REFERENCE MANUAL OF LPL

Tony Hürlimann

Version 5.0

▪ Perolles 90 ▪ 1700 Freiburg ▪ Switzerland
▪ www.virtual-optima.com ▪ info@virtual-optima.com

VIRTUAL OPTIMA

Para mi amada mujer **Lilí**

y mis niñas adoradas **Malika** ***y*** **Selma**

Acknowledgement

I would like to thank two institutions: The *Department for Informatics (DIUF)* at the University of Fribourg (Switzerland) where I obtained all the necessary infrastructure and personal support to complete the research. The second institution, the *Swiss National Science Foundation*, financed the research: (project no. 12-55989.98).

TABLE OF CONTENTS

“C'est le temps que tu as perdu pour ta rose
qui fait ta rose si importante.”

-- A. de Saint-Exupéry

“Et moi, ..., si j'avais su comment en revenir,
je n'y serais point allé.”

Jules Verne

“Wenn Gedanken die Sprache korrumpieren,
dann kann die Sprache auch die Gedanken korrumpieren.”

George Orwell

1. INTRODUCTION

This document is the Reference Manual for the modeling language **LPL** (Linear Programming Language – or Logical Programming Language). It contains the complete syntax and semantic specification. A free version of LPL and various shorter papers are available from the **LPL-Site** at: **www.virtual-optima.com**.

1.1. WHAT IS LPL?

LPL is a structured mathematical and logical modeling and programming language with a powerful index mechanism, which allows one to build, maintain, modify, and document large linear, non-linear, and other mathematical models. It allows one to create automatically different input files – like the MPSX standard file for linear models or some evaluation code – for an optimization software package. LPL contains also an Input and Report Generator, which allows the user to input data from various sources (files, databases) and to output the results in different forms (files, databases or reports). With LPL it is possible to write models close to the conventional mathematical notation. LPL can also be used as a data manipulation language to handle and to manipulate entire multi-dimensional tables (data cubes) in an easy fashion, much like a sparse matrix manipulation system. LPL can even be used as a data modeling package to generate on the fly a 3-norm database structure. LPL includes different interesting features not found in other systems: The language supports the modeler in unit declaration, goal programming, multi-objective models, logical modeling, certain scheduling problems, a method in breaking a large model into several submodels, model documentation, and sophisticated report generation. It allows the modeler to generate a LaTeX-file to document the model and includes other features. For educational purposes, a small and robust LP-solver is included in the package. For permutation problems a Tabu-heuristic solver and a local-search solver are also integrated. Several commercial solvers are linked to LPL.

The main features of LPL are:

- A simple syntax of mathematical and logical models with indexed expressions close to the mathematical notation, and directly applicable for documentation,
- Formulation of both small *and* large LP's and other mathematical and logical models with optional separation of the data (a model instance) from the model structure,
- Availability of a powerful index mechanism, making model structuring flexible,

- An innovative and high-level Input and Report Generator for input/output handling, they can read/write whole tables from and to databases, and generate reports,
- Intermediate indexed expression evaluation (much like matrix manipulation) and data modeling capability,
- Tools for debugging and viewing the model (e.g. explicit equation listing, picture),
- Built-in text editor to enter the LPL model and an easy code to use generator,
- Fast production of the MPSX standard file and other output files,
- An open interface to most LP and MIP solver packages available on the market, open programmable interface also to non-linear solvers (using the DLL),
- A sophisticated model environment to browser and editing large models,
- A small LP-solver and a heuristic solver for permutation problems,
- A simple and complete programming language.

1.2. LPL HIGHLIGHTS

LPL contains:

- **A declarative mathematical language**: LPL can be used to formulate concisely complete mathematical linear, non-linear and logical models of large size. The data can be stored in the model itself or outside in databases. The interface to various commercial solvers is integrated and easy configurable. Real-live models with 500000 constraints have been processed and solved using LPL.
- **An algorithmic programming language**: LPL is a complete programming language, which allows one to write algorithms to pre- or post-process the data or to loop through many optimizations (while adding constraints dynamically, for example). It has all necessary control structures of another programming language. The language further permits to break down a complex model into logical modules (itself models or model-parts). The modules can be themselves entire optimization models which can be processed individually or communicate their results.
- **An optimization tool:** LPL is designed to communicate with various commercial and free mathematical linear, mixed-integer, and non-linear mathematical packages to solve large optimization problems (CPLEX, Xpress, MOPS, GLPK, lp_solve, XA, Mosek, Conopt, Loqo, a.o). LPL comes with an own LP-solver for solving small LP models, as well as with a heuristic Tabu-search solver to solve certain scheduling problems.
- **A data modeling tool**: LPL can be used to generate a database. From an LPL modeling specification one can generate automatically a SQL-script, which generates a complete database optionally loaded with data. LPL can read/write from/to various database systems (mySQL, InterBase, SyBase, Oracle, DB2, Microsoft Access, Excel, a.o.).
- **A data manipulation tool**: Like a data manipulation language (DML) in database, LPL can manipulate (join, select, project, etc.) large multi-dimensional data (data cubes) in a sparse way. The data are read from or written to databases or text files in various ways. Tools exist for generate various pivot tables from a multi-dimensional data cube in order to change the view on the fly in the modeling environment.
- **A modeling environment**: LPL comes with a powerful model browser. It allows the user to view and traverse the model in different ways and to find and modify all

