REFERENCE MANUAL FOR THE

LPL MODELING LANGUAGE

****

(Version 3.5)

Tony Hürlimann

Working Paper No. 175

June 1990

INSTITUTE FOR AUTOMATION AND OPERATIONS RESEARCH

University of Fribourg

CH-1700 Fribourg / Switzerland

Bitnet: HURLIMANN@CFRUNI51

phone: (41) 37 21 95 60 fax: 37 21 97 03

Till Kajsas födelsedag

Acknowledgment

I would like to thank Paul Langenegger who pushed the LPL compiler and especially the report generator to its limits and discovered many bugs. I have also received many hints in numerous discussions from Pius Hättenschwiler who is our 'chief-modeler' and from Marco Moresino as well as from Harald Häuschen. They all work with LP/MIP real-live models with thousends of variables and restrictions.

Table of contents

I think that I shall never see

A poem lovely as a tree

-- JOYCE KILMER (1913)

INTRODUCTION

" INTRODUCTION

This paper is the Reference Manual for the linear programming model language LPL (**L**inear **P**rogramming **L**anguage). It can also be used as a Tutorial Manual, and it contains some examples as well.

1.1. WHAT IS LPL?

LPL is a mathematical modeling language with a powerful index mechanism, which allows one to build, maintain, adapt and document large LP-models and other mathematical models. It allows to create automatically different input files - like the MPS-Standard file - for an optimization software package. LPL contains also a powerful Report Generator. With LPL it is possible to build almost any LP-Model in a very convenient and fast way.
The main features of LPL are:

- a simple syntax of models with indexed expressions close to the mathematical notation, and directly applicable for documentation
- formulation of both small *and* large LP's with optional separation of the data from the model structure
- availability of a powerful tree index mechanism, making model structuring very flexible
- an innovative and powerful Report Generator
- intermediate indexed expression evaluation
- automatic or user-controlled production of row- and column-names
- tools for debugging the model (e.g. explicit equation listing)
- built-in text editor to enter the LPL model
- fast production of the MPS file
- open interface to any LP and MIP solver packages.

The presently implemented LPL compiler does not contain:

- problem analysis with the aid of the graph theory
- solution procedures for optimization

1.2. BACKGROUND

The development of the LPL language was motivated by practical modeling tasks such as model building, modification, and documentation for large LP models. Many

important elements of LPL have been created from practical modeling experience. The model used for planning alimentary self sufficiency in Switzerland, for example, contains about 2000 variables and 1700 restrictions and the matrix has about 8000 non-zero items. The entire model has been defined in a LPL.

For a long time, matrix generators have been the predominant tools to produce standard input files for (large) linear programs for computers. Some are widely used, while others have been built for special models only.

In this manual an alternative tool is presented: an efficient and readable modeling language called LPL (**L**inear **P**rogramming **L**anguage). It allows problem formulation in terms familiar to an analyst. The fundamental idea of LPL is to write a LP in a concise, symbolic form, close to the algebraic notation for variables, constraints, and objectives, and to leave as much work as possible to the machine to translate this symbolic form into a coded form like the MPS file. A LPL Compiler, which performs this translation, is available. Integrated within the language is also a Report Generator, which writes all reports to a predefined file.

A LPL model is a complete and readable formulation of a LP problem and it can also be used for documentation and expression evaluation independently of any optimization software. The LPL Compiler is implemented in TURBO PASCAL 5.5 (actually distributed version) and TURBO C 2.0 (ANSI C) from Borland under MS/DOS. It produces an internal, restriction-sorted file, which can be used to create input data files for different optimization software packages (e.g., the MPS input file) or various readable output files such as equation-listings. The modules which produce these different output files are also available with the LPL Compiler.

LPL can also be used for any calculation with indexed expressions. An LPL model needs not to be a linear programming model. Any numerical data tables can be processed by LPL. The Report Generator lets the user print the tables in any format.

1.3. FORMAT OF THE MANUAL

Chapter 2 explains the implementation and use of the LPL-package on PCs (installation, use and the outputs). A quick tutorial guide is provided in Chapter 3. Chapters 4-10 give a detailed overview of the LPL language and its syntax. Chapter 4 describes the basic elements of LPL. Chapter 5 contains all the information about the structure of an LPL model. Indices and Indexlists, the most fundamental elements of LPL, are explained in detail in Chapter 6. Data and table formats of data in a model are explained in Chapter 7. Chapters 8 and 9 give further information about variables

and restrictions. The Report Generator of LPL is described in Chapter 10. Compiler error messages, the entire syntax description, and the Editor commands are listed in Appendices A-C. Appendix D provides some LPL model examples. More models can be found on the LPL disk, which can be bought from the author.

1.4. TYPOGRAPHY

Special character fonts are used for the following purposes:

underlining	underlining is used to define important LPL concepts
italics	Italics are used to emphasize sections of text
`Courier`	is used to illustrate LPL model examples and to print tables

Note that throughout the Manual we use the syntax description summarized in Appendix B.

1.5. WHAT TO DO NOW?

1. If you have the disk, read first the README file
2. Run the DEMO on the disk by typing 'DEMO'
3. If you are a new LPL user, read first Chapter 2 and 3 (Tutorial).
4. If you have the LPL disk, do the exercises in Chapter 3 (takes 30 min.)
5. Read Chapter 4 and 5 to get an idea of LPL
6. If you have already worked with LPL, read the UPDATE.TXT file.
7. If you have the disk, take a look to all model fragments LEARN*.LPL
8. Study the LPL examples in Appendix D

INSTALLING AND RUNNING LPL

In its present state, LPL is much like a UNIX tool with no integrated interactive modeling environment. The man-machine interface is minimal, with no graphics or mouse support. This is not - in our opinion - what a modeling environment should be. But we concentrated our attention on the development of the LPL language and its compiler. The whole of LPL is on a single 5 1/4 or 3 1/2 disk (IBM PC compatible format). It comes with the LPL compiler and other modeling tools, as well as a collection of about 40 LP model examples written in LPL.

2.1. THE FILES ON DISK

The programs (tools) are:

```
- LPL.EXE: the LPL compiler, which produces the .RES and the .NOM file
- LPLEQU.EXE: produces the .EQU file
- LPLMPS.EXE: produces the .MPS (for MPS) file
- LPLINT.EXE: produces the .INT file
- LPLPIC.EXE: shows a picture of the A matrix on the screen
- LPLEDIT.COM: the LPL standalone editor.
```

LPL examples:

```
- all files with extension .LPL
- all files with extension .INC (data files)
```

The solution (with XA from Sunset [13]) of some models:

```
- all files with extension .SOL
```

Some useful batch files:

```
- DEMO.BAT   (to show a short demo)
- PURGE.BAT  (erase all files produced by LPL tools except the *.LPL and *.INC)
```

Other files:

```
- LPL.MSG (the error message file of the LPL compiler and editor )
- LPL.CFG (contains solver interface parameters)
- README (read this file first!)
```

2.2. INSTALLING LPL

Installing LPL only requires copying the files from the original disk. No other manipulations are needed. At least the following two files must be copied:

```
LPL.EXE     (LPL compiler)
LPL.MSG     (The error message file)
```

2.3. A FIRST TEST

LPL is now ready to be used. The user may start an existing model included in the package, or create a new model. To execute the first test with LPL, type the following command at the DOS prompt:

```
lpledit product.lpl
```

Type the arrow, the <PgDn>, <PgUp>, <END>and <HOME> keys to scroll around

the model text. Type <CTRL>KX and Y to quit the editor without modifying the text. (Use <CTRL>KD to save all modifications). Then type at the DOS prompt:

```
lpl robot2
lpledit robot2.nom  (type <ctrl>KX and Y to quit the editor)
lplequ robot2
lpledit robot2.equ  (type <ctrl>KX and Y to quit the editor)
lplint robot2
lpledit robot2.int  (type <ctrl>KX and Y to quit the editor)
lplmps robot2
lpledit temp.mps    (type <ctrl>KX and Y to quit the editor)
```

Now type 'DEMO' to see a small demo of the LPL system.

2.4. THE TOOL LPL.EXE

LPL.EXE is the LPL compiler. The LPL compiler reads and compiles a model-file 'ModelFile>.LPL' written in LPL syntax (see Figure 1 on the next page).

LPL is called by the following syntax:

```
LPL <modelfile> [ Option ]
```

where 'modelfile' is the filename of the model (without extension). LPL.EXE assumes a filename extension of '.LPL'. 'Option' is empty or one character. If this character is

- 's' : Only the syntax of the modelfile is tested without any output.
- 'o' : Debug information is added to the .NOM file.
- 'n' : .RES is not produced, and {$S}, and {$X ...} are simply comments.
  (no child process is executed within LPL).
- any other character: Two files with extensions .RES and .NOM are produced.

Examples:

```
LPL mod1 s   (tests only the syntax of model 'mod1.lpl')
LPL test o   (produces a file 'test.nom' with debug information)
LPL model n  (compiles model without executing any child process)
LPL Model1   (produces the file 'Model1.res' from model 'Model1.lpl')
```

If a compile error occurs during the compilation, the error message file LPL.MSG is read - if present - and the error is reported to the screen and the compilation is aborted. Furthermore, a batch file ED.BAT is produced, which may be used to start the editor LPLEDIT.COM allowing the user to correct the error.

Before compiling <ModelFile>.LPL, the LPL compiler compiles the LPL.CFG file - if present. LPL.CFG must also be written in LPL syntax.

LPL produces the file '<ModelFile>.RES', a internal representation of the model, as well as the file 'TEMP.CLP' which contains the solver call parameters.

If the Compiler directive {$S} is present within the <ModelFile>.LPL, the LPL compiler calls automaticcally the LPLMPS process which produces the MPS standard input file 'TEMP.MPS' for a LP or MIP solver. Then the solver is called, which writes the solution to the 'TEMP.SOL' file. This file is read by the LPL compiler when the solver has finished his work. Then the report file <ModelFile>.NOM is written. The Figure 1 gives a overview of all these connections.

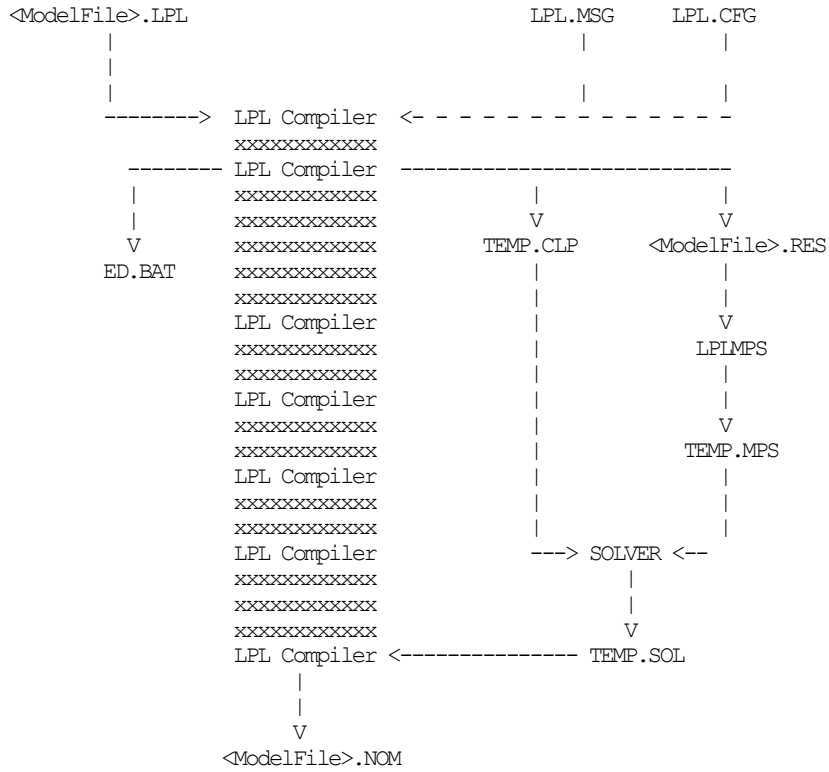


FIGURE 1

The file with extension .NOM is created each time LPL is started. It contains all model reports and - if 'option' is 'o' - it produces a list of all sets, coefficients, variables and restriction names in ASCII format and may be displayed or printed with the user's favorite word-processor.

The file with extension .RES is the most important link to the other tools. It represents the model, where all indices are expanded and all expressions are evaluated. It contains all restrictions explicitly, much like a full equation listing.

The .RES file is not an ASCII file and cannot be displayed or printed directly (to do this, use the tool LPLINT.EXE). The structure of the .RES file is a list of the following record in direct file PASCAL and C binary file format:

```
  PASCAL Version:                  C Version
RECORD                           struct {
   kind    : char;                  char kind;
   name    : string[8];             char name[9];
   NrId,nr : integer;               int NrId,nr;
   coef    : real;                  float coef;
end                              }
```

2.5. THE TOOL LPLINT.EXE

The syntax to execute LPLINT.EXE is:

```
LPLINT <modelfile>
```

This tool assumes that the file .RES of the model exists. If not, it must first be produced by the LPL compiler. No extension on the modelname is needed. LPLINT

produces a file with the same name but extension .INT. This file is the same as the file .RES but in ASCII format. The structure of this file contains lines with 5 entries per line:

a character, two positive integers, a text of up to 8 characters, and a real value.

The character at the beginning of the line can be

```
'M' : for a maximizing function
'N' : for a minimizing function
'E' : for a equal restriction
'G' : for a greater than restriction
'L' : for a less than restriction
'R' : for a real variable
'I' : for a integer variable
'e' : for a fix bound on a variable
'l' : for a lower bound on a variable
'g' : for a upper bound on a variable
```

The two integers identify the variable or the restriction. The first integer tells to which index-grouped variable this specific variable belongs. They are numbered, beginning with 1, in the order they are declared in the LPL model. The second integer indicates the major row position of this specific variable within the index-variables. If the variable is not indexed, this integer is put to 1.

The name is the internal variable or restriction name with a maximum of 8 characters. The composition of this name may be manipulated by the user with the {$N compiler directive or the alias-names (see Chapter 6.7 and 9.7).

The real value is the right hand side (RHS) of a restriction, the coefficient of a variable in a specified restriction, or the bound value.

A line beginning with 'M', 'N', 'E', 'L', or 'G' is always followed by at least two lines with kind 'R' or 'I'. This means that the corresponding variables are part of the specified restriction.

Example:

```
N 6 9 PROF     0
R 5 8 Z8       23
R 7 9 W9       24
E 1 1 RESNAME1 100
R 2 1 X1       1.2
R 2 2 X2       0.6788
G 2 1 RES2     1002
R 2 1 X1       2.3
I 2 3 X3       -1.2
l 2 3 X3       300
```

means (where 'x3' is an integer variable):

```
MINIMIZE PROF: 23*Z8 + 24*W9
RESNAME1: 1.2*X1 + 0.6788*X2 = 100
RES2: 2.3*X1 - 1.2*X3 > 1002
BOUND: X3 < 300
```

2.6. THE TOOL LPLEQU.EXE

The syntax to execute LPLEQU.EXE is:

```
LPLEQU <modelfile>
```

This tool assumes that the file .RES of the model exists. If not, it must first be produced by the LPL compiler. No extension on the modelname is needed. LPLEQU

produces a file with the same name but extension .EQU. This file is an explicit equation listing of the model in ASCII format. It may be printed or displayed with any word-processor (use a word-processor which has a line wrap function).

2.7. THE TOOL LPLMPS.EXE

The syntax to execute LPLMPS.EXE is:

```
LPLMPS <modelfile> [ <outputfile>]
```

This tool assumes that the file .RES of the model exists. If not, it must first be produced by the LPL compiler. No extension on the modelname is needed. LPLMPS produces the file TEMP.MPS by default, if the <outputfile> is not defined. This file is the known MPS input file representing the LP model. Most LP solvers accept MPS files as input. The definition of the format of this file may be found in the literature. Note that LPLMPS does not produce a RANGE section. BOUNDS are limited to FX, UP, and LO. The COLUMN section may also contain the necessary markers for integer variables.

2.8. THE TOOL LPLEDIT.COM

The syntax to execute LPLEDIT.COM is:

```
LPLEDIT <filename> [ <int1> <int2> <int3> ]
```

Note that this tool needs a filename (with extension) as parameter. LPLEDIT is a simple WORDSTAR-like editor. The user is free to use this or another text editor to enter a new model. Note, however, that the .LPL file must be in ASCII format (with no special characters indicating text formats), otherwise the LPL.EXE compiler will not be able to compile the model.

The advantage of LPLEDIT is that LPL.EXE will start it automatically when it finds a syntax error in the .LPL source model file and place the cursor where the error occurred. The LPLEDIT commands are listed in Appendix C.

The LPL editor can also be started with 3 more parameters which must be integers. <int1> is the error message read from the LPL.MSG file which will be displayed, <int2> and <int3> are the initial column and rows position of the cursor within the edit file.

2.9. THE TOOL LPLPIC.EXE

The syntax to execute LPLPIC.EXE is:

```
LPLPIC <modelname>
```

This tool assumes that the file .RES of the model exists. If not, it must first be produced by the LPL compiler. No extension on the modelname is needed. LPLPIC allows the user to browse through the matrix of the model at text and pixel level. This is especially useful for big models. You need a CGA compatible graphic screen. The variables are displayed as columns and the restriction as rows.

TUTORIAL

This chapter introduces the user to the most basic features of the LPL language in an informal way. A simple model example is used for this purpose. The user should carefully study the progressive complication of the model formulation. For the user's convenience, all reserved words in LPL are written entirely in capitals. All model formulations are available on disk as 'ROBOT?.LPL' files. This tutorial also contains some exercises. The user is invited to execute them directly on the computer while reading the tutorial. No solver is needed.

3.1. MODEL DESCRIPTION

A firm produces two type of robots called 'Marie' and 'Jules'. Three production steps must be carried out: a) Production of the components, which takes 5 hours for each robot Marie and Jules, with a total capacity of 350 hours per week; b) Mounting (capacity=480) taking 4 hours for Marie and 8 hours for Jules and c) Testing (capacity=300) taking 6 hours for Marie and 2 hours for Jules. The selling prices for Marie is $300 and for Jules $200. There are already 20 Maries and 15 Jules ordered. How many robots of each type can be produced per week, if the firm wants to maximize the selling profit?
If x and y are the unknown number of robots Marie and Jules, we can formulate this problem in the following mathematical way:

```
Components:        5x +   5y  <=  350
Mounting:          4x +   8y  <=  480
Testing:           6x +   2y  <=  300
Order1:             x         >=   20
Order2:                    y  >=   15
MAXIMIZE profit: 300x + 200y
```

3.2. THE SIMPLEST MODEL FORMULATION

This formulation can be translated directly to LPL as ('ROBOT1.LPL'):

```
VAR  x; y;
MODEL
  Components: 5*x + 5*y  <=  350;
  Mounting:   4*x + 8*y  <=  480;
  Testing:    6*x + 2*y  <=  300;
  Order1:       x        >=   20;
  Order2:             y  >=   15;
  MAXIMIZE profit: 300*x + 200*y;
END
```

- All variables must be declared before use. This is done by 'VAR x; y;'. VAR is the reserved word for introducing a variable list. Each variable must be followed by a

semicolon.

- After 'MODEL' a series of restrictions can be formulated in a way close to mathematical notation. Note that the multiplication sign is '*'. Each restriction begins with a name and a colon and ends with a semicolon.
- The objective function must begin with the reserved word MAXIMIZE or MINIMIZE depending on whether the function is to be maximized or minimized.
- The reserved word END ends the model formulation.

exercise: Type the following commands:

```
type robot1.lpl
lpledit robot1.lpl
  (add now '{$N1}' at the top of the file)
  (quit the editor with <ctrl>KD)
lpl robot1
lplequ robot1
type robot1.equ
```

3.3. NAMES AND COMMENTS

The same model with more explicit names and comments may be formulated as ('ROBOT2.LPL'):

```
(* variables *)
VAR Marie; Jules;  { Number of the two robots }

(* model *)
MODEL
  Components: 5*Marie + 5*Jules  <=  300+50;
  Mounting:   4*Marie + 8*Jules  <=  500-20;
  Testing:    6*Marie + 2*Jules  <=  30*10;
  Order1:       Marie            >=  200/10;
  Order2:                Jules   >=  4^1.95;
  MAXIMIZE profit: 300*Marie + 200*Jules;
{$Solve}
PRINT profit; Marie; Jules;
END
```

- The variables need not be 'x' or 'y'. They may have any name the user wants to give them (here 'Marie' and 'Jules').
- Comments can be added anywhere. They must be included within { ... } or within (* ... *). They help to document the model.
- Any number, such as 350, may be replaced by an arithmetical expression, like 300+50. The five operators introduced here are: '+' for addition, '-' for subtraction, '*' for multiplication, '/' for division and '^' for the exponent operator. Note that division by zero and exponent on a negative number yields an error.
- {$Solve} calls the solver and reads the results back to the LPL. If no solver is present this instruction reads only ROBOT2.SOL (the solution file) back to LPL.
- PRINT writes the results to the file ROBOT2.NOM.

exercise: Type the following commands:

```
type robot2.lpl
LPL robot2
type robot2.nom
lplequ robot2
lpledit robot2.equ  (type <ctrl>KX and Y to quit the editor)
```

3.4. INDICES

Now suppose we have not two but 10 different types of robots to produce. It would be annoying to write down 10 different variables. To simplify the formulation, we may use indices (also called sets or domains in LPL) ('ROBOT3.LPL').

```
SET  i = ( 1:10 );
VAR  Robots(i);
COEF HC(i) = / 5 5 4 5 6 5 7 8 4 7 /;
     HM(i) = / 4 8 5 6 4 8 7 6 5 3 /;
     HT(i) = / 6 2 4 6 3 4 5 2 5 3 /;
     Ordered(i)  = / 20 15 7 6 5 8 9 8 7 5 /;
     Price(i)  = / 300 200 100 50 50 100 200 100 400 200 /;
MODEL
  Components: SUM(i) HC(i)*Robots(i) <=  3500;
  Mounting:   SUM(i) HM(i)*Robots(i) <=  4800;
  Testing:    SUM(i) HT(i)*Robots(i) <=  3000;
  Order(i):   Robots(i)              >=  Ordered(i);
  MAXIMIZE profit: SUM(i) Price(i)*Robots(i);
{$Solve}
PRINT profit; Robots;
END
```

- SET introduces a index called i. The index 'i' has 10 members: they are the integers '1' through '10'.
- VAR introduces a variable list called 'Robots', which is indexed over i. This declares the 10 variables 'Robots(1)' ... 'Robots(10)'.
- COEF introduces five data lists, all indexed over i. The data are directly assigned to the lists. This means, e.g., that Price(1) has the value 300 or Price(6) has the value 100.
- Indexed items can be summed up with the SUM operator. So
  `HC(1)*Robots(1)+ ... +HC(10)*Robots(10)`
  may be abbreviated as:
  `SUM(i) HC(i)*Robots(i)`
- Much like the summation through the SUM operator, whole restriction classes may be shortened into one restriction Order(i).
- Again a PRINT may be used to output the results. In this case a whole table is produced in the ROBOT3.NOM file.

exercise: Type the following commands:

```
LPL robot3
type robot3.nom
lpl robot3 o
lpledit robot3.nom  ( type <ctrl>KX and Y to quit the editor)
```

