can be calculated. Details of this process can be found in the paper by Kalaba and Tishler (1983).

Other languages were utilized for writing programs for automatic calculation of gradient. ALGOL 60 was used by Van de Riet, who in 2 volume report (Van de Riet, 1970) gives very detailed analysis of the use of high level numerically oriented language for formula analysis, investigates such problems like formula simplification and proposes convenient tools for defining problems. A similar system, but written in PASCAL, was designed by Shearer and Wolfe (1985). ALGOL 68 was used as the implementation language by Ince and Robson (1980).

The approach presented above is evidently the simplest one. It does not require special (usually very complicated) tools - it is enough to have access to any high level language and certain "know how" to define a particular problem. This approach is, however, available only for experienced programmers and is too complicated for end users. It can however be efficiently utilized for implementing dedicated decision support systems, where the problem definition is spe-

cified on the implementation stage and remains unchanged during utilization of the decision support system.

2.2 Preprocessors to high level languages.

The direct utilization of high level languages is rather not a straightforward task. It follows from two reasons:

- the implementator must posses certain knowledge about the used method;

- the mathematical expression must be converted to postfix form; in other words, it is necessary to perform manual "parsing" of the expression. This process is time consuming and can be the source of errors.

Parsing of mathematical expressions is a well recognized task which can be easily computerized. Therefore it is possible to create a computer program which could convert the mathematical expression formulated in standard notation, into high level program according to the concepts presented above. Such programs are known as **preprocessors.**

The number of such preprocessors for formula manipulation are known. The ALGOL 60 based one was designed by Van de Riet (ABC ALGOL, Van de Riet, 1973). This constitutes in fact a full featured language, being the superset of ALGOL 60. All the standard features of ALGOL are available to the user. Additionally new standard type **formula** makes possible a rather extensive formula manipulation. The source program written in ABC ALGOL is translated to standard ALGOL and further processed by ALGOL compiler. The ABC-ALGOL can be easily implemented on any computer, because the preprocessor is written in standard ALGOL. The only requirement is availability of ALGOL compiler.

The other similar preprocessor is AUGMENT preprocessor (Kedem, 1980), a FORTRAN based system. The principle of design is exactly the same like ABC ALGOL. The user specifies his problem in extended FORTRAN and the preprocessor converts the problem description into sequence of standard FORTRAN statements. Both systems are expandable - the user can define his own operators and supply his own subroutines and procedures performing necessary actions. Similar preprocessor - CODEX and SUPER-CODEX was developed by Rall (1981).

The preprocessors mentioned above constitute a very powerful tool - but also for experienced user. Good familiarity with the host language is necessary to use these systems efficiently. Moreover, the formula-oriented features are rather sophisticated and long training as well as some understanding of the internal organization of the system is necessary to use them efficiently. The power of such systems follows from the accessibility to all features of host level language as well as the availability of the compilation product which itself is a high level language program. This makes adaptation of the resulting program relatively easy and ensures very high flexibility of this approach.

2.3 Application of very high level symbolic manipulation languages.

The number of special languages were developed which are either especially designed for formula manipulation, or are more general pur-

pose oriented, but are especially convenient for formula manipulation.

The most known language of this type is LISP (Winston, 1981). Formula differentiation and manipulation can be programmed in LISP rather easily (Nicol, 1981), however for a programmer having long experience with FORTRAN or PASCAL like languages moving to LISP is connected with serious conceptual difficulties. A number of packages for formula manipulation were implemented in LISP, most of them for applications in theoretical physics.

A similar tool is MU-SIMP language developed by SoftWarehouse for a broad range of microcomputers (Apple, IBM-PC and others) and distributed by MICROSOFT (1983). This language is in fact an extension of LISP, but unlike LISP which is a list processing language, MU-SIMP is a tree processing language. This fact and much more convenient syntax of MU-SIMP makes this language an ideal tool for implementing formula manipulation problems (Douglass, 1982). Together with MU-SIMP, the MU-MATH system for symbolic formula manipulation, implemented in MU-SIMP is distributed. The power and the flexibility of the language as well as the availability of source codes of rather sophisticated formula manipulation algorithms, make this language the most promising tool for programming the user and problem interface for decision support system. This option however, has not been sufficiently investigated as yet.

Another general purpose high level language which can be used for implementing the formula differentiation and manipulation algorithms is PROLOG (Burnham and Hall, 1985). Programming the formula manipulation in PROLOG constitutes a rather simple task - but similarly like in the case of LISP, switching to this language can be rather difficult.

All the languages listed above are oriented to processing of symbolic information and lack such features like highly efficient numerical computation, file processing and flexible access to screen and keyboard. Interface to other general purpose languages like C or PASCAL is limited or not available at all. Therefore certain effort is necessary to analyze the practical applicability of these languages for implementing the interfaces for decision support systems.

2.4 Formula manipulation languages.

From the very early history of computer science a lot of works have been done on development the special purpose languages for symbolic formula manipulation. Currently, the MACSYMA and REDUCE are most known and are most widely used (see Fateman, 1982 for detailed discussion). Symbolic mathematical computation languages are also discussed by Wolfram (1985) who presents various approaches and discusses the features of SMP - one of the most powerful symbolic manipulation languages currently available. The other language possessing certain popularity is PASCAL-SC (PASCAL for Scientific Computation, see Kulish and Miranker, 1983). PASCAL-SC was applied for gradient computation by Rall (1983, 1984).

The situation with formula manipulation languages is similar to the described above - these languages are usually very complicated, require long training, interfacing to numerical and file processing modules can be difficult. The computer resources necessary to effec-

tively run symbolic manipulation programs are rather high (for example, according to Wolfram, 1985, the kernel of SMP language contains over 120000 lines of C code). Therefore, they can be rather considered as tools for high qualified specialists in mathematics and computer science, and it seems to be rather unlikely to use them as user front end for decision support systems, or tools for implementing such front end.

3. PROBLEM INTERFACES IN EXISTING IMPLEMENTATIONS OF NONLINEAR DIDAS SYSTEM

Currently, there exist 3 versions of nonlinear DIDAS: DIDAS-N developed by Grauer and Kaden (1985), a specialized system developed by Kaden and Kreglewski (1985) for solving decision problems relating to groundwater management and general purpose DIDAS system implemented by Kreglewski and others (Kreglewski, et. all., 1985). Let us analyze the problem interfaces available in all these versions of system.

The interface used in Grauer´s and Kaden´s version of the system is rather conceptually simple, but not very user-friendly. The equations describing objective and constraints functions must be programmed in FORTRAN. The authors supply the "skeleton" FORTRAN subroutine with empty "holes", where the user must locate his FORTRAN code. This is a rather complicated task - separate parts of the problem definition must be located in various places of the code, and the code itself must be written taking into account the variable names and structure used in this skeleton subroutine. What makes defining the problem especially difficult is the fact that, writing his code, the user must properly augment all his formulas with the penalty function terms and their derivatives. This is conceptually rather difficult for a user which is not familiar with mathematical programming algorithms and can lead to numerous errors. Variable names conflict is also probable.

The general purpose version of nonlinear DIDAS developed by Kreglewski and others (1985) also needs a FORTRAN subroutine containing the problem description. Unlike to the previous system, however, the user must preserve only the general structure of the subroutine header (formal parameters declaration) and COMMON block. No variable conflict can occur, and the standards according to which the body of the subroutine must be composed are quite clear and straightforward. The main disadvantage of this interface relates to the definition of derivatives - the user must calculate these derivatives analytically. This is usually a time consuming process and the source of various errors that are difficult to detect.

To minimize the probability of occurrence of errors in analytical gradient computation, several authors proposed numerical procedures for gradient checking. Detailed analysis of the problem, and sample procedures were discussed by Wolfe (1982). This approach was utilized by Kreglewski (1985) in his version of the DIDAS system; a similar procedure was implemented in MINOS-AUGMENTED nonlinear programming system.

The simplest interface has the DIDAS-like system for solving water management problems developed by Kaden and Kreglewski. This system was designed to solve only one class of problems, therefore a model of the system was programmed only once, in a very efficient way. The user interacts with the system only on the level of input data and

reference point selection.

Concluding, the following are the basic disadvantage of the existing implementation of problem interface in nonlinear DIDAS:

- the user must compute analytically all derivatives of objective and constraint functions,

- the objective and constraint functions as well as all derivatives must be programmed in FORTRAN according to the specification supplied by the implementator of the system; this specification can be difficult to understand for non-experienced user,

- the user must be familiar with details of the computing environment of the computer which he is working with - such like program editor, compiler, linker, operating system command language etc. This is the most severe limitation, which restricts essentially the usability of the system; a long training is necessary to work with the computer efficiently and without troubles on this level of interaction.

- any changes of the model - the process being in fact one of the important stages of interactive work with the system, cannot be performed within the system.

Let us point out that problem definition and modification is one of the most important stages of working with any decision support system. The sequence: program editor - FORTRAN compiler - linker - operating system, being in the fact one of the stages of interaction with the system, slows down essentially the interaction process, makes it difficult and inefficient. Therefore, the user friendliness of the problem interface requires special attention - especially in the case of nonlinear problems.

4. POSSIBLE APPROACHES IN DESIGN OF PROBLEM INTERFACE FOR NONLINEAR DIDAS

Currently, there does not exist ready to use tools which could be applied directly and without essential modifications for building problem interface for nonlinear DIDAS. It is possible, however, to adapt some existing methodologies of defining nonlinear problems for applications in decision support systems. Let us analyze the existing options.