elements on a single mouse click. A model editor, a table browser with pivot-table functionality, a graphical modeling- and instance viewer, a drawing tool to generates various pictures, this all is integrates in an easy to understand user interface.

- **A documentation tool**: From a LPL model specification enriched with inline model documentation text, one can generate automatically a LaTeX-file or a PDF-file to produce a printable or publishable model documentation report (requires a free LaTeX-documentation system to be installed).
- **A library in other applications**: LPL comes as a dynamic link library and its whole functionality can be used (and hidden from the user) in another application, for example in a Visual Basic, C++ or Java application.
- **An Internet solving tool**: LPL is a client- and a server-application. Installing an LPL-client on a machine and a server on another, the two can communicate: The client can send the model to the server machine where the model can be manipulated, calculated or solved by a LPL-server in the same way as on the local machine. It is completely transparent to the user (true ASP: application service providing). On the server machine, one only needs to install a Tomcat WWW-Server (a free Java-based distributed software) and an LPL-server.

1.3. BACKGROUND

The development of the LPL language was initially motivated by practical modeling tasks such as model building, modification, and documentation for large LP models. Many important elements of the LPL language have been created from practical modeling experience. The model used for planning agricultural politics in Switzerland, for example, contained about 20'000 variables and 8'000 constraints using various optimization functions. The entire model is coded in LPL consisting of about 50 submodels in the meanwhile. Since the beginning, LPL has been modified and updated many times and it is now a general-purpose modeling language which allows one to represent most LP models and many others, also non-linear mathematical as well as models containing logical constraints.

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

In this manual, an alternative tool is presented: a mathematical modeling language, called LPL. It allows one to formulate models in terms more familiar to an analyst. The fundamental idea of LPL is to write a mathematical model 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 usable for an external solver. An LPL compiler, which performs this translation, is implemented in the software. Integrated within the language is also a

Report Generator, which can write complicated reports to text files or databases. Furthermore, an *Input Generator* is available, which can read data in a flexible and complex way from text files or databases.

An LPL model is a complete and readable formulation of a mathematical model and it can also be used for documentation and expression evaluation independently of any optimization. The LPL compiler is implemented in DELPHI 2007 from Borland and was developed under Windows. However, the code is written in a highly portable way and can also be compiled with *freePascal*, a free Pascal compiler (http://www.freepascal.org/) and is, therefore, easily portable to other platforms. Furthermore, it was also compiled under Kylix (Delphi on Linux). The main design principle has been simplicity without scarifying the power; simplicity for the user of the language and simplicity for the implementer of LPL.

1.4. OVERVIEW OF THE MANUAL

Chapter 2 explains the use of the LPL software. Chapters 3 to 10 give a detailed overview of the LPL language and its syntax as well as the semantic. Chapter 3 describes the basic syntax elements of LPL. Chapter 4 contains all the information about the structure and overall semantic of an LPL model. Indices and index-lists, the most fundamental elements of LPL, are explained in detail in chapter 5. Internal data and table formats of data in a model are explained in chapter 6. Chapter 7 gives detailed information about variables and constraints. The chapter 8 explains the use of units in LPL. The Input/Report Generator of LPL, which contains the whole input/output handling, is described in chapter 9. Compiler switches and the option statement are listed and explained in chapter 10. Chapter 11 explains the use of the dynamic link library (*lpl.dll* and *lplj.dll*). The entire language syntax description is collected in Appendix A.

1.5. WHAT TO DO NEXT?

- First read the README.TXT file in the software distribution,
- If you are a new LPL user, read [Hürlimann 2006b] and do the tutor examples,
- Read the paper [Hürlimann 2004a], if you want first to get an overview of the LPL modeling language.
- Read §3 and §4 to get an idea of LPL basic syntax,

- If you need to build an application in, say, Visual Basic or Java, using the LPL capabilities and its dynamic library, read §11.