3.5. WITH DATA TABLES

Another formulation of the same model is ('ROBOT4.LPL'):

```
SET  i    | HC | HM | HT | Ordered | Price =
( Robot1 |  5 |  4 |  6 |    20   |   300
  Robot2 |  5 |  8 |  2 |    15   |   200
  Robot3 |  4 |  5 |  4 |     7   |   100
  Robot4 |  5 |  6 |  6 |     6   |    50
  Robot5 |  6 |  4 |  3 |     5   |    50
  Robot6 |  5 |  8 |  4 |     8   |   100
  Robot7 |  7 |  7 |  5 |     9   |   200
```

```
  Robot8 |  8 |  6 |  2 |     8   |   100
  Robot9 |  4 |  5 |  5 |     7   |   400
  Robot10|  7 |  3 |  3 |     5   |   200
 );
VAR  Robots(i);
MODEL
  Components: SUM(i) HC * Robots <=  3500;
  Mounting:   SUM(i) HM * Robots <=  4800;
  Testing:    SUM(i) HT * Robots <=  3000;
  Order(i):   Robots  >=  Ordered;
  MAXIMIZE profit: SUM(i) Price * Robots;
{$Solve}
PRINT: 'Robots     Rob-Order   Price     Robots  Tot.Hours';
PRINT(i): Robots-Ordered, Price, Robots, HC+HM+HT;
END
```

- The members of set i need not be integers. The user can give them names ('Robot1' ... 'Robot10').
- The data which are indexed over a set need not be put into a list in the right order (!) between '/' and '/'. They can be put directly behind the members to which they belong, in a convenient table format.
- The cumbersome indices in a restriction may be dropped, since LPL already knows that 'Robots' has been defined over i.
- The PRINT may also be used to output a line of text or a list of expressions.

exercise: Type the following commands:

```
LPL robot4
lpledit robot4.nom  ( type <ctrl>KX and Y to quit the editor)
```

3.6. MORE ABOUT INDICES AND TABLES

Suppose now that we need not 3 but 8 different production steps to produce some robots. This again can be formulated with an additional index j ('ROBOT5.LPL').

```
SET    i | Ordered | Price  =
    ( Robot1 | 20 | 300     Robot2 | 15 | 200
      Robot3 |  7 | 100     Robot4 |  6 |  50
      Robot5 |  5 |  50     Robot6 |  8 | 100
      Robot7 |  9 | 200     Robot8 |  8 | 100
      Robot9 |  7 | 400     Robot10|  5 | 200
    );

       j  |  Capacity =
    ( Step1 | 3500   Step2 | 4800   Step3 | 3000   Step4 | 3400
      Step5 | 3000   Step6 | 3200   Step7 | 4000   Step8 | 2500
    );

COEF Hours(i,j) =
         | Step1  Step2  Step3  Step4  Step5  Step6  Step7  Step8 |
  Robot1 |   9      9      9      9      9      9      .      9   |
  Robot2 |   5      8      2      .      3      .      .      3   |
  Robot3 |   4      5      4      .      5      6      .      5   |
  Robot4 |   5      6      6      .      8      .      .      4   |
  Robot5 |  10      4      3      .      .      5      .      6   |
  Robot6 |   5      8      4      .      .      .      5      1   |
  Robot7 |   7      7      5      5      .      3      .      2   |
  Robot8 |   8      6      2      4      .      .      .      5   |
  Robot9 |   4      5      5      6      4      .      .      .   |
  Robot10|   7      3      3      1      1      1      1      4   |;

VAR  Robots(i);
MODEL
  Steps(j): SUM(i) Hours(i,j) * Robots(i) <= Capacity(j);
```

```
  Order(i):   Robots(i)  >=  Ordered(i);
  MAXIMIZE profit: SUM(i) Price(i) * Robots(i);
{$Solve} PRINT profit; Robots;
END
```

- Two indices have been defined. One (i) for the different types of robots, each having an ordered quantity and a price, and the other (j) for the different production steps, each having a maximal capacity.
- The time, which indicates the hours required for each type of robot i at each production step j is defined in the two-dimensional table Hours(i,j). Unused time (or zero time) is defined by a dot.

exercise: Type the following commands:

```
LPL robot5
lpledit robot5.nom  ( type <ctrl>KX and Y to quit the editor)
lplequ robot5
lpledit robot5.equ  ( type <ctrl>KX and Y to quit the editor)
```

3.7. STRUCTURE AND DATA

Structure and data of the model may be separated as in 'ROBOT6.LPL':

```
SET   i | Ordered | Price;
      j |  Capacity;
COEF  Hours(i,j);
VAR   Robots(i);
  { the data are defined in another file by: }
  {$I 'ROBOT6.INC' $}
MODEL
  Steps(j): SUM(i) Hours * Robots <= Capacity;
  Order(i):   Robots  >=  Ordered;
  MAXIMIZE profit: SUM(i) Price * Robots;
{$S} PRINT Robots:5; Order:4:1;
END
```

- This is a general, dimensionally independent formulation of the problem. In fact, we do not care about how many types of robots are produced in how many steps. The data are fully separated from the general model structure. The model may accept any number of both; the above formulation does not change.
- The data may even be put in another file, which is automatically included at compile time. A special comment beginning with '{$I ' is used for this purpose.
- Again the cumbersome indices in the restrictions may be dropped.
- The tables produced by PRINT may contain a format specification to indicate the width of digit positions.

The content of the file 'ROBOT6.INC' may be as follows:

```
SET    i  | Ordered | Price  =
    ( Robot1 | 20 | 300       Robot2 | 15 | 200
      Robot3 |  7 | 100       Robot4 |  6 |  50
      Robot5 |  5 |  50       Robot6 |  8 | 100
      Robot7 |  9 | 200       Robot8 |  8 | 100
      Robot9 |  7 | 400       Robot10|  5 | 200
    );

       j  |  Capacity =
```

```
    ( Step1 | 3500     Step2 | 4800     Step3 | 3000     Step4 | 3400
      Step5 | 3000     Step6 | 3200     Step7 | 4000     Step8 | 2500
    );
```

```
COEF Hours(i,j) =
         | Step1  Step2  Step3  Step4  Step5  Step6  Step7  Step8 |
  Robot1 |   9      9      9      9      9      9      .      9   |
  Robot2 |   5      8      2      .      3      .      .      3   |
  Robot3 |   4      5      4      .      5      6      .      5   |
  Robot4 |   5      6      6      .      8      .      .      4   |
  Robot5 |  10      4      3      .      .      5      .      6   |
  Robot6 |   5      8      4      .      .      .      5      1   |
  Robot7 |   7      7      5      5      .      3      .      2   |
  Robot8 |   8      6      2      4      .      .      .      5   |
  Robot9 |   4      5      5      6      4      .      .      .   |
  Robot10|   7      3      3      1      1      1      1      4   |;
```

If the table Hours(i,j) were sparse, it would be convenient to replace the above table with another table format:

```
COEF Hours(i,j) =
  [ Robot1  Step1  9
    Robot1  Step2  9
     { ... more tuples here ... }
    Robot10 Step7  1
    Robot10 Step8  4 ];
```

exercise: Type the following commands:

```
LPL robot6
lpledit robot6.lpl  ( type <ctrl>KX and Y to quit the editor)
lpledit robot6.sol  ( type <ctrl>KX and Y to quit the editor)
lpledit robot6.nom  ( type <ctrl>KX and Y to quit the editor)
lplequ robot6
lpledit robot6.equ  ( type <ctrl>KX and Y to quit the editor)
```

3.8. CONDITIONS AND SELECTION

Suppose we want to make sure that in the general model 'Robot5' never passes through 'Step6' independently of what the table Hours(i,j) says. Furthermore, we want only to produce Robots for which the order is greater than zero. The model can then be formulated as ('ROBOT7.LPL'):

```
SET   i | Ordered | Price;
      j |  Capacity;
COEF  Hours(i,j | i<>5 OR j<>6);
VAR   Robots(i | Ordered>0);
  { the data are defined in another file by: }
  {$I 'ROBOT6.INC' $}
MODEL
  Steps(j): SUM(i) Hours * Robots <= Capacity;
  Order(i):   Robots  >=  Ordered;
  MAXIMIZE profit: SUM(i) Price * Robots;
{$S} PRINT profit; Robots; Steps;
END
```

- (i,j) is called Indexlist and it declares as many tuples as i has members multiplied by the number of members in j. If i has 10 members and j has 8 members, 8x10=80 tuples have been declared (the Cartesian Product). An Indexlist may contain any number of indices. An Indexlist may be extended by a condition,

which limits the tuple list. The condition begins with a bar ('|'). The above condition 'i<>5 OR j<>6' excludes exactly what we wanted: 'declare 'Hours' for all tuples (i,j) except when i is 5 and j is 6. '5' and '6' indicate here the positions of the members within the sets. The condition could also be written as 'NOT (i=5 AND j=6)'. Any mathematical expression may be used for a condition. If the expression evaluates to zero, it would be interpreted as 'tuple does not exist', else as 'tuple does exist'.

- 'VAR Robots(i)' declares as many variables as i has members. But 'VAR Robots(i | Ordered>0)' again limits this number. Only the 'Robots' for which the order is greater zero are declared, the others are discarded from the model.
- Restriction names are allowed in the PRINT. In this case the slack is printed.

Any Indexlist may be followed by a condition. Suppose we want the maximizing function only summed up over all robots yielding a price greater than 100. We would then write the maximizing function as:

```
MAXIMIZE profit: SUM(i | Price>100) Price * Robots;
```

exercise: Type the following commands:

```
LPL robot7
lpledit robot7.nom  ( type <ctrl>KX and Y to quit the editor)
lplequ robot7
lpledit robot7.equ  ( type <ctrl>KX and Y to quit the editor)
```

Note that '1041'...'10410' are the variables-names and '1051'...'1058' are the restriction-names. To produce more readable names add {$N5} at the beginning of the ROBOT7.LPL file:

```
lpledit robot7.lpl
(add {$N5} and save with <ctrl>KD)
lpl robot7
lplequ robot7
lpledit robot7.equ  (type <ctrl>KX and Y to quit the editor)
```

3.9. FUNCTIONS

Suppose we want a model where the prices are never less than 100. We can use one of the two functions 'MAX()' or 'IF()' in the maximizing function ('ROBOT8.LPL'):

```
SET   i | Ordered | Price;
      j |  Capacity;
COEF  Hours(i,j);
VAR   Robots(i);
  { the data are defined in another file by: }
  {$I 'ROBOT6.INC' $}
MODEL
  Steps(j): SUM(i) Hours * Robots <= Capacity;
  Order(i):   Robots  >=  Ordered;
  MAXIMIZE profit: SUM(i) MAX(Price;100) * Robots;
  {or  MAXIMIZE profit: SUM(i) IF(Price<100;100;Price) * Robots; }
{$S} PRINT profit; Robots;
END
```

- The MAX function returns the greater of the two arguments.
- The IF function takes three (or two) arguments. The first is a condition. The

second argument is evaluated if the condition evaluates to non-zero; else, the third argument is evaluated. If the third argument is dropped, zero is returned.

Examples:

IF(2>1;1;0) gives 1

IF(2<=1;1) gives 0

3.10. INDEX-OPERATORS

Suppose we want only to produce Robots for which the total hours on all machines does not exceed 60 hours, which take no longer on any machine than 9 hours, and which must not pass through all production steps. The model takes the simple form of ('ROBOT9.LPL'):

```
SET   i | Ordered | Price;
      j |  Capacity;
COEF  Hours(i,j);
VAR   Robots(i |
       SUM(j) Hours<60 AND FORALL(j) (Hours<=9) AND EXIST(j) (Hours=0) );
  { the data are defined in another file by: }
  {$I 'ROBOT6.INC' $}
MODEL
  Steps(j): SUM(i) Hours * Robots <= Capacity;
  Order(i):   Robots  >=  Ordered;
  MAXIMIZE profit: SUM(i) Price * Robots;
{$S} PRINT profit; Robots;
END
```

- The three operators SUM, FORALL and EXIST are called Index-operators. 'SUM(j) Hours' calculates the number of hours to produce each Robot i, 'FORALL(j) (Hours<=9)' tests whether the hours exceed 9 at any production step for any robot, and 'EXIST(j) (Hours=0)' tests whether there is at least one production step with Hours=0 for each robot.

This excludes Robot(1), since the total hours to produce this Robot is 63 and exceeds the 60 limit. It excludes also Robot(5), since it takes 10 hours in step 1 to produce it, which exceeds the 9 hour limit. Furthermore Robot(10) is excluded, since it take at least 1 hour in all production step to produce this robot.

Subsequent summation on all variables 'Robots' yields only the declared list.

Example:

```
SUM(i) Robots
```

produces the addition

```
Robots(2)+Robots(3)+Robots(4)+Robots(6)+Robots(7)+Robots(8)+Robots(9)
```

exercise: Type the following commands:

```
lpl robot9 o
lplequ robot9
lpledit robot9.equ  ( type <ctrl>KX and Y to quit the editor)
lpledit robot9.nom  ( type <ctrl>KX and Y to quit the editor)
```

3.11. INTERMEDIATE EXPRESSIONS

An LPL model need not be a complete LP model. It can be used for intermediate,

index based expression evaluation. Suppose we wanted only to know how many robots of each type could be produced maximally considering the limiting capacity on each step. We must build the minimum 'Capacity/Hours' of each production step. This is done in the next formulation 'ROBOTA.LPL':

```
SET   i | Ordered | Price;
SET   j |  Capacity;
COEF  Hours(i,j);
  { the data are defined in another file by: }
  {$I 'ROBOT6.INC' $}
     How_Many_max(i) = SMIN(j | Hours) Capacity/Hours;
PRINT : '           Maxi(x)    Mini(y) How_Many_max';
PRINT(i):  x , y , How_Many_max;
END
```

- LPL allows the user to calculate any intermediate expression and print them. This LPL model is not a LP model. No solver is needed. This example shows, how LPL can be used simply to manipulate tables and to print them.
- 'How_Many_max(i)' calculates the desired list. 'SMIN' is another Index-operator which returns the minimum 'Capacity/Hours' of all steps (j). The condition 'Hours' in 'j | Hours' is necessary, because some 'Hours' could be zero and this would produce a division error.

exercise: Type the following commands:

```
lpl robotA
lpledit robotA.nom  ( type <ctrl>KX and Y to quit the editor)
```

3.12. INDEX-TREES

Suppose now that some production steps are subdivided into further steps (e.g. 'Step1' contains 'Step1a' and 'Step1b') which in turn may be composed of other steps like:

```
Step1
  Step1a
  Step1b
Step2
Step3
  Step3a
  Step3b
    Step3ba
    Step3bb
  Step3c
  Step4
  Step5
    Step5a
    Step5b
  Step6
  Step7
  Step8
    Step8a
      Step8aa
      Step8ab
        Step8aba
        Step8abb
        Step8abc
        Step8abd
    Step8b
```

This is a list with a hierarchical structure. Such a list is called an index-tree. LPL can handle index-trees like other indices lists. An index-tree is declared in LPL like any

other index. To preserve the hierarchical structure, sub-lists within other lists of members must be bracketed by '(' and ')'. Indentation helps one to more easily read the structure of the list ('ROBOTB.LPL').

```
SET   i | Ordered | Price;
      j |  Capacity;
COEF  Hours(i,j);
  { the data are defined in another file by: }
  {$I 'ROBOTB.INC' $}
      Main_Capa(k=j) = IF(k IN #j ; SUM(l=k) Capacity ; Capacity);
VAR   Robots(i |
       SUM(j) Hours<60 AND FORALL(j) (Hours<=9) AND EXIST(j) (Hours=0) );
MODEL
  Steps(j): SUM(i) Hours * Robots <= Main_Capa;
  Order(i):   Robots  >=  Ordered;
  MAXIMIZE profit: SUM(i) Price * Robots;
{$S} PRINT(i): Ordered, Price, Robots;
PRINT(j): SUM(i) Hours*Robots , Main_Capa;
END
```

where the file 'ROBOTB.INC' differs from 'ROBOT6.INC' only in the definition of the set 'j'. Data can be entered as in a normal index list.

```
SET j             | Capacity =
 (Step1
   (Step1a        |  2000
    Step1b        |  1500 )
  Step2           |  4800
  Step3
   (Step3a        |  1000
    Step3b
     (Step3ba     |   700
      Step3bb     |   300 )
    Step3c        |  1000 )
  Step4           |  3400
  Step5
   (Step5a        |  2000
    Step5b        |  1000 )
  Step6           |  3200
  Step7           |  4000
  Step8
   (Step8a
     (Step8aa     |  1400
      Step8ab
       (Step8aba  |   100
        Step8abb  |   200
        Step8abc  |   100
        Step8abd  |   200 ) )
    Step8b        |   500 ) );
```

- LPL can handle not only simple index lists but also index-trees. An index-tree is a hierarchical list of members (sets containing sets, containing sets, containing....).
- An index-tree must be declared in LPL as a nested list. Indentation helps one to read the structure.
- Like other indices, index-trees may be used to declare and define coefficients, variables, and restrictions such as 'Hours(i,j)'. Note that 'Hours' now contains 10x25 tuples.
- Every member within an index-tree can be used as an index itself. Thus, the user could declare a variable 'X(Step8)' running only over all steps defined within 'Step8'.

- More importantly, powerful arithmetical operations are possible on index-trees. An example is given in the following text.

'Capacity' only has been defined on the members, which have no further sub-lists. (These members within an index-tree are called leaves in LPL). This makes sense, since the 'Capacity' of a composed step should be calculated as a summation of the Capacities of all sub-steps (e.g. 'Step5' has capacity of 3000 since it is composed of 'Step5a' with capacity 2000 and 'Step5b' with capacity 1000). This is exactly what the expression

```
Main_Capa(k=j) = IF(k IN #j ; SUM(l=k) Capacity ; Capacity);
```

does. 'k=j' means 'k is just another name for j'. 'k' is called a dummy-index. 'k IN #j' means 'test whether a specific k is a member which has further sub-lists attached'. (They are called non-leaves in LPL.) 'SUM(l=k) Capacity' sums all capacities of the sub-steps of a specific member k.

LPL distinguishes different types of members within an index-tree:

Members which have no further sub-lists attached, the leaves.

Members which have sub-lists attached, called non-leaves.

From the point of view of a member, we may define the parent and the children.

If j is a index-tree, then the syntax '&j' is used for the set of all leaves within 'j'; '#j' is used for all non-leaves in 'j'; '$j' is used for all children of 'j'; and, if 'j' is a member, then '!j' is used for the parent of 'j'.

Example:

'Step8ab' is a non-leaf and, therefore, belongs to #j; it is the parent of 'Step8aba' and a child of 'Step8a', but not a leaf. 'Step7' is a leaf, since it has no child; its parent is 'j' itself and it is a child of 'j'.

exercise: Type the following commands:

```
LPL robotB
lplequ robotB
lpledit robotB.equ  ( type <ctrl>KX and Y to quit the editor)
lpledit robotB.nom  ( type <ctrl>KX and Y to quit the editor)
```

The attentive reader may have noticed in the 'ROBOTB.EQU' file that only 8 restrictions 'STEPS' (1, 4, 5, 11, 12, 15, 16 and 17) have been produced, although j now contains 25 members (all the steps). This is because 'Hours' has been defined only for the mainsteps (STEP1-STEP8) and, therefore, all terms 'SUM(i) Hours*Robots' for all sub-steps yield nothing. LPL is clever enough to eliminate the whole restriction in these cases. To define the restriction 'STEPS' only for the main steps (the children of j), we could formulate this restriction as:

```
MODEL Steps($j): SUM(i) Hours * Robots <= Main_Capa;
```

3.13. MORE ON INDEX-TREES AND NAMES

As an exercise, we may also build an Index-tree on the different robot type (see file 'ROBOTC.INC'):

```
SET    i               | Ordered | Price  =
   ( Robot1     '1'
      (Type1    '1A'  | 10 | 300
       Type2    '1B'  | 10 | 300)
     Robot2     '2'
      (Type1    '2A'  | 20 | 200
       Type2    '2B'  | 10 | 210
       Type3    '2C'  |  5 | 220
       Type4    '2D'  | 10 | 190)
     Robot3     '3'   |  7 | 100
     Robot4     '4'   |  6 |  50
     Robot5     '5'   |  5 |  50
     Robot6     '6'   |  8 | 100
     Robot7     '7'   |  9 | 200
     Robot8     '8'   |  8 | 100
     Robot9     '9'
      (Type1    '9A'
       (Subtype1'9AA' | 15 | 415
        Subtype2'9AB' |  5 | 410)
       Type2    '9B'  | 10 | 390
       Type3    '9C'  |  5 | 380
       Type4    '9D'  | 10 | 430)
     Robot10    '10'  |  5 | 200
   );
```

Non-leaf Robots are not 'real' robots. They are just names to collect different subtypes. Therefore, the coefficients 'Ordered' and 'Price' are not defined for the non-leaves in i and would be of no interest. Nevertheless, it may be interesting to see how many Robots of a main-type are produced. Since 'Hours' is always defined over the main steps and the main types of robots (supposing that every robot-type within a main type passes the same time on the different steps), the variables of the sub-types of robots must be summed up within the 'BALANCE' restrictions. This yields the following formulation of 'ROBOTC.LPL:

```
SET   i | Ordered | Price;
      j |  Capacity;
COEF  Hours(i,j);
  { the data are defined in another file by: }
  {$I 'ROBOTC.INC' $}
      Main_Capa(k=j) = IF(k IN #j , SUM(l=k) Capacity , Capacity);
VAR   Robots(i);
MODEL
  Balance(k=#i): SUM(l=$k) Robots = Robots;
  Steps($j): SUM($i) Hours * Robots <= Main_Capa;
  Order(&i):   Robots  >=  Ordered;
  MAXIMIZE profit: SUM(i) Price * Robots;
{$S} PRINT(i): Ordered, Price, Robots;
PRINT(j): SUM(i) Hours*Robots , Main_Capa;
END
```

exercise: Type the following commands:

```
LPL robotC
lplequ robotC
lpledit robotC.nom  ( type <ctrl>KX and Y to quit the editor)
lpledit robotC.equ  ( type <ctrl>KX and Y to quit the editor)
```

BASIC LPL LANGUAGE ELEMENTS

This Chapter gives a systematic overview of LPL's basic elements: reserved words, identifiers, numbers, operators, and expressions.

4.1. BASIC CHARACTERS

The basic alphabet of LPL consists of the following characters:

Letters: A to Z, a to z and _ (underscore)
Digits: 0 1 2 3 4 5 6 7 8 9
Special characters: + - * / = < > ( ) [|] { } . , : ; ' # & $? ! ^ "

The alphabet shows that LPL models are written entirely in printable ASCII characters and may, therefore, be handled as ordinary text files.
No distinction is made between upper and lower case letters. Names, reserved words, certain operators, and delimiters are formed using more than one character.

4.2. RESERVED WORDS

Reserved words are an integral part of LPL. They must never be used as user-defined identifiers. The reserved words are:

ABS	**AND**	**CEIL**	**CHECK**	**COEF**
COL	**COS**	**DEFAULT**	**DELETE**	**END**
ENDMASK	**EQUATION**	**EXIST**	**FILE**	**FLOOR**
FORALL	**IF**	**IN**	**INTEGER**	**LOG**
MASK	**MAX**	**MAXIMIZE**	**MIN**	**MINIMIZE**
MODEL	**NOT**	**OR**	**PRINT**	**PROD**
PROGRAM	**RND**	**RNDN**	**ROW**	**SET**
SIN	**SMAX**	**SMIN**	**SQRT**	**SUM**
TRUNC	**VAR**			

4.3. LPL MODELING STYLE

An LPL model has a free format style. The user can enter blanks, new-line characters or comments anywhere needed. LPL language tokens (see next section) must be separated by at least one of the following delimiters: a blank, an end of line character, or a comment. A good style, however, would be to begin a new declaration on a new line, and emphasize the structure of the model with indentation.

4.4. LPL TOKEN

An LPL model consists of different basic elements: identifiers to designate indices,

members of indices, coefficients, variables, and restrictions; operators to build algebraic expressions; numbers to define data; function names, and other elements. All these elements are called tokens. One or several characters form a token. The different kinds of token are explained in the subsequent sections.