4.1 Spreadsheet programs.

This method of interaction is very popular and used in almost all business oriented software. It can easily be adapted for defining linear programming problems - it is very easy to enter the standard or multiple criteria linear programming problem as set of matrices. There exist commercial codes for defining and solving linear programming problems which base on well known LOTUS spreadsheet program (General Optimization Inc., 1986). The interfacing between spreadsheet program and linear programming solver is relatively easy - each spreadsheet program can generate output files containing all data entered to the spreadsheet cells. This file can be read easily by linear programming solver; this solver can generate the solution file which can be imported to the spreadsheet.

In the case of nonlinear problems the situation is not so easy. In the principle, all spreadsheet programs make it possible to enter and to use formulas as cell contents. Unfortunately, it is usually not possible to transfer formulas to external file. It is possible to save the whole worksheet on disk, but the file format is proprietary and not described in the manual. Some options are available (like .PRN file in LOTUS) which make it possible to saving formulas in ASCII file, but usefulness of such information is questionable. The most severe limitation is caused by lack of accessibility to the formula evaluator module. Therefore, commercial spreadsheet programs like LOTUS, MULTIPLAN and others cannot be used directly for defining nonlinear decision analysis problems.

The spreadsheet interaction methodology has however several advantages, which could motivate further research in this direction. The properties of a spreadsheet which make this approach convenient for the user are as follows:

- a spreadsheet belongs to the class of **non procedural languages**, i.e. the sequence in which the separate formulas are evaluated depends only on the logical relationships between formulas and data, not on the sequence in which they are appear in the program text,

- rather extensive testing of program and data correctness can be performed in a spreadsheet program; this is possible due to existence of NULL and N/AVAIL data types,

- program entering and editing are very easy in a spreadsheet,

- a large set of "business oriented" standard functions is build into a typical spreadsheet,

- a spreadsheet is typically integrated with graphic and data base subsystems.

As it was mentioned above, commercially available spreadsheet programs cannot be used directly as the user front end for decision support systems. It would be highly interesting however, to apply the concept of spreadsheet for this purpose. To achieve this, either a spreadsheet program must be coded in one of the available high level languages, or spreadsheet codes being in public domain could be used for this purpose. One of the possible alternatives (due to several restrictions, only for experimental implementations) could be MICRO-CALC program supplied by BORLAND together with TURBO-PASCAL (Borland, 1985).

The structure of MICRO-CALC (as well as similar spreadsheet programs is quite straightforward - it consists of the interactive interface, data spreadsheet and formula evaluator. The formula evaluator calculates all formulas present in the data sheet, after every change in the spreadsheet or on user's request. To use MICRO-CALC as problem interface to DIDAS the following changes and amendments to this system must be provided:

- selected variables must be marked as decision variables, objective variables and constraints variables,

- each "numerical" cell should contain not only values of the objective, but also values of all derivatives with respect to all decision variables currently defined. Therefore such spreadsheet could be treated as a "multilayer" one.

- the formula evaluator must be properly modified in such a way, that computing the value of formula, all the derivatives should be calculated simultaneously,

- the formula evaluator should be implemented as procedure accessible both from the spreadsheet interface level and from the solver (optimization routine),

- the interface between the formula evaluator and the solver must be provided, in order to inform the solver about locations of decision variables, constraint variables and objective variables.

All the above changes are rather easy to implement; the only change which is more complicated relates to incorporating the formula differentiation process into the formula evaluator routine. This is discussed in a further section of the paper.

4.2 The TK!Solver approach.

It was recognized relatively early, that conventional spreadsheet approach is not very useful for analysis of more complicated problems arising in complex model applications. This takes place especially, if model equations are formulated in implicit form, i.e. in the form of set of equations which must be solved in order to find values of objective functions corresponding to given decision variables.

The standard spreadsheet programs can be applied even in this case, but resulting programs are not natural and this task requires some "smart" programming techniques (see, e.g. Haynes, 1985 where LOTUS was applied to dynamic analysis of electronic circuits). Therefore special approaches were developed to handle such problems.

One of the last and most interesting is the TK!Solver system developed by Konopasek (1985) and commercially distributed by Software Arts for IBM-PC computers. This system has spreadsheet like user interface, which operates on two data sheets - the variable sheet and rule sheet. In the variable sheet the user can define all the variables needed to define his problem completely. Some of them can de defined as input, the other are considered as output ones. The rules specify relationships between variables in terms of equations. The rule equations can be entered to the system in any form and any order. The second part of the system consists of the algebraic manipulator and iterative equation solver. If the user changes definition of output or input variables, modifies some rules or numerical values, the system responds very quickly with new solution of the problem. On Fig.1 a sample screen generated by TK!Solver is presented.

The TK!Solver concept is rather flexible and user friendly. What is most important is the fact, that the equations can be entered in any order and any form - therefore this approach also can be treated

as non procedural. The presented framework seems to be ideal for defining nonlinear problems. There were some attempts to apply this system for defining optimization problems (Konopasek, 1985), but these trials must be treated as very initial ones. To convert the TK!Solver program in a tool for an easy definition of nonlinear decision problems, some extensions would be necessary:

- an automatic derivative calculation routine should be build into the algebraic manipulator and equation solver modules,

- additional tools providing access to data bases should be available to the user,

- the set of available standard functions should be extended essentially.

The source code of this system is not available, therefore direct adaptation to decision support system is not possible. The concept of TK!Solver however is one of the most interesting to be adapted as user front end for decision support systems.

```
=================== VARIABLE SHEET ====================
St   Input      Name      Output     Unit       Comment
--   -----      ----      ------     ----       -------
                A         4.15
     3.2        B
     2.1        C
     3.5        D
     0.68       e
                f         0.00232082

====================== RULE SHEET =====================
S    Rule
-    ----
     A + B = C * D
     Sin(A - C) = Log(e/f)/(C + e)
=======================================================
```

Fig. 1 Sample screen generated by TK!Solver program.

4.3 Specialized programming languages.

It was recognized by many researchers, that in some cases application of general purpose formula manipulation languages like REDUCE or MACSYMA for solving rather simple problems requires too much effort and investments, especially if the user has no direct access to one of the above mentioned languages. Therefore some special software tools oriented for solving one specific class of problems were designed. One of such tools is HESQ - the Hierarchical Equation Solver developed by Derman and Van Wyk (1984), which, in fact, is a high level programming language oriented to solving the implicit nonlinear equations.

The HESQ system consists of the set of programs for interactive solving and debugging models described by set of algebraic equations, including their definition, examination and manipulation. Because models usually contain data dependent elements, HESQ permits the models

to include simple "vector" equations - i.e. equations whose left hand sides are variables with one running index that may take on several values. The following examples are taken from Derman's and Van Vyk's paper:

Income[1980:1984] = Revenue[1979:1983] + Extra

The above statement is equivalent to

Income[1908+i] = Revenue[1979+i] + Extra, $0 \leq i \leq 4$.

HESQ allows other useful shorthand notation, like

Vector [1:4] = [1, 2, alpha, -3]

or

Vector [1:4] = [1+3]

what generates the implicit arithmetic sequence [1,4,7,10].

Similarly, geometric and other sequences can be easily generated. Moreover, macros, array variables, conditions and IF - THEN - ELSE statements can be used, as well as rather broad set of standard functions and operators being at the disposal of the user. Complete example of a problem solved by HESQ system can be found in the paper quoted above.

The system lacks the derivative calculating features, but the nature of the algorithm used, which utilizes the graph representation of the model and dependencies between variables, makes necessary extensions possible.

A similar, simple to use modelling language implemented on the IBM-PC computer was described by Dunn (1983).

4.4 Extension of existing high level languages.

Another option, although not investigated until now, is an extension of existing programming languages to make them specially suitable for programming decision support systems. This can be achieved by proper modification of the syntax of language. This option will be discussed in the next section of the paper.

5. EXTENSION OF PASCAL FOR APPLICATIONS IN DECISION SUPPORT SYSTEMS.

5.1 General assumptions.

We shall follow here a general assumption that the user should have at his disposal a simple and flexible software tool for defining decision problems based on nonlinear models. This tool (or language) should possess following properties:

- it should be conceptually simple, even for user not being computer specialist,

- it should be sufficiently powerful to define even complicated decision problems,

- a friendly interface to the user should be provided, i.e., a user friendly environment for problem definition and analysis must be created,

- an interface to an optimization problem solver must constitute an integral part of the language; it means that all necessary formula manipulations must be performed without intervention of the user as well as that all information necessary for the solver to run an optimization problem must be available.

The solution presented here can be treated as **a model solution**, which is intended only to illustrate possible ways for resolving some basic problems and proposing further extensions. The proposed and implemented approach can however be applied to nontrivial practical problems without essential difficulties. Consequently, the resulting decision support system can be used as a prototype version for solving test problems.

5.2 Definition of the language.

The language is a very small subset of PASCAL, known as PL0 (Wirth, 1976), properly modified for our purposes. The basic features of this subset are as follows:

- the only available variable type is **real**, therefore there is no need to specify explicitly the variable type,

- there are no input and output features,

- the following standard constructs of PASCAL are available:

  - **const** and **var** declarations (however without type definition),
  - **if ... then** (no else!),
  - **procedure** without parameters,
  - **begin ... end** with standard PASCAL procedure nesting and visibility rules,
  - **while ... do**,
  - standard PASCAL statements and mathematical functions.

- the following important PASCAL features are not available in PL0:

  - records, pointers, arrays, sets, character, integers, strings and enumerated type,
  - user defined types,
  - repeat ... until,
  - file type, file i/o and standard i/o procedures,
  - else statement,
  - case selectors,
  - for loops.