2. INSTALLING AND RUNNING LPL

LPL can be tested on the Internet and a free version can be downloaded from the LPL-Site. It contains the LPL compiler and other modeling tools, as well as a collection of tutor models. All files at the site are stored as compressed files zipped using WINZIP [www.winzip.com]. The **README.TXT** file gives detailed instructions on how to install LPL.

2.1. FUNCTIONAL OVERVIEW

LPL is a modeling language to code mathematical and other models. The LPL-compiler first parses the code and (optionally) runs it. The source code is written in LPL-syntax and is stored in a textfile, called **LPL-file**. Depending on the compiler switches, it generates various output files. It can call a solver (another program that solve mathematical problems) automatically, specified by the solver interface parameters. Depending on the solver interface, the solver writes the results into files or passes the data directly back to LPL in memory. Finally, LPL writes all output specified by the WRITE statements to the **NOM-file,** which can be viewed as a report or result file to the model.

If a syntax error occurred during the parsing/running, it is aborted immediately and -- depending on the implementation -- an editor is called and an error pointer is placed at the position where the error occurred together with an error message read from the *lplmsg.txt* file (where all compiling and running error messages are listed). At the beginning of the parsing, the file *lplcfg.lpl* -- if present and found -- is read and parsed also. If this file is in the same directory as the launched compiler it *will* be found. This file must be in LPL syntax and contains general options such as solver interface parameters, directory paths, and others.

The LPL compiler can generate other information (besides the NOM-file) about the model as files, called the XXX-files (where XXX contains three letters). These file names can be derived from the extension of the file name: For example, if `MyModel.lpl` is the LPL-file then `MyModel.nom` is the NOM-file.

LPL can write the following files:

MPS-file : a MPSX file for linear and quadratic models. The MPS-file is the known MPSX solver input file representing a LP model. Most commercial LP solvers

accept MPSX files as input. The format of the MPS file may be found in the literature. One should note that LPL does not generate a RANGE section. BOUNDS are limited to FX, UP, and LO. The COLUMN section may, however, contain the necessary markers for integer variables, such that a MIP model is correctly translated into the MPSX standard. Furthermore, LPL can generate quadratic problems extending the MPSX standard by adding a QP-matrix section as specified by the CPLEX solver. This allows the modeler to generate quadratic problems still in the MPS format.

EQU-file : an explicit equation-listing of the model. The EQU-file is a complete and explicit constraint listing of the model with all indexes expanded and removed. It is an important debugging tool.

LPO-file : a complete instantiation of the model for analysis. The LPO-file is generated by the LPL compiler, and is the most important link to some solvers. It represents the model with all indices expanded and all expressions evaluated. It contains all constraints explicitly, much like a full equation listing. The LPO-file is just a convenient way to store the fully instantiated model, like the standard MPS-file. But contrary to the MPS-file it takes less space on disk and is not limited to linear problems. Furthermore, a simple stack based interpreter can evaluate the constraints and the derivatives which are mandatory for non-linear solvers. The LPO-file is not an ASCII file and cannot be displayed or printed directly.

LPX-file : a solution file of the model.

INT-file : an human readable form of the LPO-file.

TEX-file : a LaTeX file for documenting the model (see §10.6).

SQL-file : an SQL script for creating a complete relational database out of a LPL syntax specification (see §9).

SQ2-file : an LPL source file containing the READ/WRITE statements needed to communicate with the database created by the SQL-script file.

BUG-file : a file for debugging the model.

two **LOG-files** : which document the compiling process (files: *lpllog.txt* and *lplStat.txt*).

These files are generated using different compiler switches (see §10.4). The LPL-file itself is not generated, it is the source code written in LPL syntax of the model. The filename extension of this file must be '*.lpl*'.

2.2. PROGRAMS

LPL consists of four executables of the compiler: two console versions, a Windows

(95/98/2000/XP) version, and a dynamic link library. All executables and the library are based on exactly the same code.

2.3.1. LPLC.EXE

The executable *lplc.exe* is the console LPL compiler. It parses and runs the LPL-file, generates eventually an LPO-file or MPS-file, calls the specified solver, writes the NOM-file and exits. This execution path, however, can be modified using the compiler switches (see §10.4).

The program is called as:

```
lplc <modelfile> [ CompilerSwitches ] [ APL ]
```

where <modelfile> is the LPL-file. The LPL-file *must* have a filename extension *.lpl*. *CompilerSwitches* is empty or a set of lower case characters as specified in §10.4. The APL parameter specifies the assigned parameter list (APL) (see §10.5) to a run.