4.5. IDENTIFIERS

Identifiers are used to denote *user defined objects* such as indices, their members, variables, restrictions, and coefficients. The syntax of an identifier in LPL consists of a letter or an underscore followed by any combination of letters, digits or underscores.

SYNTAX:

```
Identifier ::= Letter { Letter | Digit }
```

A identifier is not limited in length, but only 20 characters will be significant.

Examples:

```
Var_1
TEXT
_None
ImportedProducts
3xY                         { illegal, starts with a digit }
Two words                   { illegal, must not contain a space }
This_is_A_very_long_Identifier
```

LPL does not distinguish lower and upper case letters by default. So the use of mixed upper and lower case as in 'ImportedProducts' has no functional meaning. It is nevertheless encouraged as it leads to more legible identifiers. 'ImportedProducts' is easier to read than 'IMPORTEDPRODUCTS'. The compiler, however, can distinguish lower and upper case letter through the compiler directive {$C (see Chapter 9.1).

The same identifier may be used to denote a variable, a set, and a member.

4.6. NUMBERS

Numbers are constants of real type, the only numerical type used in LPL. They constitute the *data* of the model. They may also be used to denote members of an index.

The range of real numbers is 1E-38 through 1E+38 with a mantissa of up to 11 significant digits. In boolean expressions zero is interpreted as FALSE and any other value is interpreted as TRUE.

Syntax:

```
Number ::= [ + | - ] Digit { Digit } [ . Digit { Digit } ]
```

Examples:

```
123
+234.980
-87467632.098
- 56
3.6E-3    {illegal, exponential notation is not allowed}
```

In data tables within a LPL model, a dot may be used to replace a default numerical value.

4.7. OPERATORS

Operators are used to form mathematical expressions. The following operators are defined in LPL with this precedence:

```
+ - (unary plus and minus)
^
/  *
SUM  PROD  FORALL  EXIST  SMIN  SMAX  COL  ROW
-  + (binary plus and minus)
>=  <=  >  <  <>  =  IN
AND
OR
, | (a comma or a bar)
```

Example:

```
2 + 3 * -7 < b AND c = 0
```

will be interpreted as:

```
(2 + ( 3 * ( -7 ) ) < b) AND (c = 0)
```

The parentheses may be used to change the operator's precedence in a expression as in

```
2*(3+6)
```

The operators have the following meaning (where x and y are reals or integers, except with the IN operator, which is explained below):

```
+ x   : (unary +): returns x
- x   : (unary -): returns -x
NOT x : if x=0, it returns 1, else it returns 0
x ^ y : returns y power of x, error if x<0
x / y : division, error if y=0
x * y : multiplication
SUM(i) a(i)    : returns the sum of all coefficients 'a' over index i.
PROD(i) a(i)   : returns the product of all coefficients 'a' over index i.
FORALL(i) a(i) : returns 1, if all coefficients 'a' over index i
                 are different from zero, else it returns 0.
EXIST(i) a(i)  : returns 1, if at least one of all coefficients 'a' over
                 index i is different from zero, else it returns 0.
SMIN(i) a(i)   : returns the smallest 'a' within all 'a' over index i.
SMAX(i) a(i)   : returns the largest 'a' within all 'a' over index i.
COL(i) a(i)    : expands a print mask horizontaly (see Report Generator).
ROW(i) a(i)    : expands aprint mask vertically (see Report Generator).
x - y  : subtraction
x + y  : addition
x >= y : if x>=y returns 1 else 0
x <= y : if x<=y returns 1 else 0
x > y  : if x>y returns 1 else 0
x < y  : if x<y returns 1 else 0
x <> y : if x<>y returns 1 else 0
x = y  : if x=y returns 1 else 0
x IN y : requires x to be a indexname or a dummy. y must be a Path. It returns 1 if a
specified x is in the set y else 0.
x AND y: if (x<>0) AND (y<>0) returns 1 else 0
x OR y : if (x<>0) OR (y<>0) returns 1 else 0
x , y  : returns both x and y as a indexed list of two values
```

Examples:

```
'2^4+2'   gives  18.
'3=2'     gives  0
'3<2-6'   gives  1
'1 or 6-6'  gives 1
'not (0 and 1)'  gives  1
```

Note that there is no TRUE or FALSE value. Like in C, the numerical values 1 (or any value other than zero) replaces the TRUE and 0 replaces the FALSE boolean value.

This means that an expression such as 'a<>0' can always be written simply as 'a'.

SUM, **PROD**, **SMIM**, **SMAX**, **FORALL**, **EXIST** , **COL**, and **ROW** are the Index-operators in LPL. They have the same meanings as the corresponding sum and product operator (∑ and ∏), the 'min' and 'max' function over an index in mathematics, and the All and Exist operators in the predicate logic. COL and ROW are only used for the Report Generator (see Chapter 10). The Index-operators are always followed by an index-list.

Examples:

```
SUM(i) 1  evaluates to n, if n is the number of members of index i
SUM(i) a(i)  sums all a over i  [  a(1)+a(2)+a(3)+....]
SUM(i | a(i)<100) a(i)  sums all 'a' over i such that a(i) is less than 100
SMAX(i) a(i)  returns  the largest a(i)
SMIN(i | a>0) a(i)  returns  the smallest of all a(i) greater zero
EXIST(i) a(i)  tests, whether any a(i) is not zero. If yes this evaluates to 1 else to 0
FORALL(i) a(i)  is the same as 'NOT (EXIST(i)(a(i)<>0))'
```

Another example with **FORALL**:

```
COEF a(i,j,k)
VAR x( i,j | FORALL(k) a(i,j,k) )
```

means: There exists a variable x(i,j) for every (i,j), if for all k in a(i,j,k), a(i,j,k) is not zero. The underlying operator of FORALL is AND (or the All operator in predicate logic). This could have been written as:

```
VAR x(i,j | a(,,1)<>0 AND a(,,2)<>0 AND ... AND a(,,kn)<>0 )
```

where kn is the number of members in k.

The **EXIST** Index-operator has a similar meaning. The underlying operator is the OR operator:

```
COEF a(i,j,k)
VAR x( i,j | EXIST(k) a(i,j,k) )
```

means: There exists a tuple x(i,j) for every (i,j), if for all k at least one a(i,j,k) is not zero. This could have been written as:

```
VAR x(i,j | a(,,1)<>0 OR a(,,2)<>0 OR ... OR a(,,kn)<>0 )
```

where kn is the number of members in k.

The **IN** Operator is a powerful operator and is used to test whether a specific member is within a specific set. The first argument must be a dummy (see Chapter 6) or an index; the second argument must be an index.

Example:

```
SET i = ( 1:10 );
SET j = ( 2:4 );
```

then the following expression

```
SUM(i | i IN j) a(i)   { sum all a(i), if i is also in the set j }
```

sums up the three following values: a(2)+a(3)+a(4)

The relational operators '>=' '<=' '>' '<' '<>' and '=' may also have indices as arguments. If indices are used in expressions, the four operators '>=' '<=' '>' '<' are

interpreted as 'the position of a specific member within a set'. (See Chapter 6 for the term 'position').

Examples:

```
VAR x(i,j | i<j);
```

declares a variable for every tuple (i,j), if the member of i has a position which is less than the position of the member of j. This, actually, declares the upper right triangular matrix of x(i,j), if i and j have the same numbers of members. Note that the four operators do not care about the members itself, but only on their position within the set.

This is different with the operators '<>' and '=', which looks at the members' names and compares *them*. (string comparison)

Example:

```
VAR x(i,j | i<>j)
```

means: There is a variable for every tuple (i,j), if the member name of i is different from the member name of j. If i and j represent the same set, this excludes all tuples on the diagonal.

Note that '=' is also used as an assign operator.

4.8. FUNCTIONS

Several algebraic functions are defined in LPL, where 'x', 'y' and 'z' are any expressions:

IF (x ; y [; z]) : If 'x' evaluates to non-zero (TRUE), then y is evaluated, else z is evaluated. An omitted z means zero.

```
IF(2>1 ;12 ; 13)  evaluates to  12 (since '2>1' is TRUE)
2 + IF(1=2-1 ; 4;5)  evaluates to  6
- IF(2;3;4) - IF(0;100;5)  evaluates to  8
IF(2=0;1)  evaluates to  0
```

MAX (x ; y) : returns the greater of x and y.

```
MAX(-20;10)  gives  10
MAX(100-1000;-800)  gives -800
MAX(ABS(-10);ABS(-20))  gives 20
```

MIN (x ; y) : returns the smaller of x and y.

```
MIN(-20;10)  gives  -20
MIN(100-1000;-800)  gives -900
MIN(ABS(-10);MAX(100;-20))  gives 10
```

ABS (x) : returns the positive value of x.

```
ABS(-20)  gives  20
ABS(100-1000)  gives 900
ABS(10)  gives 10
```

CEIL (x) : returns the integer greater than x.

```
CEIL(20.15)  gives  21
CEIL(-20.15)  gives -20  (not -21!)
```

FLOOR (x): greatest integer smaller than x

```
FLOOR(20.15)  gives  20
FLOOR(-20.15)  gives -21  (not -20!)
```

TRUNC (x) : truncated x to an integer

```
TRUNC(20.15)  gives  20
TRUNC(-20.15)  gives -20
```

SIN (x): sinus of x

COS (x) : cosines of x

LOG (x) : natural logarithm of x, error if x<=0

SQRT (x) : square root of x, error if x<0

RND (x ; y) : returns a uniform random value in the interval [x,y]

RNDN (x ; y) : returns a normal distributed value with mean x and stand. deviation y.

4.9. COMMENTS

A comment may be inserted between tokens anywhere in the model. They are delimited by the curly braces '{' and '}', by the symbols '(*' and '*)', or by double quotes.

Examples:

```
{ This is a comment }
(* and so is this. *)
" and this is still another one "
{ but not this *)
```

The three comment types may be nested within each other, but {...} cannot be nested within {...} , neither can (*...*) within (*...*), nor "..." within "...". This allows the LPL modeler to structure the comments. Note that comment can span over several lines.

4.10. OVERVIEW OF ALL TOKENS

The following special symbols are used as tokens in following contexts:

```
+ unary or binary plus operator
- unary or binary minus operator
* multiply operator
/ division operator, begin and end data list delimiter
^ power operator
< less operator
> greater operator
= equal operator, assign operator
( )       to bracket sets and subsets, Indexlists,
          or to change operator precedence
[ ]       to bracket Indexlists
' to enclose alias names of members, filenames or restriction-names
"         to enclose a comment
{ }       to enclose a comment
| data tables delimiter, to separate multi declaration and expressions
. decimal point, or default value in data tables
, to list items in sets, data tables, Indexlists, and expressions
          (they are necessary only in Indexlists and expressions)
: range separator in sets, path delimiter in sets
          or as terminal delimiter of the restriction-name.
; terminal delimiter of a statement
# non-leave selector of a tree index
$ children selector of a tree index
& leaves selector of a tree index
! parent selector of a tree index
? for automatically indexing
```

Composed token:

```
(*        begin comment
*)        end comment
```

```
<>        not equal operator
<=        less or equal than operator
>=        greater or equal than operator
<reserved words>
<Identifiers>
<numbers>
```

4.11. EXPRESSIONS

Numbers, operators, functions, coefficients, variables and index-names are used together to form algebraic expressions. Expressions written in the LPL language are very close to ordinary mathematical notation. There are some exceptions:

- The ∑ and the ∏ symbol used to sum or multiply terms over indices is replaced by the reserved word **SUM** and **PROD** in LPL
- The All- and Exist-Operators in predicate logic are replaced by the reserved words **FORALL** and **EXIST**.
- Indices are not subscripted as in 'a_{ij} '. Parentheses are used instead to enclose the index-list and they are separated by commas as in 'a(i,j)' or 'a[i,j]' (both are possible in LPL).

The reader may refer to Appendix B for the syntax of a well formed expression. A well formed expression always evaluates to a numerical value. If the expression is a boolean expression, it evaluates to 1 (or another non-zero) for TRUE and 0 for FALSE. Therefore, '1=2' evaluates to 0 and may be interpreted as FALSE. Coefficients return their value as entered in the table. If they are not defined they return the default value. The same is true for variables. Indices used in an expression return the position of a specified member. If indices are used in an expression, they must be bound (see Chapter 6) to an index in a previous Index-list. Expressions which return just one value, are called single expressions or just expressions. Expressions which return a whole array of values are called indexed expressions.

Examples:

```
4+7*7^4  is a single expression
4^6/c  is a single expression, if 'c' is just a single coefficient
sum(i) a(i) is a single expression, since it returns only one value
a(i,j) + c(j) is a indexed expression
a(i) + sum(i) c(i) is another indexed expression
```

Expressions are used in four different contexts within LPL:

in the Coefficient or Variable Statement to assign expressions
in the Indexlist to limit the tuples by a condition
in Model Statement to assign restrictions
in the Print statement

Examples:

```
COEF composed(i) = a(i) + sum(i) c(i);       { assign coefficient }
. . . (i,j,k,l | a(i)>b(j) OR c(k)<d(l) )    { in an Indexlist }
MODEL r: x+y-23*z + 78;                      { assign a restriction }
```

Expressions concatenated with the comma operator ',' are called expression-list. They

are a special kind of indexed expressions, since they also return a array of values. Any expression-list is also an expression.

Example:

```
a,b,a+b  {is an expression with three values}
```

A expression-list may also be assigned to an identifier as any other expression. This identifier is then called automatically indexed.

Example:

```
COEF a = 3,4,5,6;
```

'a' is now automatically indexed. It contains a list of four values. Automatically indexed identifier are treated as other indexed identifier, but there is one important difference:

the index, which is automatically produced, is not known to the user. In general the user needs not care about automatical indices. But in some cases this is needed. Suppose the user wants to assign the sum of all values of 'a' (above) to a coefficient 'b', than he should know the index, since he must sum over the automatical index. In this case LPL uses a question-mark '?' as index:

```
COEF b = SUM(?) a;
```

An automatically indexed identifier can be used in any expression. This syntax is used in the PRINT statement. The comma may also be replaced by '|' (the bar).

STRUCTURE OF A LPL MODEL

A LPL model consists of an optional Heading Section and a sequence of Statements. The model ends with the reserved word **END**. Nine different Statements are possible:

Set Statement (defines indices and there members)
Coefficient Statement (defines the data)
Variable Statement (defines the variables of the model)
Model Statement (defines the restrictions of the model)
Equation Statement (same as Model Statement)
Check Statement (checks expressions and data consistency)
Mask Statement (defines a lay-out mask for the Report Generator)
Print Statement (writes a report to the output device)
Delete Statement (deletes some tables from the heap)

Syntax:

```
Model ::= [ ModelHeading ] Statement { Statement } END
Statement ::= SetStatement | CoefStatement | VarStatement | ModelStatement
              | EquStatement | CheckStatement | MaskStatement | PrintStatement
              | DeleteStatement
```

A Statement can be a declaration, where an identifier is declared, or it can be a definition, where values are assigned to an identifier. The same identifier may be declared and assigned several times, except sets which can only be assigned once..

Example:

```
VAR   x;
MODEL R;          { two declarations }
COEF   a = 3;
MODEL  R: x+y =0; { two definitions }
```

5.1. THE HEADING SECTION

In LPL, the model heading is purely optional and has no significance for the interpretation of the model. If present, it gives the model a name, which is treated as a comment. It consists of the reserved word **PROGRAM** followed by an identifier and a optional semicolon. It must be at the very beginning of the model description.

Syntax:

```
Heading ::= PROGRAM Identifier [ ';' ]
```

Example:

```
PROGRAM MyModel;
```

5.2. SET STATEMENT

The Set Statement declares or defines the indices used in the model. A index is also called a set or domain in LPL. A set consists of a list of items called members. A member may be an integer, a range of integers or an identifier. A member may also be followed by a list of other members. This kind of nested list is called an index tree. Furthermore, sets may be itself indexed, they are called indexed sets. The Set Statement begins with the reserved word **SET** followed by zero or several index declarations or definitions, each ending with a semicolon.

Syntax:

```
SetStatement ::= SET { Set ';' }
Set ::= SetIdentifier [ QuotedString ] { '|' Identifier  } [ '=' MemberList ]
        | IndexedSet
MemberList ::= '(' Member { [','] Member } ')'
Member ::= Item [ ':' Item ] [ QuotedString ] { '|' Data }
           | SetIdentifier [ = ] MemberList
Item ::= Identifier | Number
IndexedSet := CoefOrVarDeclaration
```

Examples:

```
SET
  i;          { declares an index called 'i'}
  h = ( 1 2 );{ declares a index h and assigns the integers
              { 1 and 2 to it }
  j = (1:10); { declares an index 'j' and assigns the list of
              { the integers 1 to 10 to it }
  Seasons = ( spring, summer, autumn, winter ) ;
              { declares an index 'Seasons' and assigns 4 identifiers
              { to it }
  MixedSet = ( 1, one, another, 3:10); { commas are optional }
  NestedList = ( A ( Aa Ab Ac (Aca Acb ) ) B ( Ba Bb ( Bba ) Bc ) C );
              { an index tree example }
  IndexedSet(i,h); { a tuple list of (i,h) tuples }
```

In a sequence of index declarations, there is no need to write the reserved word **SET** before every declaration, although this is also possible:

```
SET i;
SET h;
SET j;
SET Seasons;
SET MixedSet;
SET NestedList;
```

is a legal declaration. The same index may be declared several times, but it can only be defined once. It allows the modeler to first declare an index and to assign it later on in the model. Sets are discussed in more detail in Chapter 6.

5.3. COEFFICIENT STATEMENT

The Coefficient Statement declares or defines the data used in the LP model. They are numerical values (real or integer). Single coefficients or more dimensional coefficients (vectors, matrices or n-dimensional arrays) may be defined. The dimensions are determined by the domains (indices), which have been declared in the Set Statement. The data may be put in predefined table-formats or they may be numerical expressions. The reserved word **COEF** heads a Coefficient Statement and it is followed by zero or several coefficient declarations or definitions, each ending with a semicolon.

Syntax:

```
CoefStatement ::= COEF { CoefOrVarDeclaration ';' }
CoefOrVarDeclaration ::= IdentSpec [ '=' DataList ]
IdentSpec ::= Identifier [ Indexlist ]
```

Examples:

```
COEF
  a;            { declares a single coefficient 'a' }
  b = -4.678;   { defines a single coefficient 'b' }
  c(i);     { declares a array 'c' where 'i' is a index
            { the length of the array depends on the number
            { of the members of 'i' }
  d(i,j);   { declares a two-dimensional array 'd' with index 'i' and 'j'
            { the length of the array is: number of members of i times
            { number of members of j (Cartesian product) }
  e(i) = b*c(i) { define 'e' from other coefficients }
```

'i' and 'j' are index names, which must have been *declared before* in the model. But there is no need to *define* them before - except for the last expression -, although they can be defined also.

As in the Set Statement, coefficients may be declared several times, but can also be assigned several times. This allows the user to declare a coefficient within the structure of the model, and to assign data to it later on or even in another file. This separates more fluctuating items - such as the data - from the model structure, which is, generally, more stable.

The structure '(i)' or '(i,j)' is called Indexlist. An Indexlist declares over which domains a coefficient, a variable, or a restriction runs. Indexlists are also used together with Index-operators. Indexlists are explained in more detail in Chapter 6. The data table formats are explained in Chapter 7.

5.4. VARIABLE STATEMENT

The Variable Statement declares or defines all variables used in the model. This Statement has exactly the same structure as the Coefficient Statement except that it is headed by the reserved word **VAR**.

Examples:

```
VAR X;      { declares a variable 'X' }
VAR y(i);   { declares a array of variables 'y', the length of the
            { array depends on the number of members of set 'i' }
VAR z(i,j); { two-dimensional array of variable 'z' }
```

Variables may also be declared and defined several times. Variables are treated in Chapter 8.

5.5. MODEL AND EQUATION STATEMENT

The Model Statement declares or defines all restrictions and objective functions (if any) of the model. They are declared exactly as coefficients, which are assigned by expressions, except that they are headed by the reserved words **MODEL** or **EQUATION**. A definition of a restriction assigns an arithmetical expression to the declared restriction separated by a colon. Declaration and definition of restrictions can be put in separate places within the model like any other declared or defined item.

Mostly, however, restrictions are declared and defined in the same place.

Examples:

```
EQUATION r;   { declares a restriction 'r' }
MODEL p(i);   { declares a array of restriction 'p', the length of the
              { array depends on the number of members of set 'i' }
MODEL q(i,j); { two-dimensional array of restriction 'q' }
MODEL Res: x + y - 23*z < 12; { defines a restriction 'Res' }
```

More information on the Model and Equation Statement is provided in Chapter 8.

5.6. CHECK STATEMENT

The Check Statement is useful to check the data consistency within the model. The Check Statement begins with the reserved word CHECK followed by a optional IndexList. A colon separates the first part from the expression which is to be checked.

Example:

```
CHECK(i) : a(i)>1 and a(i)<100;
```

This check tests, whether all a(i) are between 1 and 100. (See Chapter 9 for more details).

5.7. PRINT AND MASK STATEMENT

The Mask and Print Statement together define the output of the model. A mask defines, *how* the output has to be formatted and the Print Statement specifies *what* to output. The Mask Statement begins with the reserved word MASK followed by a identifier, the mask-name. Any text may follow which ends with the reserved word ENDMASK.

Example:

```
MASK m1
   This is a simple testmask
   $$$$   ###.##
ENDMASK;
```

A Print Statement uses the layout of the mask to print the output. It begins with the reserved word PRINT followed by a mask-name and an expression within parentheses. The Print Statement may also be used independently of a mask, and prints tables to the output file .NOM.

Example:

```
PRINT m1('xyz',12);
```

The Output of this Print - together with the last mask - is:

```
   This is a simple testmask
   xyz     12.00
```

The Mask and the Print Statement are explained in Chapter 10.

5.8. DELETE STATEMENT

This Statement allows to delete some tables from the model in order to free the computer memory for new tables.

5.9. PUTTING IT ALL TOGETHER

A whole LPL model is a sequence of the nine different declarations or definitions. The sequence can be in any order, except that an identifier must first be declared (not necessarily defined as long as an expression is not evaluated) before it can be used in any expressions or in another syntax structure.

Example:

```
SET   < define indices >
COEF  < declare coefficients >
VAR   < declare variables >
COEF  < declare more coefficients >
COEF  < define just one coefficient >
EQUATION < declares restrictions >
COEF  < assign the data now (definition) >
MODEL < now produce the .RES file >
END
```

Different orders are possible for the same model. The most interesting may be:

```
1. SET   < define all sets >
   COEF  < define all coefficients >
   VAR   < define all variables >
   MODEL < define all restrictions >
   MASK  < define masks >
   PRINT < reporting >
   END
```

This order merges the model with all index-members and all data. This may be an appropriate order, especially for small models. There is no need to separate data, members, and structure, since they can be all seen at a glance.

```
2. SET   < define all sets >
   COEF  < declare all coefficients >
   VAR   < define all variables >
   EQUATION < define all restrictions >
   COEF  < define all coefficients >
   MODEL < now produce the model >
   END
```

Order 2 separates the data from the rest of the model. Data are often the most fluctuating items in a model and there may be many single data. Merging them with the rest of the model would interrupt the model structure with several numerical tables and the search for and modification of single data might be more difficult. In the sequence of 2, the data can be managed separately from the model. This is especially interesting if several people are working on a model - one 'making' the model and another gathering the data.

```
3. SET   < declare all sets >           {
   COEF  < declare all coefficients >   {  MODEL
   VAR   < define all variables >       {  STRUCTURE
   EQUATION < define all restrictions > {
   SET   < define all sets >           {
   COEF  < define all coefficients >   {  INSTANTIATION
   MODEL < now write it >              {
   END
```

Formulation 3 goes a step further in separating declaration and definition. The data as well as the members of indices are separated from the rest of the model. This sequence is interesting if a modeler builds not only one single model but a whole model class.