There is also one more difference between PL0 and standard PASCAL - in order to call the procedure, **call** keyword must precede the procedure name.

The following are **extensions** to PL0 added here for the purpose

of applications in decision support systems:

- beside standard variables, **functional variables** are introduced. These variables are defined in **var** section of the program, just after declaration of standard variables. They are defined using the keywords

  vardec - for defining variables, which are the decision variables, i.e. their values will be set by the optimization routine (solver),

  varobj - for defining variables containing values of objective functions,

  varcon - for defining variables containing values of constraints functions.

There exist some additional rules for using structured variables. The **decision variables** can be used only on the right hand side of assignment statement, i.e. in expression. Their appearance on the left side of assignment statement is reported as compilation error. The **objective and constraints** variables can appear only on the left hand side of assignment statement. The attempt to use them in expression will also be reported as compilation error. They can be used in any place of the program (not necessary only once), but during the run time a value can be assigned to them **exactly once**. This is checked during the run time; an error message is reported if a value is assigned to an objective or constraint variable more than once or not assigned at all. Additionally, all functional variables must be defined as **global** ones, i.e. cannot be defined as local to any procedure.

5.3 Structure of the system.

The problem interface consists of the interactive text editor which can be used for defining and updating programs, saving and retrieving from disk etc. This part of the system is rather standard.

The compiler itself consists of the recursive top-down parser (Wirth 1976,) which converts the source code into sequence of commands of simple hypothetical stack machine. This concept was very broadly and successfully applied for compiler construction; for details see Davie and Morrison (1981) or Pemberton and Daniels (1982). The transformed program is interpreted by the procedure which emulates every command of this stack machine. Two interpreting routines are available in the system - one calculates values of all variables defined within the program, the second one - values of variables together with derivatives of all objective and constraint variables with respect to all decision variables.

The solver can invoke one of the interpreters. It depends on the current stage of optimization, which one should be invoked. Therefore the pseudocode for the procedure defining the optimization problem and called by the solver (optimization routine) should be the following (see Kreglewski at all., 1985, for details relating to interfacing with nonlinear solver):

```
Procedure Fun(X:InputArray, Var Y:ValueArray; Var Der:DerArray);
{-------------------------------------------------------------}
begin
  {Move X to Decision Variables Array}
  Case GlobFlag of
     ValueOnly           : ValInterprete;
     ValueAndDerivatives : begin
                             DifInterprete;
                            {Move Values of Derivatives
                             from  Objective and Constraints
                             Derivatives Array to Der}
                           end;
  end;
  {Move Values of Objectives and Constraints from Objective
   and Constraints Values Array to Y}
end;
```

In the above pseudocode, **GlobFlag** is the global parameter, which can take values from set (**ValueOnly**, **ValueAndDerivatives**) and is maintained by the solver.

5.4 The solver interface.

To use the system effectively, it is necessary to transform information from solver to interpreter (values of decision variables) and from interpreter to solver (values of objective and constraints rows and their derivatives. This can be achieved through the following data structure:

The following array must contain values of decision variables before invoking the interpreter:

```
DecArr  :Array[1..DecMax] of Real;
```

The following arrays will contain values of derivatives of objective and constraints variables after exit from interpreter:

```
ObjDer  :Array[1..ObjMax,1..DecMax] of Real;
ConDer  :Array[1..ConMax,1..DecMAx] of Real;
```

The following arrays will contain values of derivatives and constraints variables after exit from interpreter:

```
ConArr  :Array[1..ConMax] of Real;
ObjArr  :Array[1..ObjMax] of Real;
```

The following arrays will contain names of constraints and objective variables after successful exit from compiler. These names will appear in the same order, like in declaration section. This information can be utilized by man-machine interface for entering the reference point, defining right hand sides and types of constraints, displaying results, etc.

```
DecName :Array[1..DecMax] of Alf;
ObjName :Array[1..ObjMax] of Alf;
ConName :Array[1..ConMax] of Alf;
```

The following arrays contain Boolean flags which are set by interpreter to TRUE when value is assigned to objective or constraint variable. These variables can be inspected after exit from interpreter and error message or other action can be undertaken by the system depending on their status. Utilization of these arrays is at the disposal of system implementator.

```
ObjSet  :Array[1..ObjMax] of Boolean;
ConSet  :Array[1..ConMax] of Boolean;
```

All the above variables must be declared as global to solver, compiler and interpreter, i.e., declared in the block containing all these modules.

6. IMPLEMENTATION OF THE COMPILER AND INTERPRETER

This problem will not be analyzed here in details. Only the information necessary to understand changes in original PL0 compiler and to enable performing necessary changes in the compiler and interpreter will be discussed.

The following are extension to the compiler:

A. Extension of **Block** section, which allows declaration of functional variables. The following are the portions of code responsible for this task:

- Procedure SpecVarDeclaration, being the extension of VarDeclaration;

```
Procedure SpecVarDeclaration(VarType:Object);
{-----------------------------------------}
begin
     if LastSymRead=Ident then
        begin
             Enter(VarType);
             GetSym;
        end
        else
            Error(4);

end; {of SpecVarDeclaration}
```

- Extensions in variable declaration section of Block procedure;

```
if LastSymRead in [VarObjSym,VarConSym,VarDecSym] then
   if Lev<>0 then
      Error(33)
else
Repeat
      Case LastSymRead of
           VarObjSym : Lx:=ObjVariab;
           VarConSym : Lx:=ConVariab;
           VarDecSym : Lx:=DecVariab;
      end;
      GetSym;
      Repeat
            SpecVarDeclaration(Lx);
            While LastSymRead=comma do
```

```
                    begin
                         GetSym;
                         SpecVarDeclaration(Lx);
                    end;
               if LastSymRead=semicolon then
                  GetSym
               else
                  Error(5);
          Until LastSymRead<>Ident;
     Until not (LastSymRead in [VarObjSym,VarConSym,VarDecSym]);
```

B. Extension to Statement section, responsible for proper code generation and processing arithmetic expressions containing functional variables:

- extension of Enter procedure, which enters information about new variables to symbol table;

```
     Procedure Enter (k: object);
     {-------------------------}
     begin
          tx:=tx+1;
          with SymTable[tx] do
          begin
               Name:=LastIdRead;
               kind:=k;
               Case k of
                    Constant :begin
                                   if abs(LastNumRead)>AMax then
                                      begin
                                           Error(30);
                                           LastNumRead:=0;
                                      end;
                                   val:=LastNumRead;
                              end;
                    Variable :begin
                                   level:=lev;
                                   adr:=dx;
                                   dx:=dx+1;
                              end;
                    ObjVariab:begin
                                   level:=ObjMark;
                                   ObjPtr:=ObjPtr+1;
                                   ObjName[ObjPtr]:=Name;
                                   adr:=ObjPtr;
                              end;
                    ConVariab:begin
                                   level:=ConMark;
                                   ConPtr:=ConPtr+1;
                                   ConName[ConPtr]:=Name;
                                   adr:=ConPtr;
                              end;
                    DecVariab:begin
                                   level:=DecMark;
                                   DecPtr:=DecPtr+1;
                                   DecName[DecPtr]:=Name;
                                   adr:=DecPtr;
                              end;
```

```
                    Prozedure:level:=lev;
                end; {of Case k}
            end;
        end; {enter}
```

It should be pointed out, that for functional variables meaning of entries in Symbol Table is different, than for standard ones. Functional variables must always be **declared as global**, therefore the information usually contained in "level" field is not necessary in this case. This field was utilized for keeping information about the type of functional variable (DecMark, ConMark, ObjMark). These marks are negative, in order to make possible distinguishing between normal and functional variables during the interpretation phase. Analogically, standard address handling procedure is not applicable in this case. During the interpretation phase, values of functional variables are not saved in standard stack frame, but in arrays which play a role of solver interface (see previous section of the paper). Therefore separate address pointers for each type of variable were defined (DecPtr, ConPtr, ObjPtr). Values of these pointers are saved in "address" part of the symbol table entry. Both fields - i.e. "level" and "adr" contain full information about location and type of the variable.

- Modification of procedure Factor in the part responsible for code generation for loading a variable;

```
Procedure Factor(fsys: SymSet);
{----------------------------}
Var i: Integer;
    FctSym:Symbol;
begin
     Test(FacBegSys,fsys,24);
     While LastSymRead in FacBegSys do
           begin
                if LastSymRead=Ident then
                   begin
                        i:=position(LastIdRead);
                        if i=0 then
                            Error(11)
                        else
                        with SymTable[i] do
                             Case kind of
                                  Constant :Gen1(LIT,0,val);
                                  Variable :Gen0(LOD,lev-1,adr);
                                  DecVariab:Gen0(LOD,DecMark,adr);
                                  ObjVariab,
                                  ConVariab: begin
                                                  i:=0;
                                                  Error(34);
                                             end;
                                  Prozedure: Error(21);
                             end;
                        GetSym;
                   end
                   else
                   if LastSymRead=Number then ....
```

- Modification of the procedure Statement in the part responsible for code generation for storing functional variables;

```
begin {statement}
     if LastSymRead=Ident then
        begin
             i:=position(LastIdRead);
             if i=0 then
                Error(11)
             else
             if not (SymTable[i].kind in
                  [Variable,ObjVariab,ConVariab]) then
                begin
                     if SymTable[i].kind=DecVariab then
                        Error(35)
                     else
                        Error(12);
                     i:=0;
                end;
             GetSym;
             if LastSymRead=becomes then
                GetSym
             else
                Error(13);
             expression(fsys);
             if i<>0 then
                with SymTable[i] do
                     Case kind of
                          Variable :Gen0(STO,lev-level,adr);
                          ObjVariab:Gen0(STO,ObjMark,adr);
                          ConVariab:Gen0(STO,ConMark,adr);
                     end;
        end
        else
        if LastSymRead=CallSym then ....
```

It should be noted, that only for Objective and Constraint variables STOre code can be generated. This is not possible for Decision Variables. Analogically, only for Decision variable LOaD code can be generated. This ensures preservation of described above rules for usage and access of functional variables.