2.3.2. LPLS.EXE

The executable *lpls.exe* is the console LPL compiler/solver. It supposes that an LPO-file has already been generated by a LPL compiler and reads it. Then it calls a solver, which solves the model, finally an LPX-file is generated which contains the solution of the model, then it exits.

The program is called as:

```
LPLS <modelfile> [ CompilerSwitches ] [APL]
```

where <modelfile> is the LPO-file. The LPO-file *must* have a file extension *lpo*. *CompilerSwitches* is empty or a set of lower case characters as specified in §10.4. But only a few options are interpreted. The parameter APL is not used.

This program is used as an LPL-server for the Internet solver. A client *lplc.exe* somewhere in the Internet compiles the model and writes an LPO-file which is then transferred to the *lpls.exe* server. This can be done automatically by *lplc.exe* just by configuring the Internet-Solver. The generated LPX-file now can be retransferred to the client. *lpls.exe* must be installed in a Tomcat environment, where it can be called as a process.

2.3.3. LPLW.EXE

The executable *lplw.exe* is more than just an LPL-compiler and interpreter. It contains an entire model environment to browse and edit the model. The compiler switches are the same as for *lplc.exe*:

```
lplw <modelfile> [ CompilerSwitches ] [APL]
```

The user manual [Hürlimann 2006a] explains the modeling user interface.

2.3.4. LPL.DLL & LPLJ.DLL

The library *lpl.dll* (and *lplj.dll* for Java) is a dynamic link library that integrates the complete LPL functionality into another application. This library and its use are explained in §11.

3. BASIC LPL LANGUAGE ELEMENTS

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

3.1. BASIC CHARACTERS

The basic alphabet of LPL consists of the following characters:

```
A ... Z, a ... z _                  (letters and underscore)
0 ... 9                             (digits)
+ - * / % & ? = < > ( ) [           (and other characters)
\ ] { | } . , : ; ' $ @ # ^ " ~
```

The alphabet of LPL models only contains printable ASCII characters and can, therefore, be manipulated as ordinary text. Names (identifiers), reserved words, operators, and other elements (the "words" or tokens of the language) are formed using one or more characters. These "words" are written in sequences to form a model code.

Between the words any number of spaces (blanks), tabs or linefeed characters or other control character (character with ASCII-code <= 32) can be placed. A good style of writing a code would be to begin a new declaration or statement on a new line, and emphasize the structure of the model using indentation. Breaking a large model into smaller model components and submodels or distribute it into several files helps also to make the model readable and maintainable.

3.2. LPL TOKEN

An LPL model consists of different basic elements: the tokens (or "the words" of the language), as already mentioned. They are: (1) *Identifiers* to designate indices, elements of indices, parameters, variables, constraints, and other user name; (2) *Numbers*, *strings* and *dates* to define data; (3) *Functions* to calculate a specific value; (4) *Operators* to build expressions, and (5) various other "words" such as *comments*, *reserved words*. All these elements are called tokens (words). The different kinds of tokens are explained now in the subsequent sections.

3.3. RESERVED WORDS

Reserved words (also keywords) are an integral part of the language LPL. They cannot be used as user-defined identifiers. The reserved words are:

ABEL_DQS	**ABEL_DPL**	**ABEL_DSP**	**ABEL_QS**	**ABEL_PL**
ABEL_SP	**ABS**	**ADDCONST**	**ADDM**	**ALIAS**
AND	**ARGMAX**	**ARGMIN**	**ARCTAN**	**ASSUMPTION**
ATLEAST	**ATMOST**	**BINARY**	**BREAK**	**CEIL**
CHECK	**COL**	**CONSTRAINT**	**COS**	**DATE**
DEFAULT	**DISTINCT**	**DO**	**ELSE**	**EMPTY**
END	**EXACTLY**	**EXIST**	**EXP**	**FLOOR**
FOR	**FORALL**	**FREE**	**FREEZE**	**FROM**
FUNCTION	**IF**	**IN**	**INTEGER**	**LOG**
MAX	**MAXIMIZE**	**MIN**	**MINIMIZE**	**MODEL**
NAND	**NOR**	**NOW**	**OPTION**	**OR**
ORD	**PARAMETER**	**POSSTR**	**PRIORITY**	**PROB**
PROD	**QUERY**	**READ**	**REAL**	**RGB**
RND	**RNDN**	**ROW**	**SEMI**	**SET**
SIN	**SORT**	**SPLIT**	**SQRT**	**STRING**
SUBJECT	**SUBSTR**	**SUM**	**THEN**	**TO**
TRUNC	**UNFREEZE**	**UNIT**	**VARIABLE**	**VARIANT**
WHILE	**WRITE**	**XOR**		

The reserved words can be typed in lower- or uppercase characters, they are NOT case-sensitive (all other identifiers are case-sensitive).