Now it is time to define some terms: The first four lines of form 3 defines a model structure. A model structure together with the definition of all indices and all coefficients is called a model instance. All model instances with the same model structure build a model class. See Geoffrion [4].
The final step in separating the declaration from the definition would be to declare all items first, before any definition - restrictions included. This may be of no interest at first glance, but if we look at model as 'a list of (relational) database-tables' or 'a set of relations', this might acquire some theoretical importance.
Any combination of the above formulations is possible in LPL. Some indices or data may be put within the model, whereas others - some big tables - may be defined separately. LPL gives the user the most flexible solution. The model can be arranged as the user wants.

5.9. THE FILE LPL.CFG

If the file LPL.CFG is present in the current directory, then this file is compiled *before* the model file. *LPL.CFG is treated as a include file at the very top of each model file*. The use of this file is, that the user can predefine different compiler directives ({$P, {$N, or {$M) and place these directives in the LPL.CFG file. In this case every model will be processed with the new directives. Of course, any directive in the model file replaces these directives again.

INDICES

This Chapter explains how indices are declared and used in LPL. Indices are the most important construct in the LPL language.

6.1. DEFINITIONS

An index is a finite collection of different elements. As in mathematics, indices are used to define multidimensional objects like vectors or matrices. An expression x_{ij} is said to have two indices i and j, both representing a collection of elements, say

$i = \{ 1, \ldots , m \}$ and $j = \{ 1, \ldots , n \}$.

The integers 1 to m are the elements of i and the integers 1 to n are the elements of j. The variable x_{ij} consists of m*n single variables, which - explicitly written - are

$$\begin{array}{llll} x_{11} & x_{12} & \ldots & x_{1n} \\ x_{21} & x_{22} & \ldots & x_{2n} \\ & \ldots\ldots\ldots\ldots & & \\ x_{m1} & x_{m2} & \ldots & x_{mn} \end{array}$$

In LPL, an index may also be called a set. This is justified by the fact that a set in LPL may not only contain a list of elements but also other sets (see below on Index-trees). The elements of a set are called members. From the point of view of database theory, an index defines a domain of elements. Therefore, we also use the term domain.
Unlike in mathematics, where the members are a range of positive integers, LPL also uses strings to designate members.
Indices must be declared in the Set Statement. It begins with the reserved word SET followed by the different set declarations separated by semicolons. Sets may be declared first and defined later on.
Example:

```
SET  i;  { declare an index named 'i' }
{ some other instructions here }
SET  i = ( 1:10 );  { define that index 'i' now }
```

A set is *declared* by writing just an identifier which designates the set followed by a semicolon. A set is *defined* by an identifier followed by an equal sign and a list of members separated by commas (optional). The list of elements must be included in parentheses. The number of members in a set is not limited. The number of members a set contains is called its cardinality. The order of the members within a set is, in

general, not important. LPL, however, also allows one to refer to the position of a member within the set. In this case the order of the members is important and the set is called an ordered set. Regardless of whether or not a set is ordered, LPL assigns a number to each member called position. The position of the first member is 1 and the position of the last member is the cardinality of the set.

6.2. INTEGERS AS MEMBERS

Integers (or reals) may be used to represent members.

Example:

```
SET IntSet = (1, 2, 3, 5); { 4 members: '1', '2', '3' and '5'.
                           { The commas are optional }
```

Reals, if used, are truncated. Note that the position of a member within a set is not necessarily the same as the member itself, even if they are represented by integers. In

```
SET i = ( 6 7 8 9 10 11 12 13 14 15 16 );
```

the member '6' has position 1 and the member '16' has position 11.

6.3. IDENTIFIERS AS MEMBERS

Identifiers may be more descriptive than integers to represent members.

Example:

```
SET seasons = (spring summer autumn winter);
```

The set 'seasons' contains four members: 'spring', 'summer', 'autumn' and 'winter'. The syntax of a member identifier is the same as for all other identifiers.

6.4. RANGES AS MEMBERS

A consecutive list of integers (a range of integers) can be written as

```
SET  IntegerRangeSet = (1:10); { contains 10 members }
```

which is the same as

```
SET  IntegerRangeSet = (1 2 3 4 5 6 7 8 9 10);
```

A colon must separate the two limits. Note that the limits are included in the members list.

A identifier list like

```
SET  IdentifierRange=(P1 P2 P3 P4 P5 P5 P6 P7 P8 P9 P10);
```

may also be abbreviated as

```
SET  IdentifierRange = (P1:P10);
```

A colon separates the two identifiers. Note that the heading string of both identifiers must be identical (in this case 'P'), which must be followed by a numerical string.

6.5. INDEX-TREES

As in mathematics, a set in LPL may not only be a collections of elements but also a collection of other sets. This defines a set in a recursive manner: *A set is a finite collection of sets or members*. Such a set is called an Index-tree in LPL. The syntax

for an Index-tree is the same as for any other index except that the sets within sets must also be bracketed. This syntax is similar to the syntax of a list in the programming language LISP.

Example:

```
SET i = ( A B ( Ba Bb Bc ( Bca Bcb ) Bd ) C );
```

The set 'i' contains 9 members. But two of them are very special members ('B' and 'Bc'). They denote sets and *may be used themselves as indices*. In fact the last example defines three sets: 'i', 'B' and 'Bc', where each can be used as index. The cardinalities of the three sets 'i', 'B' and 'Bc' are 9, 6 and 2. The position of a member is defined relative to the used set. 'Ba' has position 1 within set 'B', but has position 3 within set 'i'.

Members which are followed by a left parenthesis are called non-leaves; all others are called leaves. In the last example the members 'A', 'Ba', 'Bb', 'Bca', 'Bcb', 'Bd' and 'C' are the leaves of 'i'. 'B' and 'Bc' are the non-leaves of 'i'. Of course, 'Bca' and 'Bcb' are also the leaves of 'Bc' (but 'Bc' has no non-leaves). Every set contains members, which are called the children of that set. The children of a set are the members which follow within the next parenthesis pair by skipping all members within further parenthesis pairs. 'A', 'B' and 'C' are the children of 'i'. 'Ba', 'Bb', 'Bc' and 'Bd' are the children of 'B'. 'Bca' and 'Bcb' are the children of 'Bc'. If a member x is a child of set y, then y is called the parent of x. Hence, 'B' is the parent of 'Ba'. The use of Index-trees is explained in Chapter 6.12. Model examples using Index-trees are BUDGET.LPL, SOCCER.LPL (see Appendix D):

6.6. INDEXED SETS

Sets may also be indexed, as any coefficient or variable. In this case they declare or define a tuple list.

Example:

```
SET locations = ( NY BO LA );  "a list of plant locations"
SET links(locations,locations) = [ NY BO , NY LA , LA NY ]; "'Links' is the
      allowed list of connections between the locations, which is a subset of
      the Cartesian product (locationXlocation)"
```

For an example see the model SOCCER.LPL or the model LEARN6.LPL

6.7. ALIAS-NAMES OF MEMBERS

Every member in a set has a unique identification name called its alias. The default alias-names of members are '01', '02' .. in the order the members are entered in the set.

Example:

```
SET i  =  ( spring summer autumn winter );
```

The alias-names of the four members are '01', '02', '03', and '04'. The alias-names of members may be replaced by the user by two manner:

- by adding a alias-name after the member in single quotes

- by modifying <arg2> in the {$N compiler directive (see Chapter 9.8).

Examples:

```
SET i  =  ( spring summer autumn'0C' winter );
```

The alias-names now are '01', '02', '0C', and '04'.

```
{$Nn?} SET i  =  ( spring summer autumn winter );
```

The alias-names of the members now are: '1', '2', '3', and '4'.

```
{$Nn2} SET i  =  ( spring summer autumn winter );
```

The alias-names of the members are now: 'SP', 'SU', 'AU', and 'WI'.

```
{$Nn-3} SET i  =  ( spring summer autumn winter );
```

The alias-names of the members are now: 'ING', 'MER', 'UMN', and 'TER'.

The alias-names may contain any characters, but it must not exceed 3 characters in length. The unique use of this alias-name is the control of the production of the internal variable name (see Chapter 8-9).

6.8. DATA IN THE SET STATEMENT: THE FORMAT D

One-dimensional coefficients or variables can also be declared and defined in the Set Statement. The set identifier is followed by a list of coefficient identifiers separated by a bar '|'. Each member is followed by the data in the same order, also separated by a bar '|'. The whole may be arranged in a nice looking table format. This format is called Format D.

Example:

```
SET  i   |   a  |   b  |  c   =
 (   i1  |   .  |   2  |  1
     i2  |   1  |   .  |  0
     i3  |  2.4 |  -23 |  .
     i4  | -6.7 | 5.56 | 10.3
     i5  |   8  |  .   |  .
     i6
     i7  |   .  |   8  |   . );
```

'i' is a set containing 7 members and 'a', 'b' and 'c' are coefficient indexed over 'i'. The number of data after each member must correspond to the number of coefficient or no data may follow (see 'i6' in the last example). No data following the member means that the corresponding values are default values. A dot also indicates a default value. If an alias-name is added, the data must follow the alias-name. An example of this table format where i is an Index-tree gives the model BUDGET.LPL and ROBOTC.LPL.

The above example could have been written as (see Chapter 7)

```
SET   i    = ( i1 i2   i3    i4  i5  i6  i7  );
COEF  a(i) = /  .  1  2.4  -6.7   8   .   .  /;
COEF  b(i) = /  2  .  -23  5.56   .   .   8  /;
COEF  c(i) = /  1  0   .   10.3   .   .   .  /;
```

6.9. INDEXLISTS

Indices are used in LPL to define such multidimensional objects as coefficients, variables, and restrictions. In mathematical notation, subscripts are used to define such objects as x_{ij}. 'x' is said to be a two-dimensional matrix with m rows and n column, if

m is the cardinality of index i and n is the cardinality of index j. The number of subscripts following an object is called the dimensionality or the arity of that object.
In LPL, the subscripts are replaced by the same list of indices separated by commas and bracketed by '('...')' or '['...']'. So '(i,j)' and '[i,j]' are two legal constructs in LPL. Such a list is called Indexlist. Indexlists are used to define coefficients, variables and restrictions. They are also used together with Index-operators in LPL.
Examples:

```
COEF A(i,j);       { 2-dimensional data matrix }
VAR X(i,j,k,l,m);  { 5-dimensional variable }
MODEL R(i);        { 1-dimensional restriction }
..SUM(i,j,k)..     { 3-dimensional summation }
```

If the indices in an Indexlist are replaced by any member of that index, the list is called a tuple. The list of all distinct tuples is the complete tuplelist. This is the same as the Cartesian Product of all indices in the Indexlist. Therefore, an Indexlist defines a Cartesian Product and the number of all tuples in a complete tuplelist is the product of all cardinalities of the indices and is called the cardinality of the Indexlist. One order of the complete tuplelist is called the lexicographic order. It is obtained by placing first the tuple, where all members have position 1 in their sets and then placing the tuple by changing the members with subsequent higher position from right to left within the Indexlist.
Example:

```
SET  i = (1:3);
SET  j = (A1:A2);
COEF a(i,j);
```

'(i,j)' is an Indexlist, '(1,A1)' is a tuple, The complete tuplelist in lexicographic order is: (1,A1) (1,A2) (2,A1) (2,A2) (3,A1) (3,A2). Its cardinality is 6 (3x2).

Coefficients, variables, and restrictions may be indexed by just appending an Indexlist to the corresponding identifier. This defines or creates simultaneously many individual objects (coefficients, variables or restrictions). The number is the cardinality of the appended Indexlist. Every time an indexed object is used in LPL, we may think of a loop statement, which is executed in the background in lexicographic order of the tuplelist. But the user need not care about that; LPL does it automatically.
Example:

```
SET  i = (1:10);
COEF a(i) = b(i) + 2;
```

This assignment defines a coefficient 'a' over the set 'i'. The statement, however, assigns not one value, but ten values *simultaneously*, since the cardinality of the Indexlist '(i)' is 10.
Every Index-operator must also be followed by an Indexlist. Again, the cardinality of the Indexlist determines how many times the operator is executed.
Example:

```
SET i = (1:10);
SET j = (1:20);
```

```
COEF  a = SUM(i,j) b(i,j);
COEF  c(i) = SUM(j) b(i,j);
```

The SUM operator, in the first assignment, sums 200 (10x20) terms and the sum is stored in a single coefficient 'a'. In the second assignment, 20 terms are summed up and assigned to one of 'c'. This is done 10 times.

Sometimes, the same index must be used several times in the same Indexlist or in different Indexlists in the same expression. This produces some ambiguity in the expression. Suppose, the following assignment has to be evaluated:

```
COEF a(i,j,i) = b(i,i,j) + 2;
```

It is not clear how to evaluate this term. To avoid all ambiguities in such situations, the user may use dummy indices. A dummy index is an identifier which replaces a (global) indexname. It is used locally in an expression, much like a local variable in a programming language. The dummy precedes the index it replaces in the Indexlist and is followed by an equal sign.

Example:

```
COEF a(d1=i,j,d2=i) = b(d1,d2,j) + 2;
```

'd1' and 'd2' are two dummies for 'i', which is replaced in the subsequent expression. Dummies must be added to Indexlists, as long as ambiguity arises. But they can be used everywhere as well.

Example:

```
COEF  c(index1) = SUM(index2) b(index1,index2);  { without dummies }
COEF  c(i=index1) = SUM(j=index2) b(i,j);        { with dummies }
```

6.10. INDEXLISTS WITH CONDITIONS

Every Indexlist may be extended with a condition beginning with a '|' before the right parenthesis.

Example:

```
....[ i,j,k | i=k AND a(i)<>12 ] ...
```

The condition can be any legal expression. If the condition evaluates to zero, it means that the specific tuple does not exist or must be discarded. This limits the tuplelist and the resulting tuplelist is a subset of the complete tuplelist.

Example:

```
VAR X[ i,j | a(i,j) ];
```

This declares a variable for every tuple of '(i,j)', if the corresponding a(i,j) is not zero. Subsequent use of the variable 'X' discards automatically all non-existent tuples. Therefore, a subsequent 'SUM(i,j) X(i,j)' may produce fewer variables than the cardinality of SUM-Indexlist.

A mathematical example:

Suppose we have the following equation:

$$R_t: \quad x_t = y_t + \sum_{k<t}^{t} a_k z_k \text{ with } t = \{1..5\}$$

This can be written in LPL as (note that we use a dummy k)

```
SET    t = (1:5);
MODEL  R(t):  x  =  y + sum(k=t | k<t) a*z;
```

An more complex example can be studied in the variable declaration of 'Ship' in the 'DIST.LPL' model (see Appendix D).

6.11. APPLIED INDEXLISTS AND BINDING

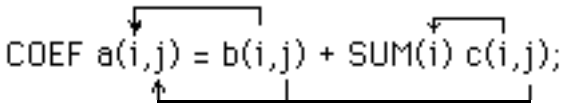
All Indexlists in expressions, except the Indexlists following an Index-operator, are called applied Indexlists. Applied Indexlists are appended to indexed coefficients and variables in expressions.

Example:

```
COEF a(i,j) = b(i,j) + SUM(i) c(i,j);
```

'(i,j)' following 'b' and 'c' are applied Indexlists.

Every index in an applied Indexlist must be bound to a previous index in an Indexlist. This is called index-binding. The next picture shows by arrows, how the bindings took place.



The index 'i' has been used two times in an Indexlist (after 'a' and 'SUM'). This may produce an ambiguity for the 'i' in the applied Indexlist after 'c': it is unclear which of the previous 'i's it is bound to. The LPL compiler interprets this automatically as shown by the arrows.

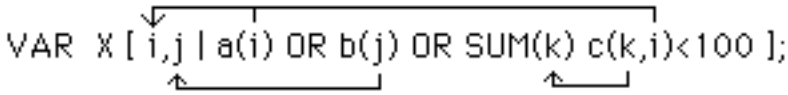
Bindings can be redirected by the user, if dummies are used. Dummies can eliminate any doubt about binding. Hence, the last example could have been written as

```
COEF a(i,j) = b(i,j) + SUM(k=i) c(k,j);
```

This eliminates any doubt about the binding of 'i' after 'c'.

Indices in a condition within the Indexlist must also be bound.

Example:



Applied Indexlists (or parts of them) may be dropped if there is no doubt how the indices have to be bound. Therefore the last two examples might have been written as:

```
COEF a(i,j) = b + SUM(i) c;
VAR  X [ i,j | a OR b OR SUM(k) c<100 ];
```

This makes the expressions much more readable.

The binding index and the bound index need not be the same identifier. LPL tries first to bind two identical index identifier; if it is not successful, it tries to bind the dummies to their corresponding global indices; if even that fails, it tries to bind to an

index, which has at least one common member with the binding index. Two set have common members, if the same member (i.e. the same spelling) is found in both sets. If the compiler fails on all three levels, it stops and displays an error.

An index in an applied Indexlist may also be replaced by a member of that index in quotes or by a number, which indicates the position of a member. (The quotes may be replaced by a beginning colon.)

Example

```
COEF b(i,j) = / ...data here... /;
COEF a(i) = b(i,'j4')
COEF a(i) = b(i,:j4)
COEF a(i) = b(i,4);
COEF a(i) = b(i,-1);
```

'j4' must be a member of 'j'. '4' says 'put here the fourth member in 'j'. If the number is negative, the position is counted backwards beginning with the last member.

An index in an applied Indexlist may also be extended by an Offset-operator and a number.

Two kinds of Offset-operators are possible: linear (+, -) and circular (++, --). The linear offset-operator adds or subtracts some positions to the index. If the calculated position outruns the definition, the specified term is discarded. No error is produced. The circular Offset-operator also adds or subtracts some positions, but instead of discarding the outrun values, it wraps around to the other end of the position.

Examples: (say i has n elements: 1 to n.)

```
SUM (i) x(i+1)  yields: x(2) + ... + x(n)
SUM (i) x(i-1)  yields: x(1) + ... + x(n-1)
SUM (i) x(i++2) yields: x(3) + ... + x(n) + x(1) + x(2)
SUM (i) x(i--2) yields: x(n-1) + x(n) + x(1) + ... + x(n-2)
```

6.12. INDEXLISTS AND INDEX-TREES

If an index in an Indexlist is an Index-tree, it can be preceded by a Select-operator. This operator acts as a selector on the all members of the Index-tree. Four select operators are available

- $: selects the children
- ! : selects the parent
- & : selects the leaves
- # : selects the non-leaves

Example:

```
COEF a($i,&j,#k);
```

declares a coefficient with three indices 'i', 'j' and 'k'. But the complete tuplelist is not selected, only a subset of it: from 'i' only the children, from 'j' only the leaves, and from 'k' only the non-leaves. This restricts the tuplelist to a subset.

If dummies are used, which are bound to other dummies, this is called dynamic binding.

Example:

```
MODEL Dyn(i=#IndexTree): SUM(j=$i)   X = X;
```

'i' and 'j' are dummies. Dummy 'j' is bound to the dummy 'i'. This produces a

restriction 'Dyn' for every non-leaf in 'IndexTree'. It sums all variables 'X', which are children of that specific non-leaf and equates it to the variable of that specific non-leaf. To study this mechanism see model 'BUDGET.LPL'.

DATA IN THE MODEL

All numerical values used in a LPL model are called data. Data may be added directly to any algebraic expression, as in '43 + 3/7' or '5*x'. Most data, however, are assigned to a coefficient in the Coefficient Statement, which is headed by the reserved word COEF followed by a coefficient identifier and a equal sign. Then follows the data in a predefined format or as an expression. Each declaration ends with a semicolon.

Example:

```
COEF  a = 10;
COEF  b(i,j,k) = 12;
```

The first definition assigns the value 10 to the identifier 'a'. 'a' may then be used in any expression replacing the value 10. The second definition assigns 12 to every tuple of 'b'. It is much like a loop statement in a programming language. In PASCAL this statement may be represented as the following sequence of instructions:

```
for i:=1 to m do
  for j:=1 to n do
    for k:=1 to o do
      b[i,j,k]:=12;
```

Other possibilities are available in LPL to enter data: four table formats, and assignment through expressions. In Chapter 6 we saw a table format, the Format D. It can be used only for one-dimensional coefficients. Now three other formats will be explained through the simple example:

```
SET i = ( A1:A2 );
    j = ( B1:B3 );
COEF a(i,j);
```

7.1. FORMAT A

For each tuple of coefficient 'a' a value is entered in lexicographic order surrounded by '/' and '/'. Optional commas may be used to separate the data. This format is very useful for small data lists. Dots may be used for default values.

Example:

```
COEF a(i,j) = / 10 20 . , 100 . 300 /;
```

7.2. FORMAT B

In this table format, the data may be entered in any order together with the member names forming a tuple: the list is enclosed in '[' and ']'.

Example:

```
COEF a(i,j) = [ A1 B1  10
                A1 B2  20
                A2 B3 300
                A2 B1 100 ];
```

Missing tuples are assumed to have a default value. They are zero, if no default value

has been defined.

7.3. FORMAT C

Format C is only useful for two-dimensional data tables. The table forms a matrix, where the rows are the members of the first index and the columns are the members of the second index.

Example:

```
COEF a(i,j) =
                |  B1    B2    B3 |
             A1 |  10    20     . |
             A2 | 100     .   300 | ;
```

The members of the second index j must be on the first line. The members of the first index i must be on the first column followed by a '|'. The members may be in any order. Each data line ends with a '|'. The table is surrounded by two '|'.

7.4. ASSIGNMENT THROUGH EXPRESSIONS

Any expression - simple as well as indexed - may be assigned to a coefficient.

Example:

```
COEF a = 2^4 + 4*6 + IF(2=1,2,4);   { 'a' is 44 }
COEF b(i) = c(i) + SUM(j) d(i,j);   { assign a list of values }
```

7.5. DEFAULT VALUES

Every coefficient can be declared with a default value by adding the reserved word DEFAULT and the value. The default value is the value of all tuples, to which no value has been explicitly assigned or which have been entered by a dot. If no default value was entered, LPL assumes zero.

Example:

```
COEF A(i,j) DEFAULT 100;
```

Furthermore, bounds may limit the range of the value of a coefficient. They are entered simply by adding '<' for a lower and '>' for an upper bound followed by a value. If no bounds have been specified, 1e30 and -1e30 are assumed.

Example:

```
COEF A(i,j) <1000  >100;
```

Coefficients may also be specified to be integers. Add the reserved word INTEGER. Otherwise reals are assumed.

Example:

```
COEF A(i,j) INTEGER;
```

Defaults, bounds, and the integer reserved word can be entered in any order after the coefficient or split into separate declarations.

Example:

```
COEF A(i,j) DEFAULT 100  <1000  >100  INTEGER;
```

is the same as the four following declarations

```
COEF A(i,j) INTEGER;
COEF A(i,j) DEFAULT 100;
```

```
COEF A(i,j) <1000;
COEF A(i,j) >100;
```

7.6. MULTIPLE DECLARATIONS

Like in the Format D, several coefficients may be listed in the same declaration.

Example:

```
COEF a | b | c | x (i,j)  DEFAULT 100;
```

This declares the four identifiers a, b, c, x, and that they are all two-dimensional over the indices (i,j). If one or all of them have been declared before in the model, then the default value 100 replaces the previous once for these identifiers. Furthermore, x may have been declared to be a variable, so x remains a variable. On the other hand, all of them must have been declared without an Indexlist or the identical Indexlist (i,j); otherwise an error occurs.

The table Format B is the only format (besides Format D), which also allows the user to assign values to more than one coefficient in the same declaration.