C. Extensions in definition of Stack Machine and interpreter.

The basic data structure of the Stack Machine is the stack, which is used as memory pool for all variables used within a program, and for all intermediate results occurring during the interpreting process. In the described extension, functional variables do not use stack. Instead, they utilize their own memory pools playing the role of solver interfaces. All other variables are located on the stack together with values of derivatives. Therefore the structure of the stack used by extended interpreter is the following:

```
s : Array[1..StacSiz] of Record
                              StacVal:Real;
                              StacDer:Array[1..DecMax] of Real;
                         end;
```

The set of commands of the Stack Machine remains unchanged, except of extension of STO and LOD commands:

```
LIT  0,a : load constant "a"
OPR  0,a : execute operation "a"
LOD  1,a : load variable, level "1", address "a"
LOD -1,a : load decision variable, address "a"
STO  1,a : store variable, level "1", address "a"
STO -2,a : store objective variable, address "a"
STO -3,a : store constraint variable, address "a"
CAL  1,a : call procedure at adress "a" and at level "1"
INT  0,a : increment t-register by "a"
JMP  0,a : jump to adress "a"
JPC  0,a : jump conditional to "a".
```

The only point requiring special treatment relates to the storing and loading the functional variables. All the interpreting routines relating to mathematical expressions must compute not only values of variables, but also values of derivatives. This can be done applying standard rules of differentiating mathematical expressions and elementary functions.

The full specification of the Stack Machine which was used in this particular implementation can be found in Wirth book (Wirth, 1976). The following is the source code of the interpreter, together with all extensions necessary for calculating derivatives. All the extensions responsible for calculating derivatives are printed in boldface.

The compiled program which is executed by the Stack Machine is stored in array Code declared as:

```
Instr = Record
          l: Integer;
          Case f:Fct of
               LIT : (LitVal:Real);
               OPR,
               LOD,
               STO,
               CAL,
               INT,
               JMP,
               FUN,
               JPC : (a: Integer);
        end;

Code : Array[0..CxMax] of Instr;
```

The basic data structure of the interpreter is the stack, used as internal memory pool. The stack is declared as:

```
s : Array[1..StacSiz] of Record
                           StacVal:Real;
                           StacDer:Array[1..DecMax] of Real;
                         end;
```

The following is the source code of the interpreter:

```
Procedure DInterpret;
{------------------}
Var    b,p,t,i,j: Integer;
       Ins    : Instr;

Function Base(l: Integer): Integer;
{--------------------------------}
Var b1: Integer;
begin
     b1:=b;
     While l>0 do
          begin
               b1:=Trunc(s[b1].StacVal);
               l:=l-1;
          end;
     base:=b1;

end; {Base}

begin
  t:=0;
  b:=1;
  p:=0;
  s[1].StacVal:=0;
  s.StacVal:=0;
  s[3].StacVal:=0;
  for i:=1 to ObjMax do
      ObjSet[i]:=False;
  for i:=1 to ConMax do
      ConSet[i]:=False;
  Repeat
     Ins:=Code[p];
     p:=p+1;
        with Ins do
           Case f of
              LIT: begin
                     t:=t+1;
{Load number:}       s[t].StacVal:=LitVal;
                     for i:=1 to DecPtr do
                     s[t].StacDer[i]:=0;
                   end;
              OPR: Case a of
                     0: begin
                          t:=b-1;
                          p:=Trunc(s[t+3].StacVal);
                          b:=Trunc(s[t+2].StacVal);
                        end;
                     1: begin
{Change sign:}            s[t].StacVal:=-s[t].StacVal;
                          for i:=1 to DecPtr do
                              s[t].StacDer[i]:=-
                                s[t].StacDer[i];
                        end;
                     2: begin
 {Addition:}              t:=t-1;
                          for i:=1 to DecPtr do
```

```
                        s[t].StacDer[i]:=
                          s[t].StacDer[i]+s[t+1].StacDer[i];
                    s[t].StacVal:=
                      s[t].StacVal+s[t+1].StacVal;
                  end;
               3: begin
{Substraction:}     t:=t-1;
                    for i:=1 to DecPtr do
                        s[t].StacDer[i]:=
                          s[t].StacDer[i]-s[t+1].StacDer[i];
                    s[t].StacVal:=
                      s[t].StacVal-s[t+1].StacVal;
                  end;
               4: begin
{Multiplication:}   t:=t-1;
                    for i:=1 to DecPtr do
                        s[t].StacDer[i]:=
                          s[t+1].StacVal*s[t].StacDer[i]+
                          s[t].StacVal*s[t+1].StacDer[i];
                    s[t].StacVal:=
                      s[t].StacVal*s[t+1].StacVal;
                  end;
               5: begin
{Division:}         t:=t-1;
                    for i:=1 to DecPtr do
                        s[t].StacDer[i]:=
                          (s[t+1].StacVal*s[t].StacDer[i]-
                           s[t].StacVal*s[t+1].StacDer[i])
                           /Sqr(s[t+1].StacVal);
                    s[t].StacVal:=
                      s[t].StacVal/s[t+1].StacVal
                  end;
               6: s[t].StacVal:=0;
               8: begin
                    t:=t-1;
                    s[t].StacVal:=
                      ord(s[t].StacVal=s[t+1].StacVal);
                  end;
               9: begin
                    t:=t-1;
                    s[t].StacVal:=
                      ord(s[t].StacVal<>s[t+1].StacVal);
                  end;
              10: begin
                    t:=t-1;
                    s[t].StacVal:=
                      ord(s[t].StacVal<s[t+1].StacVal);
                  end;
              11: begin
                    t:=t-1;
                    s[t].StacVal:=
                      ord(s[t].StacVal>=s[t+1].StacVal);
                  end;
              12: begin
                    t:=t-1;
                    s[t].StacVal:=
                      ord(s[t].StacVal>s[t+1].StacVal)
                  end;
```

```
                    13: begin
                          t:=t-1;
                          s[t].StacVal:=
                            ord(s[t].StacVal<=s[t+1].StacVal);
                        end;
                  end;
            LOD: if l>=0 then
                    begin
{Load variable:}      t:=t+1;
                      s[t].StacVal:=s[base(l)+a].StacVal;
                      for i:=1 to DecPtr do
                          s[t].StacDer[i]:=s[base(l)+a].StacDer[i];
                    end
                    else
                    begin
                      t:=t+1;
                      s[t].StacVal:=DecArr[a];
                      for i:=1 to DecPtr do
                          if i=a then
                             s[t].StacDer[i]:=1
                          else
                             s[t].StacDer[i]:=0;
                    end;
            STO: if l>=0 then
                    begin
{Store Variable:}     s[base(l)+a].StacVal:=s[t].StacVal;
                      for i:=1 to DecPtr do
                          s[base(l)+a].StacDer[i]:=s[t].StacDer[i];
                          t:=t-1;
                    end
                    else
                    Case l of
                       ObjMark :begin
                                  ObjArr[a]:=s[t].StacVal;
                                  ObjSet[a]:=True;
                                  for i:=1 to DecPtr do
                                     ObjDer[a][i]:=s[t].StacDer[i];
                                  t:=t-1;
                                end;
                       ConMark :begin
                                  ConArr[a]:=s[t].StacVal;
                                  ConSet[a]:=True;
                                  for i:=1 to DecPtr do
                                     ConDer[a][i]:=s[t].StacDer[i];
                                  t:=t-1;
                                end;
                    end;
            CAL: begin
                   s[t+1].StacVal:=base(l);
                   s[t+2].StacVal:=b;
                   s[t+3].StacVal:=p;
                   b:=t+1;
                   p:=a;
                 end;
            INT: t:=t+a;
            JMP: p:=a;
            JPC: begin
                   if s[t].StacVal=0 then
```

```
                        p:=a;
                      t:=t-1;
                    end;
              FUN: begin
{Standard funct.}     j:=a-Ord(SinProc);
                      Case j of
{Sin:}                   0 : begin
                               for i:=1 to DecPtr do
                                  s[t].StacDer[i]:=
                                    cos(s[t].StacVal)*s[t].StacDer[i];
                               s[t].StacVal:=sin(s[t].StacVal);
                             end;
{Cos:}                   1 : begin
                               for i:=1 to DecPtr do
                                  s[t].StacDer[i]:=
                                   -sin(s[t].StacVal)*s[t].StacDer[i];
                               s[t].StacVal:=cos(s[t].StacVal);
                             end;
{Ln:}                    2 : begin
                               for i:=1 to DecPtr do
                                  s[t].StacDer[i]:=
                                    1.0/s[t].StacVal*s[t].StacDer[i];
                               s[t].StacVal:=ln(s[t].StacVal);
                             end;
{Log:}                   3 : begin
                               for i:=1 to DecPtr do
                                  s[t].StacDer[i]:=
                                    1.0/s[t].StacVal*s[t].StacDer[i]
                                    /ln(10);
                               s[t].StacVal:=ln(s[t].StacVal)/ln(10);
                             end;
{Exp:}                   4 : begin
                               for i:=1 to DecPtr do
                                  s[t].StacDer[i]:=
                                    exp(s[t].StacVal)*s[t].StacDer[i];
                               s[t].StacVal:=exp(s[t].StacVal);
                             end;
{Sqrt:}                  5 : begin
                               for i:=1 to DecPtr do
                                  s[t].StacDer[i]:=
                                    1.0/2/sqrt(s[t].StacVal)
                                    *s[t].StacDer[i];
                               s[t].StacVal:=sqrt(s[t].StacVal);
                             end;
{Abs:}                   6 : begin
                               for i:=1 to DecPtr do
                                 if s[t].StacVal<=0 then
                                   s[t].StacDer[i]:=-s[t].StacDer[i];
                               StacVal:=abs(s[t].StacVal);
                             end;
                      end;
                    end;
                  end;
 Until p=0;

end; {of Dinterpret}
```

It should be rather clear from analysis of the above code, that in the fact the table algorithm was applied for calculating derivatives. It is necessary to mention additionally, that "illegal" procedure was applied for calculating the derivative of **abs** function. This is however the user´s responsibility to ensure, that an expression containing **abs** function is differentiable.