Example:

```
COEF a | b(i,j) = [ A1 B1  10  1
                   A1 B2  20  2
                   A2 B3 300  33
                   A2 B1 100  11
                   A2 B2   .  22 ];
```

meaning that the first value on a line is assigned to 'a' and the second to 'b'.

VARIABLES, RESTRICTIONS AND CHECKS

8.1. VARIABLES

All unknowns used in a LP model are called variables. Each variable used in a LPL model must be declared in the Variable Statement. Variables are declared in exactly the same way as coefficients except that a variable declaration is headed by the reserved word VAR.

Example:

```
VAR
  x;       {a single variable declaration named x }
  Name;    {another single variable 'Name' }
  y[i,j];  {an indexed variable y where i and j are sets}
  z[i,j,k,l];  {a four dimensional variable z}
  w[i | i<>5 ]; { with a condition }
```

Like coefficients, variables may also have default values, lower and upper bounds, or they may be of INTEGER type. Lower and upper bounds on variables directly produce a bound restriction in the output, and the INTEGER specification produces a mixed integer LP model. (The column Statement in the MPS file contains the specific markers, see modelLEARNC.LPL). Variables and coefficient in a model are treated identically: they have both numerical values, which can be used within any expression. One exception, however, persists: in the Model or Equation Statement, LPL tests, whether the expression is linear with respect to the variables and produces the output for the solver. But after the solver brings the values back, variables may be used in an expression as any other coefficient (e. g. in a Print Statement).

Example:

```
VAR
  x INTEGER;          { integer variable x }
  y(i)  <23;          { lower bound on y }
  z(i,j)  >34;        { upper bound on z }
  w INTEGER  >1  <10; { or all together }
```

8.2. RESTRICTIONS

The Model and Equation Statement of a LPL model contains the linear restrictions and the objective function of a LP model. The reserved word MODEL or EQUATION heads the definition, followed by a restriction identifier, a colon, and an expression containing variables. This ends with a semicolon.

Examples:

```
EQUATION t: x-y;
MODEL r: x+y = 2;
MODEL re(i): lo(i) <= xe(i) - SUM(j) ye(i,j) <= up(i);
MODEL MAXIMIZE x-y+z;
```

The expression may contain a maximum of 2 operators from the following list: '=', '<>', '>', '<', '<=', '>='. If two are used, the restriction is interpreted as range statement; if none of them is used, the expression is interpreted to be zero. Therefore, the first example above will be interpreted as 'EQUATION t: x-y = 0;'.

Since LPL accepts only linear equations, it is evident that non-linear expressions are not allowed. So expressions like x/y , x*y or even a*x/x will produce a compiling error.

Another point is important in this context: the same variable may be used several times in the restriction. LPL takes care of this automatically.

Example:

```
MODEL R: x + y =  2*x - 12*y;
```

will be translated by LPL as

```
MODEL R: 13*y - x = 0;
```

There is a difference between the Model and the Equation Statement. An Equation Statement does not produce any output to the .RES file, only the Model Statement does. Therefore, the structure of a restriction may first be defined in a Equation Statement and later on the restriction may be produced (printed to the .RES file) by a Model Statement.

Example:

```
EQUATION
  r1 : x+y+z <= a;
  r2 : x-y-z > b;
  r3 : 2*x - y -1 < d;
....{ more data may be defined here }
MODEL r1; r2; { now the two restrictions r1 and r2 are printed to the .RES
                file. r3 will be ignored at this point }
```

8.3. THE OBJECTIVE FUNCTION

The objective function may be placed anywhere within the Model or Equation Statements and has the same syntax as any other restriction except that it must begin with the reserved words MINIMIZE or MAXIMIZE.

Example:

```
MAXIMIZE profit:  x + y + z;
```

Several objective functions can be specified in a LPL model, or there may be none at all.

8.4. NOMENCLATURE

In an LPL program, any identifier up to 25 characters long may be a variable or a restriction name. These names are called the external names. Since MPS, for example, accepts variable and restriction names up to 8 characters only, the LPL nomenclature must be translated by the LPL compiler into a text string called the internal names, which must not exceed this 8 character limit in order to respect this MPS restriction. Furthermore, this translation must produce unique names for each row and column;

otherwise the names get confused.

LPL produces unique row and column-names by default. They have the format:

<integer containing three digits><integer>

Example: '10056'.

The first integer is a unique number - beginning with '100' - for each variable and restriction identifier declared, the second is a integer indicating the position in the Cartesian Product of its IndexList if the variable or restriction is indexed.

These names are not very expressive. The user, however, may control the production of these names by

- the mean of the {$N compiler directive (see in Chapter 9) and
- alias-names which is a quoted string following the variable or restriction identifier in the declaration section.

Every variable and restriction has a unique alias-name which defines how the internal name are produced. The alias-name - automatically produced by the {$N directive - can be replaced by the user, if he places a single quoted string in the declaration.

Three character used within this string have special meaning:

'a' followed by a digit means: add an alias of a member.

'i' followed by three digit means: extract some characters from a member and add this to the name.

'n' add a unique number to the internal name.

Example:

```
SET i = (1:10); j = ( spring summer autumn winter );
VAR X'Xa1Ti213X'(i,j);
```

means: X has the following names: take an 'X' and add the alias of the first index ('01', '02', '03', or '04'), add a 'T', then add the third character of the member of the second index and finally add a 'X'. An example of a produced name is: 'X01TSPRX'.

```
SET i=(1:10);  VAR AMOUNT'Xn'(i);
```

The alias-name of variable AMOUNT is 'Xn'.

```
{$N2} VAR AMOUNT(i);
```

The alias-name of variable AMOUNT is now 'AMn'.

```
{$N-3} VAR AMOUNT(i);
```

The alias-name of variable AMOUNT is now 'UNTn'.

Any alias name is ignored if no {$N directive is used within the model. Note that all {$N directives must be erased if the directive {$S is used.

8.5. CHECK STATEMENT

A check statement can be added at any place where another statement is allowed. It allows the user to check a condition, and if it is false (evaluates to zero!), the compilation stops with a user error message.

Syntax:

```
CheckStatement ::= CHECK [ IndexList ] : Expression [ QuotedString ] ';'
```

Examples:

```
check: a<>0  'A must be different of zero' ;
check(i,j): a(i,j) > 1  'all a(i,j) must be greater than one' ;
check(i): SUM(i) 1 > 20 'set I must have at least 20 members' ;
```

The first CHECK tests, whether 'a' is different of zero. If this is not the case, then the error message "A MUST BE DIFFERENT OF ZERO" is displayed and the compilation stops.

The second CHECK tests all a(i,j), and if one is less or equal 1 the compiler stops and displays the message. The second check could also be written as:

```
check: FORALL(i,j) (a(i,j) > 1)  'all a(i,j) must be greater than one' ;
```

The third check stops the compiler if set 'i' has less than 20 members.

For an example of the check statement see the model DIST.LPL. The two check statement make sure that all distribution centers are also warehouses and all factories are also distribution centers.

Note, however, that the QuotedString is actually limited to 25 characters. Note also, that the check statement is executed at parse time and not at the end of the parse. That means, that all items used in the expressions must be defined before in the model.

8.6. DELETE STATEMENT

A Delete Statement can be added at any place where another statement is allowed. It begins with the reserved word DELETE followed by a identifier or a '*' character and ends with a semicolon.

The syntax:

```
DELETE identifier | '*' ';'
```

This statement allows to delete coefficients, variables or restrictions of the model which are no more needed in the subsequent part. The Delete Statement frees the heap in order to allow more tables to be entered into a model.

Example:

```
COEF a(i,j,k); b(i,j); c(i);
...{ define now a, b and c and use them }...
DELETE a; b;  { delete now the two tables a and b }
```

The identifier may be replaced by a '*'. This means that all previous restrictions declared within a Equation or a Model statement are deleted.

COMPILER DIRECTIVES

A compiler directive in LPL is any comment beginning with the the following two characters '{$'. They have special meanings for the LPL compiler:

```
{$C}                          Case sensitivity on
{$M <maxs> <maxa>}          Memory requirements
{$R<digit>}                 Random number initializing
{$I <filename>}               File include
{$Bxy}                        Binding strictly on and off
{$X <prog> <param>}         Execute a child process
{$P <SIP>}                  Solver Interface Parameters
{$S [<SIP>]}                  LP or MIP solver Call
{$N<arg1><arg2> }           Nomenclature
```

A compiler directive may be placed anywhere within the model where a normal comment is legal - except the first two ({$C and {$M) which must be placed before any statement, normally before the first SET.

9.1. CASE SENSITIVITY {$C}

Case sensitivity means that lower and upper case letters are distinct by the compiler. The LPL compiler is not case-sensitive by default. Identifiers, however, can be made case-sensitive by the compiler-directive {$C.

Example:

```
{$Case sensitive}
```

If case sensitivity is on, it is important that all reserved word are written entirely in upper-case. Note that this compiler directive must be added within the model at the very top (or in the LPL.CFG file).

9.2. MEMORY REQUIREMENTS {$M}

The heap memory manager of LPL reserves some space for different data. The user may change their values with the {$M directive which is extended by two integers:

```
{$M <maxs> <maxa> }
```

The two numbers preset the maximal number of sets and members and the maximal number of numeric data

Example:

```
{$M 500 5000 }  {this is the default}
```

This directive is only needed, if the compiler stops with an error number 555, or 556. In this case, the appropriate integer must be bigger than the default value above. Note that in the PASCAL Version this directive has no effect.

9.3. RANDOM NUMBER INITIALIZING {$R}

The built-in random generator can be initialize with the compiler directive {$R}. Any comment is allowed within this directive as long as the directive begins with R.

Examples:

```
{$R initialize the random generator}
{$Randomize}
```

The directive may be extended by a digit which must follow the 'R' immediately.

Example:

```
{$R1}
```

This imposes the random seed which reproduce a sequence of the same random numbers every times. If no digit follows the 'R' the random seed is arbitrarily chosen by the system. To study an example see model fragment LEARN2.LPL.

9.4. FILE INCLUDE {$I}

The {$I directive redirects the LPL compiler to read from another file at this point and - at the end of that file - reading continues from the calling file. Hence, the model may be split in different LPL partial models. For an example see the ROBOT6.LPL model. Nested Include files are also possible. This means, that in an included file there may be further calls to included files through the '{$I' compiler directive. The filename may be entered with or without single quotes.

Example:

```
(in main file)          .... {$I include.dat } ...
(in include.dat file)   .... {$I 'include2.dat' }
```

The nesting depth is limited, however, to 5 levels.

9.5. BINDING STRICTLY ON AND OFF {$B}

LPL has a relaxed syntax for index binding and IndexList dropping by default. A simple example is: 'SUM(i) a(j)'. If 'i' and 'j' have some common members, then the summation takes place over the common members, since 'j' is bound to 'i'. This is called relaxed binding. Furthermore, the IndexList '(j)' after 'a' may even be dropped, if 'a' has been declared as 'a(j)'. Therefore, 'SUM(i) a' is exactly the same as 'SUM(i) a(j)'. This relaxed syntax has a price: error checking also is 'relaxed'. If the user wants to impose the compiler to check some additional errors, the directive {$Bxy} may be added at different places within the model, where 'x' and 'y' are any two characters. If 'x' is '+' then strict binding is enforced which means, that the bound index must be the same as the binding index. If 'y' is '+' then Index-list dropping is no allowed. Any other two characters mean a relaxed syntax.

Examples:

```
{$B++} strict binding enforced and dropping is no allowed
{$B+-} strict binding enforced and dropping is allowed
{$B-+} no strict binding and dropping is no allowed
{$B--} relaxed syntax (this is the default)
```

Therefore, '{B++} SUM(i) a' produces an error. It must be replaced by:

'{B++} SUM(dummy=i) a(dummy)'

9.6. EXECUTE A CHILD PROCESS {$X}

Any program may be executed from within an LPL model with the {$X directive. The 'X' must be followed be two strings indicating the program-name and its parameters.
Example:

```
{$X lplmps.exe model1 }
```

This instruction executes the LPLMPS with the parameter 'model1'. Note that the extension of the program-name must be added. (See model AMPL.LPL for an example).

9.7. THE SOLVER INTERFACE PARAMETERS {$P}.

LPL has a flexible and transparent solver interface. The communication between LPL and the LP/MIP solver is determinated by 12 interface parameters (explained below). By default, LPL calls the XA solver from SUNSET [12] without any intervention from the user. The user must only place a {$S} compiler directive in the LPL model and the LPL compiler does the whole work: producing the MPS file, calling the solver with the right parameters, and reading the solution file. Then it continues the compilation. This communication is 'hardwired' within the LPL compiler.
But the user may call the solver in a different way, or he or she has a different solver. In this case the user has three possibilities to change the solver interface:

1. include the interface parameters within the {$S directive
2. include the interface parameters within the {$P directive
3. 'hardwire' the interface parameters within the LPL.EXE code.

The first solution is useful, if the user wants occasionally change the interface parameters (see modelfile MAGIC.LPL). The second solution may be adopted, if the the user wants to include the parameters within the LPL.CFG file. In this case every run overwrites the hardwired interface parameters, since the LPL.CFG is compiled before every model file. The third solution is useful, if the user wants to change the parameters permanently. The LPL.EXE must be changed at the position where the text 'LPLMPS.EXE....' begins. This space (255bytes) is reserved for the interface parameters. The user should add a comma and a ASCII character 0 at the end of the string. The byte just before 'LPLMPS...' should be changed to the length of the string.

LPL needs 12 parameters in the following order to communicate with the solver (the default value is added in parenthesis):

1. the program name which produces the MPS input file (LPLMPS.EXE)
2. the parameters of that program (%1)
3. the program name of the solver (XA.EXE)

4. the parameters of that program (TEMP.CLP)
5. the name of the solver parameter file (TEMP.CLP)
6. the content of the solver parameter file (temp.mps\nACTIVITY temp.sol\nOUTPUT temp RHS ..rhs BOUNDS Bounds LISTINPUT NO\nOBJECTIVE %2 %3)
7. the string which will be added to the solver parameter file, if the model has to be maximized (MAXIMIZE YES)
8. the string which will be added to the solver parameter file, if the model has to be minimized ( )
9. the file name from which LPL has to read the solution (TEMP.SOL)
10. An integer indicating on which physical position on a line in the solution file the first character of the variable-name is found (2)
11. An integer indicating on which physical position on a line in the solution file the first digit of the value is found (12)
12. An integer indicating the length of the numerical value of the variable in the solution file (11)

All parameters may contain the following strings:

'\n' : which will be translated to carrige-return and new-line

'%1' : which is replaced by the model name

'%2' : which is replaced by the objectif function name

'%3' : which is replaced by the seventh or eighth parameters above, depending whether the model is to be maximized or minimized.

Any other characters or strings are taken litterally.

The parameters 9 to 12 are used to read the solution back to the LPL system. They suppose that the solver write his solution to a file (by default to TEMP.SOL) which contains one variable-name per line with its value. Any line which does not contain a variable-name is ignored.

Example: The following is the XA solver output file TEMP.SOL for the SIMPLE.LPL model (last rows are cut):

```
"U N B O U N D E D   V A R I A B L E 1021","temp    ","Thu Jul 19 12:23:40
     2,     3,  260.00000,      3
"1001    ",  254.00000,    1.00000,    0.00000,"IN   ","(NB)    ",   -0.000
"1011    ",    6.00000,    1.00000,    0.00000,"IN   ","(NB)    ",   -0.000
"1021    ",    0.00000,    1.00000, 1981.00000,"LOWER","(NB)    ",   -0.000
"1031    ",   -4.00000,   -4.00000,    1.00000,"LOWER","EQ","1001    ", -25
"1041    ",    6.00000,    6.00000,   44.00000,"LOWER","GE","1001    ",
```

The lines 1, 2, 6, and 7 are ignored, since the position 2-9 are not variable-names. LPL reads only the bold values of the remaining lines.

The solver interface parameters must be entered by the directives {$S or {$P in this order and separated by commas. The whole string must by in quotes. The string may be broken in several lines. In this case, every part on the line must be in quotes. Any text outside the quotes is treated as an ordinary comment.

Example:

```
{$P this is the default parameter list :
    'LPLMPS.EXE,%1,XA.EXE,temp.clp,temp.clp,'               (parameters 1-5)
    'temp.mps\nACTIVITY temp.sol\nOUTPUT temp '
    'RHS ..rhs BOUNDS Bounds LISTINPUT NO\nOBJECTIVE %2 %3,' (parameter 6)
    'MAXIMIZE YES, ,temp.sol,2,12,11'                       (parameters 7-12)
}
```

For the CPLEX solver these parameters might be:

```
{$P 'LPLMPS.EXE,%1,'
    'CPLEX3.EXE,<temp.clp,'
    'temp.clp,'
    'r temp.mps\n s d 0\n s l temp.sol y\n %3\n opt,'
    'c s 0 ma, ,'
    'temp.sol,1,25,19'
}
```

For the HYPER-LINDO solver they might be:

```
{$P 'LPLMPS.EXE,%1,'
    'LINDO.EXE,<temp.clp,'
    'temp.clp,'
    'BATCH\n DIVERT temp.sol\n RMPS temp.mps\n %3\n GO\n NO\n QUIT,'
    'MAX,MIN,'
    'temp.sol,4,13,12'
}
```

9.7. SOLVE A LINEAR MODEL {$S}.

This directive is the interface to a LP/MIP solver. It calls LPLMPS and produces the MPS input file. Then it calls the XA solver to solve the model, and, finally, it reads the values of the variables back into the LPL (if the solver has found a solution), so that this data may be printed by the report generator. The solver interface parameters may also be added within this directive like in the {$P directive.

Examples:

```
{$S}
{$Solve this model now}
{$Solve now comes the interface parameters ......}
```

Note, that the values of the variables can only be read back, if no {$N directive is used within the model. So erase any {$N directive, before using the {$S directive.

9.8. NOMENCLATURE {$N}

The compiler directive {$N determines how the alias-names (explained in Chapter 8) are produced automatically.

The syntax of '{$N' is:

```
{$N<arg1><arg2> ( any comment may follow here) }
```

where <arg1> is the character 'n' or a digit between -9 and 9 and <arg2> is 'n', '?', '??' , '???' or a digit between -9 and 9.

The compiler directive '{$N' may be placed several times in a model file, each replaces the former directive. Each set, variable or restriction can have its own directive. An example can be studied in the TEST8.LPL model fragment.

Examples:

```
{$Nnn}    <arg1> is 'n',<arg2> is 'n' : this is the default directive.
{$Nn1     <arg1> is 'n' and <arg2> is '1'}
{$N3??}   <arg1> is '3' and <arg2> is '??'}
{$Nn-3}   <arg1> is 'n' and <arg2> is '-3'}
{$N-1?}   <arg1> is '-1' and <arg2> is '?'}
{$N2}     omits <arg2>, which does not change.
{$N any text}    this directive will be ignored
```

<arg1> is

- 'n', which means: produce a unique integer for each variable's and restriction's alias-name. This is the default.
- a digit i, which means: produce a alias-name for each variable and restriction composed of the first i characters of the variable's or restriction's name. If the digit is negative, the last i characters are taken instead. Lower case letters are translated in uppercase letters.

<arg2> is

- 'n', which means: add an unique integer to the alias-name of a variable or restriction, if they are indexed.
- '?', which means: produce alias-names for each member of a set. They are produce as: '1' .. '9', 'A' .. 'Z' .. (ASCII-characters with higher ordinal position) in this order. Add them to the variable's or restriction's alias-name.
- '??', which means: produce alias-names for each member of a set. They are produce as: '01', '02' .. '99' , 'A0' .. 'Z9' .. in this order. Add them to the variable's or restriction's alias-name.
- '???', which means: produce alias-names for each member of a set. They are produce as: '001', '002', .. in this order. Add them to the variable's or restriction's alias-name.
- a digit i, which means: produce alias-names for each member of a set composed of the first i characters of the member's name. If the digit is negative, the last i characters are taken instead. Lower case letters are translated in uppercase letters (unless {$C} is on). Add them to the variable's or restriction's alias-name. All variables and restriction alias-names are restricted to 15 characters. (see also Chapter 8.4).

Again, do not use any {$N directive in the model together with the {$S directive, since otherwise LPL will not be able to read the solver output file.

REPORT GENERATOR

Two statements give the user a powerful tool to produce output tables from an LPL model in any format: he Mask and the Print Statement. Together, they define the Report Generator of LPL.

10.1. MASK STATEMENT:

A Mask Statement can be added at any place where another statement is allowed. It begins with the reserved word MASK followed by a mask-identifier. The content of the mask begins at the beginning of the next line and extends to the reserved word ENDMASK. An semicolon must end the Mask Statement.

The syntax:

```
MASK MaskName
< Content of the mask >
ENDMASK ;
```

The Mask Statement allows the user to define the layout or the format of a specific Print Statement (explained below). The content of the mask may contain any characters (tabs and returns included) and may extend over several lines. Only the two characters '$' and '#' have a special meaning. A consecutive stream of $'s in the mask is called an alphanumeric place-holder, and a consecutive stream of #'s together with an optional dot in its middle is called a numeric place-holder. A subsequent Print Statement prints the mask content exactly as defined by the Mask Statement, where the place-holders are filled with alphanumeric and numeric data.

Example:

```
MASK m1
Here begins the mask. Now comes a alphanumeric place-holder: $$$$$$$$EndLine

This is the third line of the mask. (line two is empty).
######.#####  $$$$$$$######## $$$$$####.###
Line four contains five place-holder (two alphanumeric).
ENDMASK;
```

The place-holders can be filled with one single item or with a whole array of items.

An complete example with the Mask and Print Statement can be studied in the models LEARN8.LPL and LEARN9.LPL.

10.2. PRINT STATEMENT

A Print Statement can be added at any place where another statement is allowed. It begins with the reserved word PRINT. The Print Statement is a very powerful table generator. Three different kind of prints can be used with LPL: print simple tables, print expressions, and print expressions togethter with masks.

Syntax:

```
PRINT Identifier [Format ] [ '(' Expression ')' ] ';'
| PRINT [ IndexList ] ':' Expression [ Format ] ';'
Format := ':' Integer [ ':' Integer ]
```

The simplest report can be made by using the PRINT keyword followed by a identifier. This identifier may represent a data table, a variable (indexed or not) or a restriction.

Example:

```
PRINT a; b;
```

A format specification may be added to the identifier which determines the width of the data. The first integer after the colon limits the total width and the second the number of decimal places.

Example:

```
PRINT a:12:7; b:3;
```

If the user may print an (indexed) expression, the second kind of print may be used. The PRINT keyword is followed by a optional IndexList, by a colon and an expression.

Example:

```
PRINT : a*b; { if a and b are not indexed }
PRINT(i,j) : a(j,i); { note the order of the indices }
PRINT(i) :  SUM(j) a; {an expression }
```

Note, that this Print Statement requires a colon which separates the word PRINT from the expression. Examples can be seen in different models. Again a format specifier may be added to the expression as in the first example.

If the format and the layout of the report is more complex, then the user has the possibility to use the print together with a mask. The reserved word PRINT is followed by a mask-identifier. This identifier is followed by a expression surrounded by parentheses. The Print Statement ends with a semicolon. The identifier must be defined as a mask before used in a Print Statement. The mask-name defines the lay-out of the print. The expression (which is, in general, an expression-list) returns the values which are inserted into the mask at the place-holders. There must be at least as many place-holders in the mask as expressions separated by commas in the Print Statement. The return values of the expressions are inserted into the mask from left to right and top to bottom.

Expressions in the Print Statement have a slightly different behavior as expressions in other parts of the model:

- The index-operators COL and ROW are only allowed in the Print Statement.
- Quoted strings are allowed in the Print Statement only. In this case the expression returns the string itself instead of a numeric value.
- If a set-identifier is used, the expression returns the member-name as a string if the

corresponding place-holder is alphanumeric (and not its position as a numeric value).