7. EXTENSIONS

The software tool described above is rather a simple and a straightforward approach to the problem of building interface for defining decision and optimization problems described by nonlinear models. Several improvements and extensions are necessary, both to improve the efficiency of the proposed interface, and for extending class of problems which could be solved utilizing this approach. The following are the problems which could be investigated:

- Improvement of the efficiency. In the existing implementation **all** formulas entered to the system are differentiated, independently, whether this is necessary, or not. Evidently, in some cases this is redundant - e.q. derivatives should not be calculated when computing logical conditions in **while** or **if** statements. This can be achieved rather easily, by extending definition of stack machine instruction:

```
Instr = Record
          DerCalc:Boolean;
          l: Integer;
          Case f:Fct of
               LIT : (LitVal:Real);
               OPR,
               LOD,
               STO,
               CAL,
               INT,
               JMP,
               FUN,
               JPC : (a: Integer);
        end;
```

where **DerCalc** is boolean flag set by compiler to False, when calculation of derivatives is not necessary. This flag must be tested during interpreting phase; according to its value calculation of derivatives can be skipped:

```
4 : begin
      if DerCalc then
         for i:=1 to DecPtr do
            s[t].StacDer[i]:=
            exp(s[t].StacVal)*s[t].StacDer[i];
      s[t].StacVal:=exp(s[t].StacVal);
    end;
```

Evidently, some rather trivial changes in code generation procedures (Gen0 and Gen1) must be performed.

It can be, however, rather difficult to perform more deep optimization of the calculation of derivatives. This is caused by the fact, that it is not possible to analyze the dependencies between

variables defined within a program without making analysis of all possible passes of control. This is especially difficult (or even impossible) using the recursive descent, one pass compiler. The following is an illustration of this difficulty:

```
var    n,...
vardec x,...
...
procedure p1;
begin
  n:=n+1;
end;
procedure p2;
begin
  if a>b then
     n:=x
  else
     n:=1;
end;
call p2;
call p1;
```

In the above example it is not possible to decide, whether statement n:=n+1 should be differentiated or not, without knowing the possible values of a and b. Moreover, when compiling the procedure p1 it is not possible to know in advance about the dependence of variable n on decision variable x. Therefore, global analysis of the program structure is necessary.

The easiest possible way to overcome this difficulty, is to incorporate some tools into language, which would make possible direct control by the user, which statements should be differentiated. This could be achieved by introducing a new class of functional variables - namely, the **intermediate variables**. They could be declared by **varint** declaration. Using this variables, the following rules could be established:

- all statements, which their left hand side of assignment instruction are functional variables, are differentiated,

- all statements, which their left hand side are ordinary variables (declared by **var**) and contain functional variables in right hand side part of assignment instruction are treated as illegal. Such situation is detected and reported during compilation phase,

- all statements, which do not contain functional variables are not differentiated.

The above rules can be used easily for deciding about necessary value of the above mentioned DerCalc flag.

The other possible improvement of efficiency can be achieved by applying more sophisticated algorithm for formula differentiation. The computational effort, necessary for gradient calculation was recently analyzed by Kim and others (1984). Similar remarks relating to this problem were given by Wolfe (1982).

The algorithm applied in current implementation requires for gradient computation the effort, which can be approximately estimated as **k*n**, where **k** is the cost of calculation of function, and **n** - number of decision variables. It was suggested by Wolfe and proven by Kim, that under proper arrangement of the calculation process, this effort can be reduced to **l*k**, where **l** is a small constant, not dependent on **n** (Wolfe suggests, that in most cases value of **l** is between 1.5 and 2.5). Rather essential reduction of computational effort can be expected when applying Kim's approach, especially for problems with many decision variables. This is however rather difficult to implement this algorithm - the complete expression tree must be known for generating the code for gradient evaluation. Therefore, a one pass compiler will probably be not a proper tool for implementing this algorithm or essential changes in parser structure and design should be necessary.

- Extension of PASCAL subset. The PL0 subset used for model implementation is rather extremely small subset of PASCAL, and many language features are missing. It seems, that arrays, structures, for loops and other constructs available in full sized PASCAL could be useful for advanced user.

The subset, which could be considered as an ideal compromise between simplicity and usability, is PASCAL-S. This subset, proposed by Wirth for educational applications (Wirth, 1981) can be easily implemented due to availability of the source code of compiler and interpreter. The general design of the compiler is the same, like for PL0 - the compiler generates code for stack machine, which is emulated by interpreting program. Therefore, the proposed approach for computing derivatives can be applied without essential difficulties.

- Extension of the class of problems. The PL0 subset was efficiently used for creating a language for simulation dynamic population models (Lewandowska, 1986). Further extension of this approach in this direction could be achieved by introducing new class of variables - **the state variables**. Combination of these two extensions - i.e. extension for dynamic simulation and one for automatic differentiation, could result in the system with automatic generation of conjugate equations for gradient calculation. This could simplify essentially solving decision problems described by dynamic models of differential equations or difference equations type.

8. REFERENCES.

Burnham W.D. and Hall A.R. (1985). Prolog - Programming and Applications. MacMillan Education Ltd.

Davie A.J.T and Morrison R. (1981). Recursive Descent Compiling. Ellis Horwood Publishers, Series: Computers and Their Applications, Vol. 14.

Derman E. and Van Wyk Ch. (1984). A Simple Equation Solver and its Application to Financial Modelling. Software-Practice and Experience, Vol. 14, December 1984.

Douglas B. (1982). Copernica Matematica - Scientists and Mathematicians Will Welcome the Precision of MU-MATH. 80 Microcomputing, June/July.

Dunn G. (1983). Mainframe to Micro: Adapting a Financial Modelling Language. The BYTE Journal, December 1983.

Fateman R.J. (1982). Symbolic Manipulation Languages and Numerical Computation: Trends. In: The Relationship Between Numerical Computation and Programming Languages, J.K. Reid, Ed., North-Holland.

Fourer R. (1983). Modelling Languages Versus Matrix Generators for Linear Programming. ACM Trans. on Math. Software, Vol. 9, No. 2, June 1983.

General Optimization Inc. (1986). What´s the Best - Linear Programming Through a 1-2-3 Worksheet Interface - User´s Manual.

Gentelman W.M. (1982). Programming Languages for Symbolic Algebra and Numerical Analysis. In: The Relationship Between Numerical Computation and Programming Languages, J.K. Reid, Ed., North-Holland.

Grauer M, Lewandowski A. and Wierzbicki A. (1984). DIDAS - Theory, Implementation and Experiences. in: M. Grauer and A. Wierzbicki, Eds.: Interactive Decision Making. Proceedings of an International Workshop on Interactive Decision Analysis and Interpretative Computer Inteligence, Lecture Notes in Economics and Mathematical Systems, Vol. 229, Springer Verlag, 1984.

Haynes J.L. (1985). Circuit Design with LOTUS 1-2-3. The BYTE Journal, Vol. 11, 1985.

Ince D.C. and Robson K. (1980). An Algol 68 Based Algebraic Manipulation System. Software-Practice and Experience, Vol. 10, pp. 427-430.

Kalaba R. and Tishler A. (1983). A Computer Program to Minimize a Function with Many Variables Using Computer Evaluated Exact Higher Order Derivatives. Appl. Mathematics and Comput. Vol. 13, pp. 143-172, 1983.

Kalaba R. and Spingarn K. (1984). Automatic Solution of Optimal Control Problems. I. Simplest Problem in the Calculus of Variation. Appl. Math. and Comp. Vol.14, pp. 131-158, 1984.

Kalaba R., Rasakhoo N. and Tishler R. (1983). Nonlinear Least Squares via Automatic Derivative Evaluation. Appl. Math. and Comp., Vol. 12, pp. 119-137, 1983.

Kalaba R. and Spingarn K. (1984). Automatic Solution of Optimal Control Problems. IV. Gradient Methods. Appl. Math. and Comput., Vol. 14, pp. 289-300, 1984.

Kaden S. and Grauer M. (1984). A Nonlinear Dynamic Interactive Decision Analysis and Support System (DIDAS-N): User´s Guide. WP-84-23, International Institute for Applied Syetems Analysis, Laxenburg, Austria.

Kedem G. (1980). Automatic Differentiation of Computer Programs. ACM Trans. on Math. Software, Vol. 6, No. 2, June 1980.

Kim K.V., Nesterov Yu.E. and Cherkaskij B.V. (1984). An Estimate of the Effort in Computing the Gradient. Soviet Math. Dokl., Vol. 29, No.2, American Math. Society, 1984.

Konopasek M. and Jayaraman S. (1985). Constraint and Declarative Languages for Engineering Applications: The TK!Solver Contribution. Proceedings of the IEEE, Vol. 73, No. 12, December 1985.

Kreglewski, T. and Kaden, S. (1985). Decision Support System MINE. Problem Solver for Nonlinear Multi-Criteria Analysis. International Institute for Applied Systems Analysis, Laxenburg, Austria.

Lewandowska A. (1986). POPMAN - the Interactive Program for Analysis of Dynamic Population Models. IIASA Working Paper, to appear.

Lewandowski, A., Kreglewski, T. and Rogowski, T. (1985). DIDAS-MZ and DIDAS-MM: the Trajectory-Oriented Extensions of DIDAS. In: Theory, Software and Test Examples for Decision Support Systems, A. Lewandowski and A. Wierzbicki, Eds. International Institute for Applied Systems Analysis, Laxenburg, Austria.

Lewandowski, A., Kreglewski, T. and Rogowski, T. (1985). DIDAS-NL - the Nonlinear Version of DIDAS System. In: Theory, Software and Test Examples for Decision Support Systems. A. Lewandowski and A. Wierzbicki, Eds. International Institute for Applied Systems Analysis, Laxenburg, Austria.

Microsoft (1983). MU-MATH/MU-SIMP System for MS-DOS.

Miller A.R. (1984). TK!Solver - A Tool for Scientists and Engineers. The Byte Journal, December, 1984.

Nicol R.L. (1981). Symbolic Differentiation a´la LISP. The BYTE Journal, September 1981.

Nori K.V., Amman U., Jensen K., Nageli H.H and Jacobi Ch. (1981). PASCAL-P Implementation Notes. In: PASCAL - The Language and its Implementation. D.W. Barron, Ed., John Willey & Sons, 1981.

Orchard-Hays W. (1978). DIF: Automatic Differentiation of Fortran Coded Polynomials. RM-78-45, Research Memorandum, International Institute for Applied Systems Analysis, Laxenburg, Austria.

Pemberton S. and Daniels M.C. (1982). PASCAL Implementation - The P4 Compiler. Ellis Horwood Ltd, Chichester, 1982.

Rall L.B. (1980). Application of Software for Automatic Differentiation in Numerical Computation. Computing, Suppl. 2, 1980.

Rall L.B. (1981). Automatic Differentiation: Techniques and Applications. Lecture Notes in Computer Science, Vol. 120. Springer Verlag.

Rall L.B. (1983). Differentiation and Generation of Taylor Coefficients in PASCAL-SC. In: A New Approach to Scientific Computation, U.W. Kulish and W.L. Miranker Ed., Academic Press, 1983.

Rall L.B. (1984). Differentiation in PASCAL-SC: Type GRADIENT. Softwa-

re-Practice and Experience. Vol. 10, No. 2, June 1984.

Rump S.M. (1983). Language Extension PASCAL-SC. In: A New Approach to Scientific Computation, U.W. Kulish and W.L. Miranker Ed., Academic Press, 1983.

Shearer J.M. and Wolfe M.A. (1985). AGLIB, a Simple Symbol Manipulation Package. Communications of the ACM, August 1985, Vol. 28, no. 8.

Tomovic R. and Vukobratovic M. (1972). General Sensitivity Theory. Elsevier.

Van de Riet R.P. (1973). ABC ALGOL - A Portable Language for Formula Manipulation Systems. Part 1: The Language, Part 2: The Compiler. Mathematical Centre Tracts, Amsterdam, 1973.

Van de Riet R.P. (1970). Formula Manipulation in ALGOL 60, Part 1 and Part 2. Mathematical Centre Tracts, Amsterdam, 1970.

Wengert R. (1964). A Simple Automatic Derivative Evaluation Program, Comm. of the ACM, Vol. 7, pp. 463-464, 1964.

Wierzbicki A. (1977). Models and Sensitivity of Control Systems. WNT Publ., Warsaw (in Polish). English edition - Elsevier, 1985.

Winston P.H. and Horn B.K.P. (1981). LISP. Addison-Wessley.

Wirth N. (1976). Algorithms + Data Structures = Programs. Prentice-Hall, Englewood Clifs, N.J., 1976.

Wirth N. (1981). PASCAL-S: A Subset and its Implementation. In: PASCAL - The Language and its Implementation. D.W. Barron, Ed., John Willey & Sons, 1981.

Wolfe P. (1982). Checking the Calculation of Gradients. ACM Trans. on Math. Software, Vol. 8, No. 4, December 1982.

Wolfram S. (1985). Symbolic Mathematical Computation. Communications of the ACM, Vol. 28, No. 4, April 1985.

APPENDIX

Several changes has been made in PL0 compiler published in Wirth book - real arithmetic was added, standard PASCAL functions were defined, as well as other changes relating to the application of the PL0 language as problem interface in decision support systems were performed. Moreover, the program was modified to be compiled by Turbo-Pascal. Therefore, for readers´s convenience, the modified code of the compiler is attached to the paper.

Before invoking the compiler, the program text must be located in the array declared as:

```
TextArray: Array[1..MaxNoOfLines] of String[80]
```