Example:

```
PRINT m1( 'First' , 345.67 , 'Second' , 2315.78,'Third' , 23 );
```

together with the mask m1 (defined in the last section 10.1) produces the following output:

```
Here begins the mask. Now comes a alphanumeric place-holder: First   EndLine

This is the third line of the mask. (line two is empty).
   345.67     Second      2315Third   23.
Line four contains five place-holder (two alphanumeric).
```

Alphanumeric place-holder are flushed left, whereas numeric are flushed right. The insertion takes place from left to right within the print expression. They are inserted from left to right and from top to down within the mask. 'First' is inserted at the first alphanumeric place-holder on the first line, '345.67' is inserted at the next place-holder at the third line, etc. But the real power of the Print Statement comes from the two new index-operators COL and ROW. The COL index-operator extends a place-holder horizontally, whereas the ROW index-operator extends a place-holder vertically.

Example:

```
SET i=(P1:P4);
MASK m2
   begin x$$$$$end
ENDMASK;
PRINT m2(COL(i) i);
PRINT m2(ROW(i) i);
```

The first print produces the following output to the report file .NOM:

```
   begin xP1   xP2   xP3   xP4   end
```

Note that the last character just before the place-holder is repeated too. This has the important function of a separator.

The second print produces the following output:

```
   begin xP1   end
   begin xP2   end
   begin xP3   end
   begin xP4   end
```

Note that the whole line of the place-holder within the mask is also repeated with the ROW operator.

APPENDIX A: COMPILER ERROR MESSAGES

The following is a listing of the error messages of the integrated editor and the LPL compiler. When the compiler encounters a syntax error, it will stop and enter the editor, if the file LPLEDIT.COM is found. It prints the error message and the user is placed at the position in the LPL model where the error occurred. All error messages are collected in the file LPL.MSG, which must always be in the same directory as the file LPL.EXE.

```
500  ; expected
501  / expected
502  Identifier expected
503  File not found
504  Dummy index stack overflow
505  ) expected
506  Number expected
507  : expected
508  ' (quote) expected
509  ( expected
510  Set (or member) already defined
511  SetIdentifier expected
512  Statement begin expected
513  IndexList does not correspond with a previous declaration
514  Empty model restriction is illegal
515  Only one declaration allowed here
516  Unknown identifier
517  Only allowed as second argument of IN operator
518  This is not a linear equation
519  Maskname expected
520  Incorrect arguments with IN operator
521  Only two indices allowed here
523  Expression stack overflow
524  Number is out of range
525  Negative or zero argument in LOG function
526  Negative argument in SQRT function
527  Incorrect Set range
529  Set member expected
530  String too long
531  Unknown set member
532  Identifier already defined
533  Illegal number of indices
534  Not allowed operator in expression
535  | expected
536  No binding is possible, use dummies
538  Ambiguous binding, use dummies
539  Include file not found
540  Missing index-list ( {$B is on )
541  Unexpected end of file
543  Use ?index in previous IndexList
544  Define two Restriction-names for a range
545  Syntax error in expression
546  , (comma) expected
547  Not allowed empty set
549  Exponent of negative number
550  Divide by zero
551  ! is not allowed here
552  User defined error in Check Statement
553  This {$ directive must be at the beginning of the model
554  Syntax is: "LPL <ModelName> [COption]"
555  <lmax> in {$M directive is too small
556  <mmax> in {$M directive is too small
```

```
558  Parameter in mask must be a string
559  Parameter in mask must be a numeric
560  Too many parameters
```

APPENDIX B: LPL SYNTAX

The syntax of LPL (linear programming language) is presented here using the formalism known as the extended Backus-Naur form. The following symbols are metasymbols, unless they are included in quotes (as '{' or '['). A single quote is written as: ''' (three quotes).

::=	means "is defined as" (defines a production)
|	means "or"
{ }	enclose items which may be repeated zero or more times
[ ]	enclose items which may be repeated zero or one times
...	'A' | ... | 'Z' means "all letters from 'A' to 'Z'"

All other symbols are part of LPL. Reserved words are in upper case letters. The beginning symbol is 'model'. Lower- and uppercase letters within the syntax are treated identically. Comments can be inserted anywhere within the syntax.

```
Model ::= [ ModelHeading ] Statement { Statement } END
Statement ::= SetStatement | CoefStatement | VarStatement | ModelStatement
              | EquStatement | CheckStatement | PrintStatement | MaskStatement
              | DeleteStatement
ModelHeading   ::= PROGRAM Identifier [ ';' ]
SetStatement   ::= SET { Set ';' }
CoefStatement  ::= COEF { CoefOrVarDeclaration ';' }
VarStatement   ::= VAR { CoefOrVarDeclaration ';' }
ModelStatement ::= MODEL { Restriction ';' }
EquStatement   ::= EQUATION { Restriction ';' }
CheckStatement ::= CHECK [ IndexList ] : Expression [ QuotedString ] ';'
PrintStatement ::= PRINT Identifier [ Format ] [ '(' Expression ')' ] ';'
                   | PRINT [ IndexList ] ':' Expression [ Format ] ';'
MaskStatement  ::= MASK Identifier AnyString ENDMASK ';'
DeleteStatement ::= DELETE Identifier | '*' ';'

Set ::= SetIdentifier [ QuotedString ] { '|' Identifier  } [ '=' MemberList ]
        | CoefOrVarDeclaration
MemberList ::= '(' Member { [','] Member } ')'
Member ::= Item [ ':' Item ] [ QuotedString ] { '|' Data }
           | SetIdentifier [ = ] MemberList
Item ::= Identifier | Number

Format := ':' Integer [ ':' Integer ]
CoefOrVarDeclaration ::= IdentSpec [ '=' DataList ]
IdentSpec ::= IdList [ Indexlist ] { Options }
IdList ::= Identifier { '|' [ COEF | VAR | MODEL ] Identifier }
Options ::= INTEGER | DEFAULT Number | '<' Number | '>' Number
DataList ::= AList | BList | CList | Expression
AList ::= '/' Data { [ ',' ] Data } '/'
BList ::= '[' TupleEntry { TupleEntry } ']'
CList ::= '|' SetIList '|' DataLine { DataLine }
TupleEntry ::= SetIdentifier { SetIdentifier } { Data }
SetIList ::= SetIdentifier { SetIdentifier }
DataLine ::= SetIdentifier '|' Data { Data } '|'
Data ::= Number | dot

Indexlist ::= lpar index { ',' index } [ '|' Expression ] rpar
lpar ::= '(' | '['
rpar ::= ')' | ']'
index ::= [ dummy '=' ] [ select ] path
```

```
dummy ::= Identifier
select ::= '&' | '#' | '$' | '!'
path ::= SetIdentifier { : SetIdentifier }

Expression ::= [ IndexOperator IndexList ] Factor { RelOp Factor }
unary ::= '+' | '-' | NOT
IndexOperator ::= SUM | FORALL | EXIST | SMIN | SMAX | PROD | COL | ROW
Factor ::= [ unary ] number
          | [ unary ] VarOrCoefIdentifier [ AIndexList ]
          | [ unary ] function
          | [ unary ] '(' Expression ')'
          | SetIdentifier
          | QuotedString
RelOp ::= ',' | '|' | OR | AND | IN | '=' | '<>' | '<' | '>' | '<=' | '>='
         | '+' | '-' | '*' | '/' | '^'
function ::= IF '(' Expression ';' Expression [ ';' Expression ] ')'
             | MathFunc '(' Expression ')'
             | MathFunc2 '(' Expression ';' Expression ')'
MathFunc ::= ABS | CEIL | FLOOR | TRUNC | SIN | COS | LOG | SQRT
MathFunc2 ::= MAX | MIN | RND | RNDN

AIndexlist ::= lpar AIndex { ',' AIndex } rpar
AIndex ::= [ select ] path [ OffsetOp number ]
          | number [ OffsetOp number ]
          | ''' SetIdentifier ''' [ OffsetOp number ]
          | : path  [ OffsetOp number ]
OffsetOp ::= '+' | '-' | '++' | '--'

Restriction ::= [ MAXIMIZE | MINIMIZE ] ResSpec [ ':' Expression ]
ResSpec ::= Identifier [ '|' Identifier ] [ Indexlist ]

QuotedString ::= ''' AnyString '''
AnyString ::= Char { Char }
Char ::= Digit | Letter | Special
SetIdentifier ::= Identifier
Identifier ::= Letter { LetterOrDigit }
LetterOrDigit ::= Letter | Digit
Letter ::= '_' | 'A' | ... | 'Z' | 'a' | ... | 'z'
Digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
Special ::= '+' | '-' | '*' | '/' | '=' | '<' | '>' | '(' | ')' | '['
           | ']' | '{' | '}' | . | ',' | ':' | ';' | ''' | '#' | '&'
           | '^' | '$' | '?'
dot ::= '.'
Number ::= [+ | -] UnsignedNumber
UnsignedNumber ::= UnsignedInteger [. UnsignedInteger]
UnsignedInteger ::= Digit { Digit }

Comment ::= '{' AnyString | CompilerDirective [ AnyString ] '}'
           | '(*' AnyString '*)' | '"' AnyString '"'
CompilerDirective ::= '$I'<filename>} | '$C' | '$R'<digit>
                     | '$B'<two characters> | '$M'<four integers>
                     | '$X'<two strings> | '$N'<arg1><arg2>
                     | '$S' [<SIP>] | '$P' <SIP>
```

APPENDIX C: THE EDITOR

LPL has an integrated editor, which is like WordStar or the editor with TURBO PASCAL. A compiling error automatically executes to the editor with a syntax error message at the position where the error occurred. The editor is a RAM editor and, therefore, limited.

The following commands are available with this editor:

```
<DEL>              delete last character
<ESC>              undo last deletions
Ctrl a             left word
Ctrl c             page down  (<PgDn>)
Ctrl d             left character  (<left arrow>)
Ctrl e             line up  (<up arrow>)
Ctrl f             right word
Ctrl g             delete right character
Ctrl h             destructive backspace
Ctrl i             tabulation  (<TAB>)
Ctrl j             begin/end of line (<HOME> <END>)
Ctrl k             utility suppercommand
    Ctrl kb begin block            Ctrl ks save file
    Ctrl kc copy block             Ctrl kt define tab width
    Ctrl kd save and exit          Ctrl kv move block
    Ctrl kh hide block             Ctrl kw write block to disk
    Ctrl kk end block              Ctrl kx exit editor
    Ctrl km set marker prompt      Ctrl ky delete block
    Ctrl kr read file into window  Ctrl k (1..9)   set marker #
Ctrl l             replace or find next
Ctrl m             insert new line  (<RET>)
Ctrl n             insert new line  (<RET>)
Ctrl p             insert control character
Ctrl q             quick supercommand
    Ctrl qa find and replace
                   Options are: g (general), b (backwards), u (uppercase)
                   w (whole words), n (automatic replacement)
    Ctrl qb cursor to beginning of block   Ctrl qj  jump to marker (prompt)
    Ctrl qc cursor to end of file          Ctrl qk  cursor to end of block
    Ctrl qd cursor to end of line          Ctrl qr  cursor to top of file
    Ctrl qf find pattern                   Ctrl qs  cursor to begin of line
    Ctrl qi toggle autoindentation         Ctrl qy  delete to end of line
Ctrl r             page up  (<PgUp>)
Ctrl s             left character  (<left arrow>)
Ctrl t             delete right word
Ctrl u             interrupt last command
Ctrl v             toggle insert mode
Ctrl w             scroll up
Ctrl x             down a line  (<down arrow>)
Ctrl y             delete the current line
Ctrl z             scroll down
```

APPENDIX D: LPL EXAMPLES

Here a short list of some LP models in LPL formulation. All examples are also on disk.

```
(* GENERAL.LPL: the general LP model with i rows and j columns,
   the A(,i,j) coefficient matrix, b(i) as right hand side,
   the c(j) cost vector and the X(j) variables. *)

PROGRAM  General_LP_Model;

SET  i; j;
COEF A(i,j);  b(i);  c(j);
VAR  X(j);
EQUATION restriction(i):  sum(j)  A*X <= b;
  MAXIMIZE obj_function:  sum(j)  c*X;

{ ----------------- the data follow now ----------------- }
SET
  i = (1:30);    { thirty rows declared }
  j = (1:20);    { twenty columns declared }
COEF
  A(i,j) =
       |  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20  |
  {----|-------------------------------------------------------|-}
    1  |  . 4 . 3 . . 6 . .  . 31  5  .  .  4  3  .  .  0  .  |
    2  |  7 9 . . . 3 . 4 .  9  .  4  1  .  3  .  1  5  0  .  |
    3  | 37 9 4 . 4 3 . . .  . 32  .  .  .  .  1  4  4  .  5  |
    4  |  . . . 3 . 5 6 . .  9  .  .  4  3  .  .  .  .  .  4  |
    5  | 14 . . . 4 5 . 4 8  9 37  .  .  .  .  3  6  .  .  .  |
              {  ... some rows are cut ... }
   20  | 20 . 4 . 4 . 9 4 8  . 10  .  .  .  .  .  .  7  0  .  |
   21  |  . . 2 . 4 . 9 6 8  . 20  .  5  3  .  .  .  .  .  4  |
   22  | 63 1 . . . . . 4 0  9  .  .  6  3  .  .  1  .  0  .  |
   23  | 57 . . . 4 3 . . .  . 20  1  5  .  .  3  9  4  .  9  |
   24  |  7 . 7 3 . . 4 . 8  1  .  .  .  3  3  .  4  .  .  .  |
   25  | 14 . 5 3 4 . 9 . .  .  .  1  1  .  .  .  .  .  0  .  |
   26  |  . . . 3 . . . 4 8  .  .  .  .  .  .  .  3  .  7  .  .  |
   27  |  . . . . 2 . 1 . 8  .  .  .  3  .  4  5  6  .  .  .  |
   28  | 14 . 4 3 . 3 . . 0  .  .  .  3  .  4  .  .  .  .  .  |
   29  |  . . 6 3 . 3 9 . .  . 67  .  1  .  3  .  .  4  .  .  |
   30  | 10 . 7 . . . 4 . 0  .  .  .  4  .  3  .  .  .  .  .  |;

  c(j)=/  9 1 1 3 4 . 2 . .  4 5   1  2  .  3  3  .  .  0  .  /;
  b(i) = [ 1  50    2 100    3 120    4 130    5 150    6 700
           7 300    8 400    9 200   10 230   11 234   12 230
          13 210   14 800   15 209   16 230   17 789   18 340
          19 357   20 120   21 230   22 345   23 567   24 456
          25 345   26 456   27 567   28 567   29 456   30 200 ];
MODEL restriction; obj_function;
{$Solve this model with XA}
PRINT obj_function;
PRINT : c,X;
END.
```

```
(* PRODUCT.LPL: see also model 'PRODUCT1.LPL' *)
Set
  Periods = (Summer'1' Winter'2');
  ProductionMode = (normal  overtime);
  {three products (P1,P2,P3) with there storing costs, storing capacity
   and resale value per unit}
  Products |Store|StoreCapa|ResaleValue =
         (P1|1|20|2   P2|1|20|2   P3|1|.|1);
  Machines = (M1:M3);

Coef
    MachineTable(Periods,ProductionMode,Products,Machines) =
              /4 7 3   5 6 .   6 6 .
               3 6 2   4 5 .   5 5 .
               5 8 4   6 7 .   7 7 .
               4 7 3   5 6 .   5 6 . /;
    cost(Periods,ProductionMode,Products,Machines) =
              /2 4 1   3 3 .   4 2 .
               3 5 2   4 4 .   5 3 .
               3 5 2   4 4 .   5 3 .
               4 6 3   5 5 .   6 4 ./;
    MachAvail(Periods,ProductionMode,Machines) =
              /100 100 40   80  90 30
               110 110 50   90 100 40/;
    Price(Periods,Products)   = /10 10  9   11 11 10/;
    Demands(Periods,Products) = /25 30 30   30 25 25/;

Var
    QUANTITY(Periods,ProductionMode,Products,Machines | MachineTable);
    STORED(Periods,Products);
    SOLD(Periods,Products);

Model
  (* the profit function: *)
    MAXIMIZE profit:
    sum[Periods,ProductionMode,Products,Machines](Price-Cost)*QUANTITY
    - sum[Products]Store*STORED['Summer',]
    - sum[Products](ResaleValue-Price['Winter',])*STORED['Winter',];
  (* constraints of machine availability: *)
    Mach_Avail(Periods,ProductionMode,Machines):
     sum[products] MachineTable * QUANTITY  <=  MachAvail;
  (* stock balances in the two periods *)
    Stock(Products,Periods):
     sum[ProductionMode,machines]  QUANTITY + STORED[Periods- 1,]
            = STORED + SOLD;
  (* minimum demands to be satisfied *)
    Demand(Periods,Products):   SOLD >= demands;
  (* upper bound on storage *)
    Bound(Products):  STORED['Summer',]  <=  StoreCapa;
{$Solve}
PRINT profit; QUANTITY; STORED; SOLD;
End
```

```
(* NETWORK1.LPL: see also model 'NETWORK.LPL' *)


SET  { definition of the node set }
  nodes = (a b c d e f g);
COEF  flows[nodes,nodes] =
                        |   b   c   d   e   f   g  |
                   {----------------------------------}
                    a    |  12  10  14   .   .   .  |
                    b    |   .   9   .  10   .   .  |
                    c    |   .   .   .  10   8   .  |
                    d    |   .   9   .   .   7   .  |
                    e    |   .   .   .   .   8  30  |
                    f    |   .   .   .   9   .  30  |;

VAR  x(nodes,i=nodes | flows(,i) );
MODEL
  { bounds on the single flows }
    bounds[nodes,i=nodes]:  x(,i) <= flows(,i) ;
  { constraints }
    rel[i=nodes | i>1 AND i<SUM(nodes) 1]:
         sum[nodes] x[,i] = sum[nodes] x[i,];
  { objective function }
    maximize flow: sum[nodes] x[,'g'];
{$Solve}
PRINT flow:10; x:6;
END

(*********************************************************************)
(* VACATION.LPL:  Round trip problem
   Problem: Visit all towns at least once at minimum costs *)

set  i = (GraniteCity'gc',StGeorges'sg',Cutler'cu',Chester'ch',Denver'de');
set  j = (GraniteCity'gc',StGeorges'sg',Cutler'cu',Chester'ch',Denver'de');

coef  { distances between towns in miles }
  Miles(i,j)   =    /   .   1600    90    85  1000
                      1620    .   2000  2300   800
                        95  2005    .     35  1300
                       100  2400    40    .   1800
                      1500   825  1250  2000    .  /;
  UnitCost = 1; { costs per unit/mile }

var  VISITS(i,j | i<>j)  {INTEGER};

model
  {Visit each location at least once}
    V(j): sum(i) VISITS >= 1;
  {Leave all location at most once}
    L(i): sum(j) VISITS <= 1;
  {insure a unique circle tour}
    T(i,j | j>i): VISITS(i,j) + VISITS(j,i) <= 1;
  {Minimizing cost function}
    minimize cost: sum(i,j=i) Miles*UnitCost*VISITS;
{$S} print VISITS:6; cost;
end
```

```
(* AMPL.LPL: a small production model from: FOURER R.[3], A Modeling Language for
Mathematical Programming, 1989, p.5 *)
SET
  p;     { list of products }
  r | init_stock | cost | value;
         { list of raw materials,
         { each with a maximum initial stock, cost and
         { an estimated residual value or a disposal cost (if < 0) }
  t;     { number of storage periods }
  t = ( 1:5 );
  p = (nuts bolts washers);
  r        | init_stock | value | cost  =
  ( iron   |    35.8    | 0.025 | 0.03
    nickel |     7.32   | -0.01 | 0.025 );
COEF
  last = SUM(t) 1; { the last period }
  max_prd;    { maximum unit of production }
  units(r,p); { quantity of raw material needed to produce a product }
  profit(p,t | t<>last);
              { estimated profit or disposal cost (if < 0) of a unit
              { of product in each period except the last one }
  max_prd = 123.7;
  profit =        |  1     2    3    4   |
          bolts   | 1.82  1.9  1.7  2.5  |
          nuts    | 1.73  1.8  1.6  2.2  |
          washers | 1.05  1.1  0.95 1.33 |;
  units =        | nuts washers bolts |
          nickel | 0.21   0.08   0.17 |
          iron   | 0.79   0.92   0.83 |;
VAR
  X(p,t | t<>last);
              { number of units of product manufactured
              { in each period except the last one }
  S(r,t);     { number of units of raw material in storage
              { at the beginning of each period }
MODEL
  maximize total_profit:
     sum[t | t<>last] (sum[p] profit*X - sum[r] cost*S) + sum(r) value*S(,-1);
            { maximize the total over all periods of the estimated profit
            { minus the storage costs plus the value of remaining raw
            { materials after the last period }
  limit[t | t<>last]: sum[p] X <= max_prd;
            { total production in each period is limited }
  start[r]: S(,1) <= init_stock;
            { storage on raw materials in first period is limited }
  balance[r,t | t<>last]: S(,t+1) = S - sum[p] units*X;
            { storage of each raw material in a period must equal
            { storage in the last period plus the used units for production }

{---- the interface to the XA solver ----}
{$Solve}
PRINT total_profit; X; S;
END
```

```
(* DIST.LPL: a model from Fourer R. [3]  *)

PROGRAM dist;
SET
  prod | wt | tc | dt | cpp;
  whse;
  dctr | dsr;
  fact | rmin | rmax | omin | omax | hd | dp;

COEF
  rtmin = 0;   rtmax =  8;
  otmin = 0;   otmax = 96;
  pt | rpc | opc [prod,fact];
  sc | msr [whse,dctr];
  ds [whse,prod];
    { data are in file DIST.DAT: }
    {$I 'DIST.INC' $}
  dem [whse,prod] = dt*ds / (SUM[whse] ds);

  check(dctr): dctr IN whse 'some dctr are not in whse';
  check(fact): fact IN dctr 'some fact are not in dctr';

VAR
  Rprd [prod,fact | rpc];         { regular time production }
  Oprd [prod,fact | opc];         { overtime production }
  Ship [prod,dctr,whse            { shipments }
        | dctr<>whse {AND sc AND (whse in dctr OR SUM[prod] dem > 0)
          AND NOT (msr AND SUM[prod] 1000*dem/cpp < dsr)} ];
  Trans [prod,dctr];              { transshipments }

MODEL
  MINIMIZE cost  : SUM[prod,fact] (rpc*Rprd + opc*Oprd)
                   + SUM[prod,dctr,whse]  sc*wt*Ship
                   + SUM[prod,dctr] tc*Trans;
  rtlim     :  rtmin <= SUM[prod,fact] pt*Rprd/(dp*hd) <= rtmax;
  otlim     :  otmin <= SUM[prod,fact] pt*Oprd <= otmax;
  rlim[fact]:  rmin <= SUM[prod] pt*Rprd/(dp*hd) <= rmax;
  olim[fact]:  omin <= SUM[prod] pt*Oprd <= omax;
  bal[prod,whse]:  SUM[dctr] Ship + Rprd[,whse] + Oprd[,whse]
                   = dem + SUM[i=whse] Ship[,whse,i];
  trde[prod,dctr]: Trans >= SUM[whse] Ship - ( Rprd[,dctr]+Oprd[,dctr] );
{$Solve}
PRINT cost; Rprd; Oprd; Trans;
PRINT(whse,pr,dctr): Ship;
END
```

```
(* BUDGET.LPL: a budget distribution model from Hättenschwiler [6] *)

PROGRAM budget;
SET  accounts | target | ptarget | upper | lower | weight;
VAR
   E | Pd | Nd | Ppd | Npd (accounts);
   Pgd; Ngd;         { global deviation control }

EQUATION
  balance (i=#accounts):
      E = sum (j=$i) E;
  dollar_target (accounts | target):
      E - Pd + Nd = target;
  perc_target (i=accounts | ptarget AND ptarget(!i) ):
      E(!i) - 1/ptarget * E
      - ptarget* (Ppd + Npd )
      - ptarget*ptarget(!i) * (Pgd + Ngd) = 0;
  bnl | bnu (accounts):
      lower <= E <= upper;
  MINIMIZE deviation:
      sum(accounts) weight * (Pd + Nd + Ppd + Npd + 0.001*(Pgd + Ngd));

{ data: a simple family budget }
{ <note: data are not for a feasible solution> }
SET
  accounts            | target | ptarget | upper  | lower  | weight  =

  ( total_expense     |  10000 |   1     | 10500  | 8500   |  1000
    ( household       |   4000 |   0.4   |  4500  | 2500   |   900
      ( bar           |    250 |   0.05  |   300  |  100   |   400
        divers        |    250 |   0.05  |   300  |  100   |   100
        telephone     |    300 |   0.1   |   400  |  100   |   400
        ( tel1        |    200 |   0.7   |   290  |   50   |   100
          tel2        |    100 |   0.3   |   110  |    0   |   200
        )
        wine          |    100 |   0.05  |   300  |    0   |   500
        clothes       |   1900 |   0.3   |  2000  |    0   |   100
        him           |    500 |   0.2   |   800  |    0   |    50
        her           |    500 |   0.2   |   800  |    0   |    50
      )
      dwelling        |   4000 |   0.4   |  4050  | 2950   |   200
      ( rent          |   1000 |   0.20  |  1000  |  500   |   200
        extras        |    200 |   0.05  |   300  |    0   |   300
        electricity   |    200 |   0.05  |   250  |  100   |   400
        oil           |    600 |   0.10  |   700  |  200   |   200
        water         |    200 |   0.05  |   250  |  150   |   100
        investments   |   1000 |   0.25  |  1050  |  750   |   100
        repair        |    800 |   0.20  |   900  |  300   |    50
      )
      rest            |   2000 |   0.2   |  2000  |    0   |   700
    )
  );

MODEL balance; dollar_target; perc_target; bnl; bnu; deviation;
{$S}
PRINT : '                target  expense  difference';
PRINT(accounts): target, E, target-E;
END
```

```
(* SOCCER.LPL: from BYER James, Sunset Software, personal communications *)
PROGRAM Soccer ;    {$R1}
SET
  t = (1:15);       " the list of teams "
  p = (1:180);      " the list of players "
  must_be_in(p,t);  " player p must be in team t "
  reject_from(p,t); " player p is rejected from a team t "
  together_groups;  " groups of players which must be together in a team "
  never_groups;     " groups of players which must never be together"
" skill and age of player p "
COEF  Skill(p) = trunc(rnd(3;8));      Age(p)   = trunc(rnd(10;12));
SET
  must_be_in(p,t)  = [  1 2 ,  2 6 , 3 7 ];
  reject_from(p,t) = [ 10 1 , 20 2 , 1 1 , 1  2 , 1  3 , 1  4 , 1  5 , 1 6 ,
                        1  7 , 1  8 , 1 9 , 1 10 , 1 11 , 1 12 , 1 13 ];
  together_groups
    = ( group1 (   2   3 )   group2 ( 112  76 )
        group3 (  89   9 )   group4 (  34 135 )
        group5 (   1  35  47  81  98  ) );
  never_groups
    = ( group1 (  21 , 22 )   group2 (  55 , 56 )
        group3 (  11 , 35 , 45 , 56 , 67 , 78 , 89 , 90 , 21 ) );

" check that a player p can be only in one team "
CHECK(p): sum(t) must_be_in <= 1   'Illegal must_be_in';
{----------- end data -----------}

VAR  work(p,t)  INTEGER   <=1; " =1 if player p is in team t else 0 "

MODEL  " maximize the assignment "
    MAXIMIZE obj: SUM(p,t) work;
    " player p must be only in one team "
    Bounds(p): SUM(t) work = 1;
    " total skill level of team t must be at least 59 "
    Skill_level(t): SUM(p) work * Skill >= 59;
    " total player count per team t must be 12 "
    Heads(t): SUM(p) work = 12;
    " total team age of team t must be at least 124 "
    Team_age(t): SUM(p) work * age >= 124;
    " player p must be in team t "
    Must(p,t | must_be_in): work = 1;
    " player p is rejected from a team t "
    Reject(p,t | reject_from): work = 0;
    " players which must be in the same team "
    Same(t,i=#together_groups,j=$i | j<>1): work[j-1,t] - work[j,t] = 0;
    " players which must not be in the same team "
    Never(t,i=#never_groups): SUM(j=$i) work[j,t] <= 1;
{$Solve}
MASK m1
 player  Skill Age  in Team
------------------------------
 $$$$$$    ##   ##    $$$$

 group Team-skill Team-age Aver-age Aver-Skill | players
-----------------------------------------------|-----------------------
 $$$$$   #####     #####   ###.###   ###.###   | $$$

ENDMASK;
PRINT  m1(ROW(p) (p , Skill , Age , COL(t|work) t) ,
          ROW(t) (t , SUM(p|work) Skill , SUM(p|work) Age ,
                      (SUM(p|work) Skill)/12
                  , (SUM(p|work) Age)/12 , COL(p|work) p) );
END
```

```
(* LEARN8.LPL: Index-operators and Report Generator example *)
SET
  i=(1:10);
  j=(1:5);
COEF
  a(i) = /2 4 5 2 7 2 4 3 0 2/;
  SUMa=SUM(i) a;
  MAXa=SMAX(i) a;
  MINa=SMIN(i) a;
  PRODa=PROD(i) a;
  EXISTa=EXIST(i) a;
  FORALLa=FORALL(i) a;
  rate(i) = trunc(rnd(400;700))/10000;  { between 4% and 7% }
  discount_factor(i) = 1 / (PROD(j=i | j<i) (1+rate));
  c(i)= PROD(j=i | j<=i) i;  { = i^i (ith power of i) }
  d(i)= PROD(j=i | j<=i) j;  { = i!  (faculty) }

{ --------- report some results -----------}

{ 1. simple PRINT statement }
PRINT : ' *********** SIMPLE PRINT STATEMENT **************';
PRINT b; a:6:1; rate:6:2; SUMa:6;

{ 2. Print expressions without use of masks }
PRINT : ' *********** EXPRESSION PRINT STATEMENT **************';
PRINT   : '              a(i)      rate      discount';
PRINT(i): '  |',a,100*rate,discount_factor:10:2;

{ 3. Print statements with a mask }
MASK m1

     And this is the output of all the calculated values
  ******************************************************

     SUMa  MAXa  MINa    PRODa  EXISTa  FORALLa
     ####    ##    ##  #######   $$$$$   $$$$$

  EXISTa tells, if there is a 'a(i)' different of zero.
  FORALLa tells, if all 'a(i)' are different of zero.

      i     a(i)   rate(i)  discount(i)    c(i)        d(i)
   I-------------------------------------------------------I
   I $$     ##      ##.##%  ####.#####  ###########  ####### I
   I-------------------------------------------------------I

ENDMASK;
PRINT : ' *********** PRINT STATEMENT WITH A MASK **************';
PRINT m1(SUMa,MAXa,MINa,PRODa,if(EXISTa;'TRUE';'FALSE'),
         If(FORALLa;'TRUE';'FALSE'),
         ROW(i)(i,a,100*rate,discount_factor,c,d));
END
```

Index of all LPL models on disk

ALLOY.LPL: a simple illustrative blending problem.
AMPL.LPL: A small production model to compare AMPL and LPL model language
BUDGET.LPL: A budget distribution model showing hierachical indexing
CLASS.LPL: an integer program to generate a set of assignments
CROP.LPL: a small production model
DIET.LPL: a diet model to compare MPL with LPL.
DIET1.LPL: the general DIET Problem: *)
DIST.LPL: A distribution model - compares AMPL with LPL
EGYPT.LPL: A static model of fertilizer production
EXPORT.LPL: A import/export model
FIT.LPL: The Order-Truck Fit Model
GENERAL.LPL: the general LP model with i rows and j columns,
LEARN1.LPL: Learn simple and indexed expressions *)
LEARN2.LPL: Learn the math functions *)
LEARN3.LPL: Learn IN operator and index replacement
LEARN4.LPL Tutor example
LEARN5.LPL: Learn index replacement options
LEARN6.LPL: Learn data formats and reassignment
LEARN7.LPL: Learn and use expression-lists *)
LEARN8.LPL: Learn IndexOperators and Report Generator
LEARN9.LPL: Report generator with print and mask
LEARNA.LPL: Learn index tree operations
LEARNB.LPL: Learn nomenclature - internal names
LEARNC.LPL: Learn the MPSX output for MIP models
MAGIC.LPL: scheduling of generators: compare LPL with MAGIC (and GAMS)
MEDIA.LPL: maximize the number of effective exposures in media vehicules.
MEXICAN.LPL: the Mexican Steel Industry Example.
NETWORK.LPL: a simple network problem:
NETWORK1.LPL: a simple network model (like network.lpl)
NETWORK2.LPL: a simple network model (like network.lpl)
OMP.LPL: a production planning model - compares OMP with LPL
PAM.LPL: a transport model fragment - compares PAM with LPL
PAPER.LPL: a simple paper trimming problem:
PLANNING.LPL: production planning model - compares MPL with LPL
PLANTS.LPL: a simple distribution problem:
PORC5.LPL: a model with circular time lag operators
PORTFOL.LPL: Fixed income portfolio selection, a stochastic model.
PROD.LPL: a multiperiod production model - compares AMPL with LPL
PRODPLAN.LPL: production/planning model - compares LPL with MPL *)
PRODUCT.LPL: a production model - see also model product1.lpl *)
PRODUCT1.LPL: a production model - compares UIMP with LPL
REL.LPL: a model fragment which constructs a complex relation R
ROBOT1.LPL Tutor example
ROBOT2.LPL Tutor example
ROBOT3.LPL Tutor example
ROBOT4.LPL Tutor example
ROBOT5.LPL Tutor example
ROBOT6.LPL Tutor example
ROBOT7.LPL Tutor example
ROBOT8.LPL Tutor example
ROBOT9.LPL Tutor example
ROBOTA.LPL Tutor example
ROBOTB.LPL Tutor example
ROBOTC.LPL Tutor example
SCHEDULE.LPL: a production scheduling problem for several periods
SHIP0.LPL: a simple transportation problem with 60 variables
SHIP2.LPL: a transportation problem with 2000 variables *)
SHIP3.LPL: a transportation problem with 3600 variables *)
SIMPLE.LPL: a simple LP-Model with three variables
SOCCER.LPL: an assignment problem of 180 player to 15 teams
SOCCER1.LPL: an assignment problem of 180 player to 15 teams
TANGLE.LPL: A Tanglewood Chair Manufacturing Co. Example.
TRAIN.LPL: a model of railroad passenger car allocation
TRANS.LPL: a small transportation model *)
TRANS1.LPL: transportation model *)
TRAVEL.LPL: travel optimization - compares MGG with LPL
VACATION.LPL: a small round trip problem
XPRESS.LPL: multi-period blending problem - compares XPRESS with LPL

REFERENCES

[1] BROOKE A., KENDRICK D., and MEERAUS A., GAMS, A User's Guide, The Scientific Press, 1988. (Demo Version for PC)

[2] CUNNINGHAM K., and SCHRAGE L, The LINGO Modeling Language, University of Chicago, Preliminary, 27 February 89.

[3] FOURER R., GAY D.M.*, and KERNIGHAN B.W.*, AMPL: A Mathematical Programming Language, Technical Report 87-03, Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, Illinois 60209-3119, revised June 1989. (* AT&T Bell Laboratories Murray Hill, New Jersey 07974)

[4] GEOFFRION A., SML: A Model Definition Language for Structured Modeling, Western Management Science Institute, University of California, Los Angeles, Working Paper No.#360, revised Nov. 1989.

[5] GREENBERG H.J., An Overview of the Development of An Intelligent Mathematical Programming System (IMPS), University of Colorado, Denver, Working Paper, Dec. 1989.

[6] HÄTTENSCHWILER P., Personal Communication, May 23, 1989 and June 1, 1989, from University of Fribourg, Institute of Automation and Operations Research, Switzerland.

[7] HÜRLIMANN T., Kohlas J., A Structured Language for Linear Programming Modeling, Working Paper No. 122, November 1986, Institute for Automation and Operations Research, University Fribourg, Switzerland.

[8] HÜRLIMANN T., LPL: A Structured Language For Modeling Linear Programs, Dissertation, Peter Lang Verlag, Bern, 1987.

[9] HÜRLIMANN T., Reference Manual for LPL Version 3.1, Working Paper No. 372, Western Management Science Institute, University of California, Los Angeles, October 1989.

[10] HÜRLIMANN T., Update of LPL from Version 3.1 to Version 3.5, Working Paper, Institute for Automation and Operations Research, University Fribourg, Switzerland, May 1990.

[11] LANGENEGGER P., Formale Modellierung der Konsumlenkung für die schweizerische Lebensmittelrationierung mit LPL 3.5, Diplomarbeit für das Diplom in Wirtschaftinformatik bei Prof. Dr. J. Kohlas und Dr. P. Hättenschwiler, Universität Freiburg,.Sommersemester 1990.

[12] SCHRAGE L., Linear Programming Models with LINDO, The Scientific Press, Palo Alto, Calif., 1981.

[13] SUNSET SOFTWARE, XA, A Professional Linear and Mixed 0/1 Integer Programming System, 1613 Chelsea Road, Suite 153, San Marino, California 91108, phone: 818-441-1565, fax: 818-441-1567.

INDEX

UPDATE NOTES FROM LPL 3.1 TO 3.5

CORRECTIONS AND MODIFICATIONS FROM LPL 3.1 TO 3.5

- Most important modification: Every expression is evaluated at parse time now not at the end of the parsing. This means that every set, coefficient etc. must be defined (not only declared) before using in an expression.
- The parameters of MAX(), MIN() and IF() must be separated by a semicolon now. (exp.: MAX(3;4) instead of MAX(3,4).
- The compiler directives {$C and {$R are replaced be a new one {$N. (explained below).
- In Manual 3.1 page 1: reading or writing directly from and to dBASEIII is not available. (This was possible in older versions of LPL. This will be reinstalled).
- page 22 line 13: erase 'Hence, we declared .... leaves (&i)'.
- page 23 reserved words: PRINT is not documented, it is now explained below.
- page 23 reserved word FILETABLE has changed to FILE used for future versions.
- The files LPLVAR.EXE, LP83BAT.BAT, and MIP83BAT.BAT are no more needed.
- in LPLMPS: reals in .MPS file are limited to 10 positions (not 11!).
- in LPL: 'i IN j' produced always TRUE, this is fixed now.
- CEIL(x) produced an erroneous value if x was negative.
- Error described in model LEARN4.LPL

NEW FEATURES

Six important enhancements have taken place in version 3.5 of LPL:

1. A powerful Report Generator has been added to the LPL language through the MASK and the PRINT statement.
2. The expression syntax has been enhanced: New functions and index-operators have been added, and the concept of automatically indexing has been introduced.
3. A CHECK statement has been added which allows to test different parts of a model.
4. The compiling step can be better controlled through several compiler directives.
5. A DELETE statement has been added, which allows to delete old items from the heap no more needed in the rest of the model.

6. Transparent and open solver interface to any LP and MIP solver.

NEW FUNCTIONS:

Some mathematical functions have been added and can be used in any expression:
FLOOR(x): greatest integer smaller than x
TRUNC(x): truncated x to an integer
SIN(x): sinus of x
COS(x): cosines of x
LOG(x): natural logarithm of x, error if x<=0
SQRT(x): square root of x, error if x<0
RND(x;y) : returns a uniform random value in the interval [x,y]
RNDN(x;y) : returns a normal distributed value with mean x and standard deviation y.
PROD is a new reserved word. It is a index-operator. Much like SUM, PROD produces the product over a set.
Example:

```
SET i=(1:5);
COEF a(i) = / 2 3 4 2 5 /;
COEF b = PROD(i) a;   { = a(1)*a(2)*a(3)*a(4)*a(5) }
COEF c = PROD(i) i;  { 'a' is 5! (faculty of 5) }
```

For an example see model LEARN8.LPL.
COL and ROW are also two new index-operators. They are explained in the Section PRINT statement below.

AUTOMATICAL INDEXING

',' (the comma) is now also a binary operator in an expression. Expressions concatenated with ',' are called expression-list. Any expression-list is also an expression.
Example:

```
a,b,a+b  {is an expression with three values}
```

An expression-list returns normally a list of values. A list of values may also be assigned to an identifier as any other expression. This identifier will be automatically indexed.
Example:

```
COEF a = 3,4,5,6;
```

'a' is now automatically indexed. It contains a list of four values. Automatically indexed identifier are treated as other indexed identifier, but there is one important difference:
the automatically index is not known to the user. In general the user needs not care about automatically indices. Suppose the user wants to assign the sum of all values of

'a' (above) to a coefficient 'b', than he should know the index, since he must sum over the automatically index. In this case LPL uses a question-mark ('?') as index:
COEF b = SUM(?) a;
An automatically indexed identifier can be used in any expression. This syntax is used in the PRINT statement. The comma may also be replaced by '|' (the bar).

Here is a summary of the expression syntax in version 3.5:

```
Expression ::= [ IndexOperator IndexList ] Factor { RelOp Factor }
unary ::= '+' | '-' | NOT
IndexOperator ::= SUM | FORALL | EXIST | SMIN | SMAX | PROD | COL | ROW
Factor ::= [ unary ] number
        | [ unary ] VarOrCoefIdentifier [ AIndexList ]
        | [ unary ] function
        | [ unary ] '(' Expression ')'
        | SetIdentifier
        | QuotedString
RelOp ::= ',' | '|' | OR | AND | IN | '=' | '<>' | '<' | '>' | '<=' | '>=' | '+'
       | '-' | '*' | '/' | '^'
function ::= IF '(' Expression ';' Expression [ ';' Expression ] ')'
          | MathFunc '(' Expression ')'
          | MathFunc2 '(' Expression ';' Expression ')'
MathFunc ::= ABS | CEIL | FLOOR | TRUNC | SIN | COS | LOG | SQRT
MathFunc2 ::= MAX | MIN | RND | RNDN
```

Note the modifications in an expression:
- New index-operators PROD, COL and ROW. Note that COL and ROW are only allowed in the PRINT statement.
- New binary operators: ',' and '|' with highest precedence.
- Semicolon in IF, MAX, MIN, RND and RNDN functions (instead of ';').
- New functions: FLOOR, TRUNC, SIN, COS, LOG, SQRT, RND and RNDN.
- Factor may also be a quoted string; note, however that this syntax is only allowed in the PRINT statement.

CHECK STATEMENT

A check statement can be added at any place where another statement is allowed. It allows the user to check a condition, and if it is false (evaluates to zero¡), the compilation stops with a user error message.
Syntax:

```
CheckStatement ::= CHECK [ IndexList ] : Expression [ QuotedString ] ';'
```

Examples:
check: a<>0 'A must be different of zero' ;
check(i,j): a(i,j) > 1 'all a(i,j) must be greater than one' ;
check(i): SUM(i) 1 > 20 'set I must have at least 20 members' ;

The first CHECK checks if 'a' is different of zero. If this is not the case, then the error message "A MUST BE DIFFERENT OF ZERO" is displayed and the compilation stops.
The second CHECK tests all a(i,j), and if one is less or equal 1 the compiler stops and displays the message. The second check could also be written as:
check: FORALL(i,j) (a(i,j) > 1) 'all a(i,j) must be greater than one' ;
The third check stops the compiler if set 'i' has less than 20 members.
The check statement is very powerful and gives the user a tool to test different model parts, especially data consistency.
For an example of the check statement see the model DIST.LPL. The two check statement make sure that all distribution centers are also warehouses and all factories are also distribution centers.
Note, however, that the QuotedString is actually limited to 25 characters. Note also, that the check statement is executed at parse time and not at the end of the parse.

DELETE STATEMENT:

A DELETE statement can be added at any place where another statement is allowed. It begins with the reserved word DELETE followed by a identifier or a '*' character and ends with a semicolon.
The syntax:
DELETE identifier | '*' ;
This statement allows to delete coefficients, variables or restrictions of the model which are no more needed in the subsequent part. The Delete statement free the heap in order to allow more tables to be entered in a model.
Example:
COEF a(i,j,k); b(i,j); c(i);
...{ define now a, b and c and use them }...
DELETE a; b; { delete now two tables }
The identifier may be replaced by a '*'. This means that all previous restrictions declared within a Equation or a Model statement are deleted.

MASK STATEMENT:

A Mask Statement can be added at any place where another statement is allowed. It begins with the reserved word MASK followed by a mask-identifier. The content of

the mask begins at the beginning of the next line and extends until the reserved word ENDMASK. An semicolon ends the Mask Statement.
The syntax:
MASK MaskName
< Content of the mask >
ENDMASK ;
The Mask Statement allows the user to define the lay-out or the format of a specific Print Statement (explained below). The content of the mask may contain any characters (tabs and returns included) and may extend over several lines. Only the two characters '$' and '#' have a special meaning. A consecutive stream of $'s in the mask is called an *alphanumeric place-holder*, and a consecutive stream of #'s together with an optional dot in the middle is called a *numeric place-holder*. A subsequent PRINT statement prints the mask content exactly as defined by the MASK statement, where the place-holders are filled with alphanumeric and numeric data.
Example:
MASK m1
Here begins the mask. Now comes a alphanumeric place-holder: $$$$$$$$EndLine

This is the third line of the mask. (line two is empty).
######.##### $$$$$$$######## $$$$$####.###
Line four contains five place-holder (two alphanumeric).
ENDMASK;

An complete example with the Mask and Print Statement can be studied in the model fragments LEARN8.LPL and LEARN9.LPL.

PRINT STATEMENT

A Print Statement can be added at any place where another statement is allowed. It begins with the reserved word PRINT. The Print Statement is a very powerful table generator. Three different kind of prints can be used with LPL: 1. print simple tables, 2. print expressions and 3. print expressions together with masks.
Syntax:
PRINT Identifier [Format] ['(' Expression ')'] ';'
| PRINT [IndexList] ':' Expression [Format] ';'
Format ::= ':' Integer [':' Integer]

1. The simplest report can be made by using the PRINT keyword followed by a identifier. This identifier may represent a data table, variables or a restriction.
Example:

PRINT a; b;

A format specification may be added to the identifier which determines the width of the data. The first integer after the colon limits the total width and the second the number of decimal places.

Example:

PRINT a:12:7; b:3;

2. If the user may print an indexed expression, the second kind of print may be used. The PRINT keyword is followed by a optional IndexList, by a colon and an expression.

Example:

PRINT : a*b; { if a and b are not indexed }

PRINT(i,j) : a(j,i); { note the order of the indices }

PRINT(i) : SUM(j) a; {an expression }

Again a format specifier may be added to the expression as in the first example.

3. If the format and the layout of the report is more complex, then the user has the possibility to use the print together with a mask. The reserved word PRINT is followed by a mask-identifier. This identifier is followed by a expression surrounded by parentheses. The Print Statement ends with a semicolon. The identifier must be defined as a mask before used in a Print Statement. The mask-name defines the lay-out of the print. The expression (which is, in general, an expression-list) returns the values which are inserted into the mask at the place-holders. There must be at least as many place-holders in the mask as expressions in the Print Statement. The return values of the expressions are inserted into the mask from left to right and top to bottom.

Expressions in the Print Statement have a slightly different behavior as expressions in other parts of the model:

- The index-operators COL and ROW are only allowed in the Print Statement.
- Quoted strings are allowed in the Print Statement only. In this case the expression returns the string itself instead of a numeric value.
- If a set-identifier is used,the expression returns the member-name as a string if the corresponding place-holder is alphanumeric (and not its position as a numeric value).