and number of lines of the program must be assigned to the integer variable NoOfLinesInBuff. Prior to calling the compiler, the CompInit procedure must be invoked. The user must supply the Error procedure which is responsible for error handling.

```
Const
      NoRw    = 21;       {no. of reserved Words}
      NoKw    = 14;       {no. of keywords}
      FctRw   = 15;       {first std. proc in keyword table}
      TxMax   = 100;      {length of identifier table}
      NMax    = 24;       {max. no. of digits in numbers}
      AMax    = 1.0e35;   {maximum number}
      LevMax  = 8;        {maximum depth of block nesting}
      CxMax   = 500;      {size of Code Array}
      StacSiz= 50;
      DecMark= -1;
      ObjMark= -2;
      ConMark= -3;
      DecMax = 20;
      ObjMax = 10;
      ConMax = 10;

Type
    Symbol=
    (Null,Ident,Number,PlusOp,MinusOp,TimesOp,slash,oddsym,
    eql,neq,lss,leq,gtr,geq,lparen,rparen,comma,semicolon,
    period,becomes,BeginSym,EndSym,IfSym,ThenSym,
    WhileSym,DoSym,CallSym,ConstSym,VarSym,VarObjSym,VarDecSym,
    VarConSym,ProcSym,SinProc,CosProc,LnProc,LogProc,ExpProc,
    SqrtProc,AbsProc);

    Alf     = String[32];
    Object  = (Constant,Variable,DecVariab,ObjVariab,
               ConVariab,Prozedure,StdFunct);
    SymSet  = set of Symbol;
    Fct     = (LIT,OPR,LOD,STO,CAL,INT,JMP,JPC,FUN);    {functions}
    Instr = Record
                  l: Integer;
                  Case f:Fct of
```

```
                    LIT : (LitVal:Real);
                    OPR,
                    LOD,
                    STO,
                    CAL,
                    INT,
                    JMP,
                    FUN,
                    JPC : (a: Integer);        {displacement address}
          end;

Var
    LastCharRead : char;              {last character read}
    LastSymRead  : Symbol;            {last Symbol read}
    LastIdRead   : Alf;               {last identifier read}
    LastNumRead  : Real;              {last number read}
    CharCount    : Integer;           {character count}
    IntErrNo     : Integer;           {interpreter Error}
    CodeAlocIdx  : Integer;           {Code allocation index}

    s            : Array[1..StacSiz] of Record
                                          StacVal:Real;
                                          StacDer:Array[1..DecMax] of Real;
                                        end;

    Code    : Array[0..CxMax] of Instr;
    Word    : Array[1..NoRw]  of Alf;
    WSym    : Array[1..NoRw]  of Symbol;
    SSym    : Array[char]     of Symbol;
    MnCode  : Array[Fct]      of Alf;

    DeclBegSys,
    StatBegSys,
    StdFctSym,
    FacBegSys : SymSet;

    DecPtr,ObjPtr,ConPtr : Integer;

    LineNo:Integer;

    SymTable: Array[0..TxMax] of Record
                                   Name: Alf;
                                   Case kind:object of
                                        Constant: (val: Real);
                                        Variable,
                                        Prozedure:(level,adr: Integer)
                                 end;

{------------------------------------------ Solver Interface }

    DecArr  :Array[1..DecMax] of Real;
    ObjDer  :Array[1..ObjMax,1..DecMax] of Real;
    ConDer  :Array[1..ConMax,1..DecMAx] of Real;
    ConArr  :Array[1..ConMax] of Real;
    ObjArr  :Array[1..ObjMax] of Real;

    DecName :Array[1..DecMax] of Alf;
    ObjName :Array[1..ObjMax] of Alf;
```

```
    ConName :Array[1..ConMax] of Alf;

    ObjSet  :Array[1..ObjMax] of Boolean;
    ConSet  :Array[1..ConMax] of Boolean;

Procedure CompInit;
{-----------------}
Var i:Integer;
begin
      for LastCharRead:='A' to ';' do
          SSym[LastCharRead]:=Null;

      Word[ 1]:='BEGIN';      Word[ 2]:='CALL';
      Word[ 3]:='CONST';      Word[ 4]:='DO';
      Word[ 5]:='END';        Word[ 6]:='IF';
      Word[ 7]:='ODD';        Word[ 8]:='PROCEDURE';
      Word[ 9]:='THEN';       Word[10]:='VAR';
      Word[11]:='WHILE';      Word[12]:='VAROBJ';
      Word[13]:='VARDEC';     Word[14]:='VARCON';

      Word[15]:='SIN';        Word[16]:='COS';
      Word[17]:='LN';         Word[18]:='LOG';
      Word[19]:='EXP';        Word[20]:='SQRT';
      Word[21]:='ABS';

      WSym[ 1]:=BeginSym;     WSym[ 2]:=CallSym;
      WSym[ 3]:=ConstSym;     WSym[ 4]:=DoSym;
      WSym[ 5]:=EndSym;       WSym[ 6]:=IfSym;
      WSym[ 7]:=oddsym;       WSym[ 8]:=ProcSym;
      WSym[ 9]:=ThenSym;      WSym[10]:=VarSym;
      WSym[11]:=WhileSym;     WSym[12]:=VarObjSym;
      WSym[13]:=VarDecSym;    WSym[14]:=VarConSym;

      WSym[15]:=SinProc;      WSym[16]:=CosProc;
      WSym[17]:=LnProc;       WSym[18]:=LogProc;
      WSym[19]:=ExpProc;      WSym[20]:=SqrtProc;
      WSym[21]:=AbsProc;

      SSym['+']:=PlusOp;      SSym['-']:=MinusOp;
      SSym['*']:=TimesOp;     SSym['/']:=slash;
      SSym['(']:=lparen;      SSym[')']:=rparen;
      SSym['=']:=eql;         SSym[',']:=comma;
      SSym['.']:=period;
      SSym['<']:=lss;         SSym['>']:=gtr;
      SSym[';']:=semicolon;

      MnCode[LIT]:='LIT';     MnCode[OPR]:='OPR';
      MnCode[LOD]:='LOD';     MnCode[STO]:='STO';
      MnCode[CAL]:='CAL';     MnCode[INT]:='INT';
      MnCode[JMP]:='JMP';     MnCode[JPC]:='JPC';
      MnCode[FUN]:='FCT';

      DeclBegSys:=[ConstSym,VarSym,VarObjSym,VarConSym,VarDecSym,ProcSym];
      StatBegSys:=[BeginSym,CallSym,IfSym,WhileSym];
      StdFctSym :=[CosProc,SinProc,ExpProc,LogProc,LnProc,SqrtProc,AbsProc];
      FacBegSys :=[Ident,Number,lparen]+StdFctSym;

      ErrNo:=0;
```

```
       ErrLine:=0;
       IntErrNo:=0;
       LineNo:=1;
       CharCount:=0;
       CodeAlocIdx:=0;
       LastCharRead:=' ';
       DecPtr:=0;
       ObjPtr:=0;
       ConPtr:=0;

end; {of CompInit}

Procedure Compile;
{---------------}

Procedure Error(n:Integer);
{-----------------------}
begin
     if ErrLine=0 then
        begin
             ErrLine:=LineNo;
             ErrNo:=n;
        end;

end; {of Error}

Procedure GetSym;
{--------------}
Var i,j,k :Integer;
    a:Alf;
    v:Real;
     Procedure GetCh;
     {-------------}
     Var LineLength:Integer;
     begin
          LineLength:=Length(TextArr[LineNo]);
          if CharCount=LineLength then
             begin
                  if LineNo<=NoOfLinesInBuff then
                     begin
                          LineNo:=LineNo+1;
                          CharCount:=0;
                          LastCharRead:=' ';
                     end
                     else
                     begin
                          WriteLn('Program Incomplete');
                          Error(29);
                          Exit;
                     end;
             end
             else
             begin
                  CharCount:=CharCount+1;
                  LastCharRead:=TextArr[LineNo][CharCount];
                  If LastCharRead in ['a'..'z'] then
                     LastCharRead:=Chr(Ord(LastCharRead)-32);
             end;
```

```
        end {getch};

begin {GetSym}
      While LastCharRead = ' ' do
            GetCh;
      if LastCharRead in ['A'..'Z'] then
         begin
              a:='';
              Repeat
                    a:=a+LastCharRead;
                    GetCh;
              Until not (LastCharRead in['A'..'Z','0'..'9']);
              j:=0;
              For i:=1 to NoRw do
                  if a=Word[i] then
                     j:=i;
              if j<>0 then
                 LastSymRead:=WSym[j]
              else
                 LastSymRead:=Ident;
              LastIdRead:=a;
         end
         else
         if LastCharRead in ['0'..'9'] then
            begin
                 a:='';
                 k:=0;
                 LastSymRead:=Number;
                 Repeat
                       a:=a+LastCharRead;
                       k:=k+1;
                       GetCh;
                 Until not (LastCharRead in ['0'..'9','.']);
                 if LastCharRead='E' then
                    begin
                         a:=a+LastCharRead;
                         k:=k+1;
                         GetCh;
                         Repeat
                               a:=a+LastCharRead;
                               k:=k+1;
                               GetCh;
                         Until not (LastCharRead in ['0'..'9','-','+']);
                    end;
                 Val(a,LastNumRead,i);
                 if (k>NMax) or (i<>0) then
                    Error(30);
            end
            else
            if LastCharRead=':' then
               begin
                    GetCh;
                     if LastCharRead='=' then
                        begin
                             LastSymRead:=becomes;
                             GetCh;
                        end
```

```
                end
                else
                if LastCharRead='<' then
                begin
                     GetCh;
                     if LastCharRead='>' then
                        begin
                             LastSymRead:=neq;
                             GetCh;
                        end
                        else
                        if LastCharRead='=' then
                           begin
                                LastSymRead:=leq;
                                GetCh;
                           end
                           else
                               LastSymRead:=lss;
                end
                else
                if LastCharRead='>' then
                   begin
                        GetCh;
                        if LastCharRead='=' then
                           begin
                                LastSymRead:=geq;
                                GetCh;
                           end
                           else
                               LastSymRead:=gtr;
                   end
                   else
                   begin
                        LastSymRead:=SSym[LastCharRead];
                        GetCh;
                   end;
 end {GetSym};

Procedure Gen0(x:Fct;y,z:Integer);
{-------------------------------}
begin
     if CodeAlocIdx>CxMax then
        Error(1000)
     else
     begin
          With Code[CodeAlocIdx] do
               begin
                    f:=x;
                    l:=y;
                    a:=z;
               end;
          CodeAlocIdx:=CodeAlocIdx+1;
     end;

end; {of Gen0}

Procedure Gen1(x:Fct;y:Integer;z:Real);
{-----------------------------------}
```

```
begin
     if CodeAlocIdx>CxMax then
        Error(1000)
     else
     begin
          With Code[CodeAlocIdx] do
               begin
                    f:=x;
                    l:=y;
                    LitVal:=z;
               end;
          CodeAlocIdx:=CodeAlocIdx+1;
     end;

end; {of Gen1}

Procedure Test(s1,s2:SymSet;n:Integer);
{-----------------------------------}
begin
     if not (LastSymRead in s1) then
        begin
             Error(n);
             s1:=s1+s2;
             While not (LastSymRead in s1) do
                  GetSym;
        end;

end; {of Test}

Procedure Block(lev,tx:Integer;fsys:SymSet);
{-----------------------------------------}
Var dx:Integer;
    i,tx0:Integer;
    cx0,xx0:Integer;
    Lx:Object;

    Procedure Enter(k:object);
    {-----------------------}
    begin
         tx:=tx+1;
         with SymTable[tx] do
         begin
              Name:=LastIdRead;
              kind:=k;
              Case k of
                   Constant :begin
                                  if abs(LastNumRead)>AMax then
                                     begin
                                          Error(30);
                                          LastNumRead:=0;
                                     end;
                                  val:=LastNumRead;
                             end;
                   Variable :begin
                                  level:=lev;
                                  adr:=dx;
                                  dx:=dx+1;
                             end;
```

```
                    ObjVariab:begin
                                  level:=ObjMark;
                                  ObjPtr:=ObjPtr+1;
                                  ObjName[ObjPtr]:=Name;
                                  adr:=ObjPtr;
                              end;
                    ConVariab:begin
                                  level:=ConMark;
                                  ConPtr:=ConPtr+1;
                                  ConName[ConPtr]:=Name;
                                  adr:=ConPtr;
                              end;
                    DecVariab:begin
                                  level:=DecMark;
                                  DecPtr:=DecPtr+1;
                                  DecName[DecPtr]:=Name;
                                  adr:=DecPtr;
                              end;
                    Prozedure:level:=lev;
               end; {of Case k}
          end;

    end; {enter}

Function Position(id:Alf):Integer;
{------------------------------}
Var i:Integer;
begin
     SymTable[0].Name:=id;
     i:=tx;
     While SymTable[i].Name <>id do
          i:=i-1;
     Position:=i;

end; {of Position}

Procedure ConstDeclaration;
{------------------------}
Var Sgn:Symbol;
begin
     if LastSymRead=Ident then
        begin
             GetSym;
             if LastSymRead in [eql, becomes] then