Example:

PRINT m1('First' , 345.67 , 'Second' , 2315.78,'Third' , 23);

together with the mask m1 (defined in the last section) produces the following output:

Here begins the mask. Now comes a alphanumeric place-holder: First EndLine

This is the third line of the mask. (line two is empty).

345.67 Second 2315Third 23.

Line four contains five place-holder (two alphanumeric).

Alphanumeric place-holder are flushed left, whereas numeric are flushed right. The insertion takes place from left to right within the Print expression. They are inserted from left to right and from top to down within the mask. 'First' is inserted at the first alphanumeric place-holder on the first line, '345.67' is inserted at the next place-holder at the third line, etc. The real power of the PRINT statement comes from the two new index-operators COL and ROW. The COL index-operator extends a place-holder horizontally, whereas the ROW index-operator extends a place-holder vertically.

Example:

```
SET i=(P1:P4);
MASK m2
  begin x$$$$$end
ENDMASK;
PRINT m2(COL(i) i);
PRINT m2(ROW(i) i);
```

The first print produces the following output:

```
  begin xP1   xP2   xP3   xP4   end
```

Note that the last character just before the place-holder is repeated too. This has the important function of a separator.

The second print produces the following output:

```
  begin xP1   end
  begin xP2   end
  begin xP3   end
  begin xP4   end
```

Note that the whole line of the place-holder within the mask is also repeated.

An complete example with the MASK and PRINT statements can be studied through the LEARN8.LPL and LEARN9.LPL model fragment.

EQUATION STATEMENT

The EQUATION statement is exactly the same as the MODEL statement, except that the production to the output file .RES is postponed. A model restriction may be defined through a EQUATION statement, but nothing is written to the .RES file. Only a subsequent MODEL statement produces the restriction.

Example:

```
EQUATION
  r1 : x+y+z <= a;
  r2 : x-y-z > b;
```

{ nothing has been written now }
.....
MODEL r1; r2; { now the two restriction are printed to the .RES file }

NEW COMPILER DIRECTIVES

LPL 3.1 knew three compiler directives: {$I for include files, {$C and {$R both for intern names of variables and restrictions. {$C and {$R are replaced by a single one: {$N.
The following compiler directives are now integrated in LPL 3.5:
{$I <filename>} File include (same as in LPL 3.1)
{$C} Case sensitivity on
{$R<digit>} Random number initializer
{$Bxy} Binding strictly on and off
{$M<int1><int2><int3><int4>} Memory requirements
{$X <prog><param>} Execute a child process
{$P <SIP> define solver interface parameters (SIP)
{$S [<SIP>]} Call the LP Solver and returns the variable-values
{$N<arg1><arg2> Nomenclature

Case sensitivity {$C} :
Case sensitivity means that lower and upper case letters are distinct by the compiler. The LPL compiler is not case-sensitive by default. Identifiers can be made case-sensitive by a new compiler-directive: {$C.
Example:
{$Case sensitive}
If case sensitivity is on, it is important that all reserved word are written entirely in upper-case.
This enhancement has different advantages
- Reserved word can also be used as identifiers, if they are written with at least one lower-case letter ('Var' is no more a reserved word).
- The identifiers in the outputs (actually the .NOM file) are printed as entered in the model file.
Note that this compiler directive must be added within the model at the very beginning (or in the LPL.CFG file) for obvious reasons.

Random number initializer {$R<digit>} :
The built-in random generator can be initialize with the new compiler directive {$R}. Any comment is allowed within this directive as long as the directive begins with R.
Examples:

{$R initialize the random generator}
{$Randomize}
The directive may be extended by a digit which must follow the 'R' immediately.
Example:
{$R1}
This imposes the random seed which reproduce a sequence of the same random numbers every times. If no digit follows the 'R' the random seed is arbitrarily chosen by the system.
To study an example see model SOCCER.LPL and LEARN2.LPL.

Binding strictly on and off {$Bxy} :
LPL has a relaxed syntax for index binding and IndexList dropping by default. A simple example is: 'SUM(i) a(j)'. If 'i' and 'j' have some common members, then the summation takes place over the common members. This is called *relaxed binding*. Furthermore, the IndexList '(j)' after 'a' may even be dropped, if 'a' has been declared as 'a(j)'. Therefore, 'SUM(i) a' is exactly the same as 'SUM(i) a(j)'. This relaxed syntax has a price: relaxed error checking. If the user wants to impose the compiler to check some additional errors, the new directive {$Bxy} may be added at different places within the model, where 'x' and 'y' are any two characters. If 'x' is '+' then *strict binding* is enforced which means, that the bound index must be the same as the binding index. If 'y' is '+' then IndexList dropping is no allowed any other two characters mean a relaxed syntax.
Examples:
{$B++} strict binding and dropping is no allowed
{$B+-} strict binding and dropping is allowed
{$B-+} no strict binding and dropping is no allowed
Therefore, '{B++} SUM(i) a' produces an error. It must be replaced by '{B++} SUM(dummy=i) a(dummy)'. See model LEARN6.LPL for an example.

Memory requirements {$M<imaxs><imaxa>}
The heap memory manager of LPL reserves some space for different data. The user may change their values with the {$M directive. Four different numbers may be changed:
- maxs: the number of sets and members
- maxa: the number of numeric data
Example:
{$M 500 5000 } {this is the default}
Note: this directive is only available on the LPL C Version.

Execute a child process {$X <prog><param>} :
Any program may be executed from within an LPL model with the {$X directive. The 'X' must be followed be two strings indicating the program-name and its parameters.
Example:
{$X lplmps.exe model1 }
This instruction executes the LPLMPS with the parameter 'model1'. Note that the extension of the program-name must be added. (See model LEARN5.LPL).

Solver Interface Parameters{$P <SIP>}.
This directive allows the user to define the solver interface parameters. (see Section solver interface).

Solve a linear model {$S [<SIP>] }.
This directive is the interface to the solver. It calls LPLMPS and produces the MPS input file. Then it calls the LP/MIP solver to solve the model, and, finally it reads the values of the variables back into the LPL (if the solver has found a solution), so that this data may be printed by the report generator.
Examples:
{$S}
{$Solve this model now}
Note, that the values of the variables can only be read back, if no {$N directive is used within the model. So, delete any {$N directive, before using the {$S directive.
Like the {$P directive, the user may add the solver interface parameters.

Nomenclature {$N<arg1><arg2> :
Most solver accept only a limit number of characters for row (restriction) and column-names (variables). MPSX, for example, limits this number to 8 character. Therefore, a translation from the LPL extern names to these intern names is needed. Furthermore, the translation must produces unique names. LPL produces unique row and column-names by default. They have the format:
<integer containing three digits><integer>
Example: '10056'.
The first integer is a unique number - beginning with '100' - for each variable and restriction identifier declared, the second is a integer indicating the position in the Cartesian Product of its IndexList if the variable or restriction is indexed.
These names are not very expressive. The user, however, may control the production of these names by

1) the mean of the {$N compiler directive, which replaces both compiler directives '{$C' and '{$R' in version 3.1 and former. (Chapter 8.1. in the Reference Manual version 3.1. is no longer valid). They are replaced by a simpler syntax, or by
2) short-names which is a quoted string following the variable or restriction identifier in the declaration section.

The compiler directive {$N replaces both directive {$R and {$C from version 3.1. The compiler directive {$N determines how the *short-names* (explained below) are produced.
The syntax of '{$N' is:
{$N<arg1><arg2> (any comment may follow here) }
where <arg1> is the character 'n' or a digit between -9 and 9 and <arg2> is 'n', '?', '??' , '???' or a digit between -9 and 9.
The compiler directive '{$N' may be placed several times in a model file, each overrides the former directive. Each set, variable or restriction can have its own directive. An example can be studied in the TEST7.LPL model fragment.
Examples:
{$Nnn} <arg1> is 'n',<arg2> is 'n' : this is the default directive.
{$Nn1 <arg1> is 'n' and <arg2> is '1'}
{$N3??} <arg1> is '3' and <arg2> is '??'}
{$Nn-3} <arg1> is 'n' and <arg2> is '-3'}
{$N-1?} <arg1> is '-1' and <arg2> is '?'}
{$N2} omits <arg2>, which does not change.
{$N any text} this directive will be ignored
<arg1> is
- 'n', which means: produce a unique integer for each variable's and restriction's short-name. This is the default.
- a digit i, which means: produce a short-name for each variable and restriction composed of the first i characters of the variable's or restriction's name. If the digit is negative, the last i characters are taken instead. Lower case letters are translated in uppercase letters.
<arg2> is
- 'n', which means: add an unique integer to the short-name of a variable or restriction, if they are indexed.
- '?', which means: produce short-names for each member of a set. They are produce as: '1' .. '9', 'A' .. 'Z' .. (ASCII-characters with higher ordinal position) in this order. Add them to the variable's or restriction's short-name.
- '??', which means: produce short-names for each member of a set. They are produce

as: '01', '02' .. '99' , 'A0' .. 'Z9' .. in this order. Add them to the variable's or restriction's short-name.
- '???', which means: produce short-names for each member of a set. They are produce as: '001', '002', .. in this order. Add them to the variable's or restriction's short-name.
- a digit i, which means: produce short-names for each member of a set composed of the first i characters of the member's name. If the digit is negative, the last i characters are taken instead. Lower case letters are translated in uppercase letters (unless {$C} is on). Add them to the variable's or restriction's short-name.

Every variable and restriction has a unique short-name which defines how the intern name are produced. The short-name - automatically produced by the {$N directive can be overwritten by the user, if he places a single quoted string in the declaration.
Two character used within this string have special meaning:
'a' followed by a digit means: add an alias of a member.
'i' followed by three digit means: extract some characters from a member and add this to the name.
Example:
SET i = (1:10); j = (spring summer autumn winter);
VAR X'Xa1Ti213X'(i,j);
means: X has the following names: take an 'X' and add the alias of the first index ('01', '02', '03', or '04') add a 'T', then add the third character of the member of the second index and finally add a 'X'. An example of a produced name is: 'X01TSPRX'.
SET i=(1:10); VAR AMOUNT'Xn'(i);
The short-name of variable AMOUNT is 'X'.
{$N2} VAR AMOUNT(i);
The short-name of variable AMOUNT is now 'AMn'.
{$N-3} VAR AMOUNT(i);
The short-name of variable AMOUNT is now 'UNTn'.
Each member of an index has also a short-name (formerly called alias-name). The default short-names of members are '01', '02' .. in the order the members are entered in the set
Example:
SET i = (spring summer autumn winter);
The short-names of the four members are '01', '02', '03', and '04'.

The default short-names takes place if the user do not specify any rules. The short-names of members may be overwritten by the user by two manner:
- by adding a short-name after the member in single quotes

- by modifying <arg2> in the {$N directive.

Examples:

SET i = (spring summer autumn'0C' winter);

The short-names now are '01', '02', '0C', and '04'.

{$Nn?} SET i = (spring summer autumn winter);

The short-names of the members now are: '1', '2', '3', and '4'.

{$Nn2} SET i = (spring summer autumn winter);

The short-names of the members are now: 'SP', 'SU', 'AU', and 'WI'.

{$Nn-3} SET i = (spring summer autumn winter);

The short-names of the members are now: 'ING', 'MER', 'UMN', and 'TER'.

All member short-names are limited to 3 characters at most and all variables and restriction short-names are restricted to 15 characters.

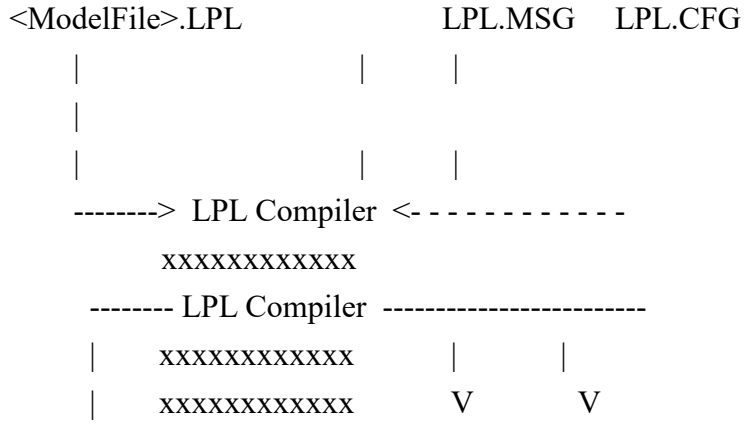
The solver interface

The LPL compiler reads and compiles a ModelFile 'ModelFile>.LPL' written in LPL syntax. If a compile error occurs during the compilation, the error message file LPL.MSG is read - if present - and the error is printed to the screen and the compilation is aborted. Furthermore, a batch file ED.BAT is produced, which may be used to start the editor LPLEDIT.COM allowing the user to correct the error.

Before compiling <ModelFile>.LPL, the LPL compiler compiles the LPL.CFG file if present. LPL.CFG must also be written in LPL syntax.

LPL produces the file '<ModelFile>.RES', a internal representation of the model, as well as the file 'TEMP.CLP' which contains the solver call parameters.

If the Compiler directive {$S} is present within the <ModelFile>.LPL, the LPL compiler calls automatically the LPLMPS process which produces the MPS standard input file 'TEMP.MPS' for a LP or MIP solver. Then the solver is called, which writes the solution to the 'TEMP.SOL' file. This file is read by the LPL compiler when the solver has finished his work. The the report file <ModelFile>.NOM is written. The Figure 1 gives a overview of all these connections.



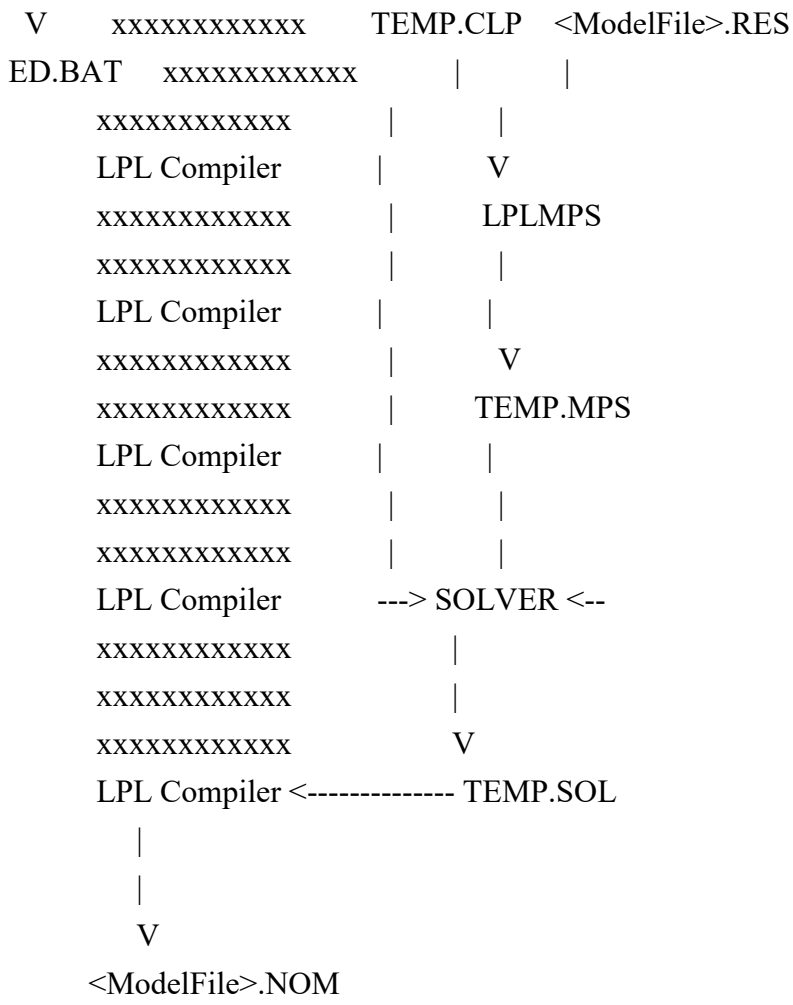


FIGURE 1

LPL has a flexible and transparent solver interface. The communication between LPL and the solver is determinate by 12 interface parameters (explained below). By default, LPL calls the XA solver from SUNSET without any intervention from the user. The user must only place a {$S} compiler directive in the LPL model and the LPL compiler does the whole work: producing the MPS file, calling the solver with the right parameters, and reading the solution file. Then it continues the compilation. This communication is 'hardwired' within the LPL compiler.

But the user may call the solver in a different way, or he or she has a different solver. In this case the user has three possibilities to change the solver interface:
1. include the interface parameters within the {$S directive
2. include the interface parameters within the {$P directive
3. 'hardwire' the interface parameters within the LPL.EXE code.
The first solution is useful, if the user wants occasionally change the interface

parameters (see modelfile MAGIC.LPL). The second solution may be adopted, if the the user wants to include the parameters within the LPL.CFG file. In this case every run overwrites the hardwired interface parameters, since the LPL.CFG is compiled before every model file. The third solution is useful, if the user wants to change the parameters permanently. The LPL.EXE must be changed at the position where the text 'LPLMPS.EXE....' begins. This space (255bytes) is reserved for the interface parameters. The user should add a comma and a ASCII character 0 at the end of the string. The byte just before 'LPLMPS...' should be changed to the length of the string.

The solver interface parameters:
LPL needs 12 parameters in the following order to communicate with the solver
the default value is added in parenthesis:

1. program name which produces the MPS file (LPLMPS.EXE)
2. the parameters of that program (%1)
3. program name of the solver (XA.EXE)
4. the parameters of that program (TEMP.CLP)
5. the name of the solver parameter file (TEMP.CLP)
6. the solver parameter file contents (temp.mps\nACTIVITY temp.sol\nOUTPUT temp RHS ..rhs BOUNDS Bounds LISTINPUT NO\nOBJECTIVE %2 %3)
7. the string which will be added to the solver parameter file, if the model has to be maximized (MAXIMIZE YES)
8. the string which will be added to the solver parameter file, if the model has to be minimized ( )
9. the file name to which the solver has to read the solution (TEMP.SOL)
10. An integer indicating on which physical position on a line in the solution file the first character of the variable-name is found (2)
11. An integer indicating on which physical position on a line in the solution file the first digit of the value is found (12)
12. An integer indicating the length of the numerical value of the variable in the solution file (11)

Any parameters may contain the following strings:
'\n' : which will be translated to carriage-return and new-line.
'%1' : which is replaced by the ModelName
'%2' : which is replaced by the objective function name
'%3' : which is replaced by the seventh or eighth parameters above depending whether the model is to be maximized or minimized.
Any other characters or strings are taken literally.

The parameters 9 to 12 are used to read the solution back to the LPL system. They suppose that the solver write his solution to a file (by default to TEMP.SOL) which contains a variable-name by line with its value. Any line which does not contain a variable-name is ignored.
Example: The following is the XA solver output file TEMP.SOL for the SIMPLE.LPL model (last rows are cut):

```
"U N B O U N D E D   V A R I A B L E 1021","temp    ","Thu Jul 19 12:23:40
    2,     3,  260.00000,      3
"1001    ",  254.00000,    1.00000,    0.00000,"IN   ","(NB)    ",   -0.000
"1011    ",    6.00000,    1.00000,    0.00000,"IN   ","(NB)    ",   -0.000
"1021    ",    0.00000,    1.00000, 1981.00000,"LOWER","(NB)    ",   -0.000
"1031    ",   -4.00000,   -4.00000,    1.00000,"LOWER","EQ","1001    ", -25
"1041    ",    6.00000,    6.00000,   44.00000,"LOWER","GE","1001    ",
```

The lines 1, 2, 6, and 7 are ignored, since the position 2-9 are not variable-names. LPL reads only the bold values of the remaining lines.

The solver interface parameters must be entered by the directives {$S or {$P in the order listed above and separated by commas. The whole string must by in quotes. The string may be broken in several lines. In this cases every part on the line must be in quotes. Any text outside the quotes is treated as an ordinary comment.
Example:

```
{$P this is the default parameter list :
   'LPLMPS.EXE,%1,XA.EXE,temp.clp,temp.clp,'
   'temp.mps\nACTIVITY temp.sol\nOUTPUT temp '
   'RHS ..rhs BOUNDS Bounds LISTINPUT NO\nOBJECTIVE %2 %3,'
   'MAXIMIZE YES, ,temp.sol,2,12,11'
}
```

For the CPLEX solver these parameters might be:

```
{$P 'LPLMPS.EXE,%1,'
   'CPLEX3.EXE,<temp.clp,'
   'temp.clp,'
   'r temp.mps\n s d 0\n s l temp.sol y\n %3\n opt,'
```

```
   'c s 0 ma, ,'
   'temp.sol,1,25,19'
}
```

For the HYPER-LINDO solver they might be:

```
{$P 'LPLMPS.EXE,%1,'
   'LINDO.EXE,<temp.clp,'
   'temp.clp,'
   'BATCH\n DIVERT temp.sol\n RMPS temp.mps\n %3\n GO\n NO\n QUIT,'
   'MAX,MIN,'
   'temp.sol,4,13,12'
}
```

New comments

Any text between double quotes (") is now considered as a comment. LPL has now three different kind of comments:

(* .....*), {.....} ,and "....."

All three can be nested within each other.

Syntax-Update:

```
Model ::= [ ModelHeading ] Statement { Statement } END
Statement ::= SetStatement | CoefStatement | VarStatement | ModelStatement
            | EquStatement | CheckStatement | PrintStatement | MaskStatement
            | DeleteStatement
ModelHeading   ::= PROGRAM Identifier ';'
SetStatement   ::= SET { Set ';' }
CoefStatement  ::= COEF { CoefOrVarDeclaration ';' }
VarStatement   ::= VAR { CoefOrVarDeclaration ';' }
ModelStatement ::= MODEL { Restriction ';' }
EquStatement   ::= EQUATION { Restriction ';' }
CheckStatement ::= CHECK [ IndexList ] : Expression [ QuotedString ] ';'
PrintStatement ::= PRINT Identifier [ Format ] [ '(' Expression ')' ] ';'
                 | PRINT [ IndexList ] ':' Expression [ Format ] ';'
MaskStatement  ::= MASK MaskName <MaskContent> ENDMASK ';'
DeleteStatement ::= DELETE identifier | '*' ;

Comment ::= '{' AnyString | CompilerDirective '}'
          | '(*' AnyString '*)'
```

| "" AnyString ""

CompilerDirective ::= '{$I'<filename>}
| '{$C}'
| '{$R'[<digit>] }
| '{$B'<two characters> }
| '{$M'<two integer>}
| '{$X'<two strings>}
| '{$P <SIP>'}
| '{$S [<SIP>]'}
| '{$N'<arg1><arg2>}

Nested Includes

Nested Include files are now possible. This means, that in an included file there may be further calls to included files through the '{$I' compiler directive. The filename may be entered with or without single quotes. The nesting depth is limited, however, to 5 levels.

Example:

(in main file) .... {$I include.dat } ...

(in include.dat file) .... {$I 'include2.dat' }

The File LPL.CFG

If the file LPL.CFG is present in the current directory, then this file is compiled *before* the model file. LPL.CFG is treated as a include file at the very top of each model file. The use of this file is, that the user can predefine different compiler directives as '{$P', '{$N' or '{$M' and place these directives in the LPL.CFG file. In this case every model will be processed with the new directive. Of course, this directive can be overwritten in any model file again.

A new tool: LPLPIC.EXE

The syntax to execute LPLPIC.EXE is:

LPLPIC <modelfile>

This tool assumes that the file .RES of the model exists. If not, it must first be produced by the LPL compiler. No extension on the modelname is needed. LPLPIC allows the user to browse through the matrix of the model at text and pixel level. This is especially useful for big models. You need a CGA compatible graphic screen and a graphic driver file from Borland.