                begin
                     if LastSymRead=becomes then
                        Error(1);
                     GetSym;
                     if LastSymRead=Number then
                        begin
                             Enter(Constant);
                             GetSym;
                        end
                        else
                        if LastSymRead in [PlusOp,MinusOp] then
                           begin
                                Sgn:=LastSymRead;
                                GetSym;
```

```
                                    if LastSymRead=Number then
                                       begin
                                            if Sgn=MinusOp then
                                               LastNumRead:=-LastNumRead;
                                            Enter(Constant);
                                            GetSym;
                                       end
                                       else
                                           Error(2);
                                end
                                else
                                    Error(2);
                        end
                        else
                            Error(3);
                end
                else
                    Error(4);

end; {of ConstDeclaration}

Procedure VarDeclaration;
{----------------------}
begin
     if LastSymRead=Ident then
        begin
             Enter(Variable);
             GetSym;
        end
        else
            Error(4);

end; {of VarDeclaration}

Procedure SpecVarDeclaration(VarType:Object);
{-------------------------------------------}
begin
     if LastSymRead=Ident then
        begin
             Enter(VarType);
             GetSym;
        end
        else
            Error(4);

end; {of SpecVarDeclaration}

Procedure Statement(fsys:SymSet);
{-----------------------------}
Var i,cx1,cx2: Integer;
    Procedure Expression(fsys:SymSet);
    {-------------------------------}
    Var addop: symbol;
        Procedure Term(fsys:SymSet);
        {--------------------------}
        Var mulop: symbol;
            Procedure Factor(fsys:SymSet);
            {----------------------------}
```

```
Var i: Integer;
    FctSym:Symbol;
begin
  Test(FacBegSys, fsys,24);
  While LastSymRead in FacBegSys do
     begin
       if LastSymRead=Ident then
          begin
            i:=position(LastIdRead);
            if i=0 then
               Error(11)
            else
            with SymTable[i] do
                 Case kind of
                      Constant : Gen1(LIT,0,val);
                      Variable : Gen0(LOD,lev-level,adr);
                      DecVariab: Gen0(LOD,DecMark,adr);
                      ObjVariab,
                      ConVariab: begin
                                      i:=0;
                                      Error(34);
                                 end;
                      Prozedure: Error(21);
                 end;
                 GetSym;
          end
          else
          if LastSymRead=Number then
             begin
               if abs(LastNumRead)>AMax then
                  begin
                    Error(30);
                    LastNumRead:=0;
                  end;
                Gen1(LIT,0,LastNumRead);
                GetSym;
             end
             else
             if LastSymRead=lparen then
                begin
                  GetSym;
                  expression([rparen]+fsys);
                  if LastSymRead=rparen then
                     GetSym
                  else
                     Error(22);
                end
                else
                if LastSymRead in StdFctSym then
                   begin
                     FctSym:=LastSymRead;
                     GetSym;
                     if LastSymRead=lparen then
                        begin
                          GetSym;
                          expression([rparen]+fsys);
                          if LastSymRead=rparen then
                             begin
```

```
                                        GetSym;
                                        Gen0(FUN,1,ord(FctSym));
                                    end
                                    else
                                        Error(22);
                                end
                                else
                                    Error(23);
                            end;
                            test(fsys,[lparen],23);
                end;

        end; {of factor}

    begin {Term}
          factor(fsys+[TimesOp,slash]);
          While LastSymRead in [TimesOp,slash] do
                begin
                     mulop:=LastSymRead;
                     GetSym;
                     factor(fsys+[TimesOp,slash]);
                     if mulop=TimesOp then
                        Gen0(OPR,0,4)
                     else
                        Gen0(OPR,0,5);
                end;
     end; {of Term}

begin {expression}
      if LastSymRead in [PlusOp, MinusOp] then
         begin
              addop:=LastSymRead;
              GetSym;
              term(fsys+[PlusOp, MinusOp]);
              if addop=MinusOp then
                 Gen0(OPR,0,1);
         end
         else
              term(fsys+[PlusOp,MinusOp]);
       While LastSymRead in [PlusOp,MinusOp] do
             begin
                  addop:=LastSymRead;
                  GetSym;
                  term(fsys+[PlusOp,MinusOp]);
                  if addop=PlusOp then
                     Gen0(OPR,0,2)
                  else
                     Gen0(OPR,0,3);
             end;
end; {of Expression}

Procedure Condition(fsys:symset);
{----------------------------}
Var relop:symbol;
begin
     expression([eql,neq,lss,gtr,leq,geq]+fsys);
     if not (LastSymRead in [eql,neq,lss,leq,gtr,geq]) then
        Error(20)
```

```
          else
          begin
               relop:=LastSymRead;
               GetSym;
               expression(fsys);
               Case relop of
                    eql:Gen0(OPR,0,8);
                    neq:Gen0(OPR,0,9);
                    lss:Gen0(OPR,0,10);
                    geq:Gen0(OPR,0,11);
                    gtr:Gen0(OPR,0,12);
                    leq:Gen0(OPR,0,13);
               end;
          end;

     end; {of Condition}

begin {statement}
       if LastSymRead=Ident then
          begin
               i:=position(LastIdRead);
               if i=0 then
                  Error(11)
               else
               if not (SymTable[i].kind in
                       [Variable,ObjVariab,ConVariab]) then
                  begin
                       if SymTable[i].kind=DecVariab then
                          Error(35)
                       else
                           Error(12);
                       i:=0;
                  end;
               GetSym;
               if LastSymRead=becomes then
                  GetSym
               else
                  Error(13);
               expression(fsys);
               if i<>0 then
                  with SymTable[i] do
                       Case kind of
                            Variable :Gen0(STO,lev-level,adr);
                            ObjVariab:Gen0(STO,ObjMark,adr);
                            ConVariab:Gen0(STO,ConMark,adr);
                       end;
          end
          else
          if LastSymRead=CallSym then
             begin
               GetSym;
               if LastSymRead<>Ident then
                  Error(14)
               else
               begin
                    i:=position(LastIdRead);
                    if i=0 then
                       Error(11)
```

```
            else
            with SymTable[i] do
                  if kind=Prozedure then
                     Gen0(CAL,lev-level,adr)
                  else
                     Error(15);
            GetSym;
      end;
end
else
if LastSymRead=IfSym then
   begin
        GetSym;
        condition([ThenSym,DoSym]+fsys);
        if LastSymRead=ThenSym then
           GetSym
        else
           Error(16);
        cx1:=CodeAlocIdx;
        Gen0(JPC,0,0);
        statement(fsys);
        Code[cx1].a:=CodeAlocIdx;
   end
   else
   if LastSymRead=BeginSym then
      begin
           GetSym;
           statement([semicolon,EndSym]+fsys);
           While LastSymRead in [semicolon]+StatBegSys do
                 begin
                      if LastSymRead=semicolon then
                         GetSym
                      else
                         Error(10);
                      statement([semicolon,EndSym]+fsys);
                 end;
           if LastSymRead=EndSym then
              GetSym
           else
              Error(17);
      end
      else
      if LastSymRead=WhileSym then
         begin
              cx1:=CodeAlocIdx;
              GetSym;
              condition([DoSym]+fsys);
              cx2:=CodeAlocIdx;
              Gen0(JPC,0,0);
              if LastSymRead=DoSym then
                 GetSym
              else
                 Error(18);
              statement(fsys);
              Gen0(JMP,0,cx1);
              Code[cx2].a:=CodeAlocIdx;
         end;
       test(fsys,[],19);
```

```
end; {of Statement}

begin {block}
      dx:=3;
      tx0:=tx;
      SymTable[tx].adr:=CodeAlocIdx;
      Gen0(JMP,0,0);
      if lev > LevMax then Error(32);
      Repeat
            if LastSymRead=ConstSym then
               begin
                    GetSym;
                    Repeat
                          ConstDeclaration;
                          While LastSymRead=comma do
                                begin
                                     GetSym;
                                     ConstDeclaration;
                                end;
                          if LastSymRead=semicolon then
                             GetSym
                          else
                             Error(5);
                     Until LastSymRead<>Ident;
               end;
                if LastSymRead=VarSym then
                   begin
                        GetSym;
                        Repeat
                              VarDeclaration;
                              While LastSymRead=comma do
                                    begin
                                         GetSym;
                                         VarDeclaration;
                                    end;
                              if LastSymRead=semicolon then
                                 GetSym
                               else
                                 Error(5);
                        Until LastSymRead<>Ident;
                   end;
               if LastSymRead in [VarObjSym,VarConSym,VarDecSym] then
                   if Lev<>0 then
                      Error(33)
                   else
                   Repeat
                        Case LastSymRead of
                             VarObjSym:Lx:=ObjVariab;
                             VarConSym:Lx:=ConVariab;
                             VarDecSym:Lx:=DecVariab;
                        end;
                        GetSym;
                        Repeat
                              SpecVarDeclaration(Lx);
                              While LastSymRead=comma do
                                    begin
                                         GetSym;
```

```
                                        SpecVarDeclaration(Lx);
                                    end;
                                if LastSymRead=semicolon then
                                   GetSym
                                 else
                                   Error(5);
                            Until LastSymRead<>Ident;
                        Until not (LastSymRead in
                                VarObjSym,VarConSym,VarDecSym]);
                    While LastSymRead=ProcSym do
                        begin
                            GetSym;
                            if LastSymRead=Ident then
                               begin
                                    enter(Prozedure);
                                    GetSym;
                               end
                               else
                                   Error(4);
                               if LastSymRead=semicolon then GetSym
                                  else Error(5);
                               block(lev+1,tx,[semicolon]+fsys);
                               if LastSymRead=semicolon then
                                   begin
                                     GetSym;
                                     Test(StatBegSys+[Ident,ProcSym],fsys,6)
                                   end
                                   else
                                       Error(5);
                        end;
                    Test(StatBegSys+[Ident],DeclBegSys,7);
        Until not (LastSymRead in DeclBegSys);
        Code[SymTable[tx0].adr].a:=CodeAlocIdx;
        SymTable[tx0].adr:=CodeAlocIdx;
        cx0:=CodeAlocIdx;
        Gen0(INT,0,dx);
        statement([semicolon,EndSym]+fsys);
        Gen0(OPR,0,0);
        Test(fsys,[ ],8);

end; {of block}

begin
        GetSym;
        block(0,0,[period]+DeclBegSys+StatBegSys);
        if LastSymRead<>period then
           Error (9);

end; {of Compile}
```