3.4. IDENTIFIERS

Identifiers are used to denote *user defined objects* such as indices, their elements, variables, constraints, and parameters. The syntax of an identifier in LPL consists of a letter or an underscore followed by any combination of letters, digits or underscores. Identifiers are case-sensitive, that is, LPL distinguishes lower and upper case letters (except in reserved words). Hence, 'ImportedProducts' and 'IMPORTEDPRODUCTS' are two different identifiers. Examples:

```
Var_1
TEXT
_None
ImportedProducts
This_is_A_long_Identifier
3xY                          -- illegal, starts with a digit
Two words                    -- illegal, must not contain a space
```

LPL can handle qualified identifiers. These are identifiers containing one or several dots. Examples:

```
Submodel.a     -- denotes declaration a in model Submodel
b.a.i          -- denotes i in model a within model b
```

(See §4.4 on how to use qualified identifiers.)

3.5. NUMBERS

Numbers are constants of real type. They constitute the numerical data of the model. They may also be used to denote elements of a set. If nothing is indicated, numbers are considered as DOUBLE (8-bytes). In Boolean expressions, zero is interpreted as FALSE and any other value is interpreted as TRUE. Examples:

```
123
+234.980
-87467632.098
- 56
3.6E-3      (or 3.6e-3)  (is the same as: 0.0036)
.5          (a number can begin with a decimal point)
5.          (a number can end with a decimal point)
```

In data tables within an LPL model, a dot (.) may be used to replace a default numerical value. The default value can be defined by the user, if he did not, it is zero.

3.6. DATES

Dates are values to define a time point. They can be placed anywhere where numbers are allowed. Internally they are stored as doubles anyway, so LPL does not make any difference between dates and numbers (except in output and input). One can calculate with them in the same way as numbers, although not many operators make sense for them. In the LPL code, however, they have a specific syntax. They always begin with a '@' character followed by four digits for the year. Optional then it follows a dash and two digits for the month, another dash and two digits for the day, followed by an optional time, initiated by a 'T' and two digits for the hour, a colon and two digits for the minutes, another colon and two digits for seconds. Examples:

```
@2003-10-07T12:01:01   7th of Oct 2003 at midday plus 1 min. and 1 sec.
@2003                  at the beginning of year 2003
@2002-02               at the beginning of February 2002
@2001-05-12            at the beginning of 12th of May 2001
```

3.7. STRINGS

Strings are sequences of arbitrary characters. They can be used in expressions. They must be written within single quotes. Strings can also be used to define elements. Examples:

```
'MyString'
'another string'
'%@*456\'h$£'
```

Non-printable characters can be included within strings. They must be headed by the backslash character. The following characters are defined:

```
\a        bell
\b        backspace
\f        form feed
```

```
\n        new line
\r        carriage return
\t        tab
\\        \
\'        ' (quote)
\"        " (double quote)
```

Hence if the user needs to add tabs to the string, it must be written as follows

```
'ABC\tDEF\tGHI\n'  (string containing two tabs and a new line)
```

To break a string over several lines, a backslash must be added to the end of a line. This removes the end-of-line and all blanks at the beginning of the following line.

Note that path-names in Windows contain the backslash character: such as `c:\lpl\models`. As a consequence, this string must be written as `c:\\lpl\\models` in LPL (alternatively the syntax `c:/lpl/models` can be used in the context of file names). The non-printable characters above can also be used in comment strings delimited by "...".

3.8. DELIMITERS

A blank (the space), all character with ASCII code lower then 32 (such as *new-line* or *tab*), all characters with ASCII code greater than 127 are considered as token delimiters. They separate the words of the language.

3.9. ELEMENTS

Elements of a set have a special syntax. (They have the same syntax than the read-tokens, see the *read* statement). A sequence of characters all with ASCII code smaller than 128 and greater than 32 which does not contain any of the following 13 characters

( ) * , / : ; [|] ' " =

or a single quoted string can be used as an element name in a set. Examples:

```
4to5
06-31-93
<g.u#45$>
'an element'
'l\'element* '    (the backspace is an escape character)
un(hj             (illegal, because the forbidden '(' occurs)
```

Note also that the element syntax is case-sensitive. Hence, “June” and “JUNE” are considered different by the LPL compiler.

3.10. OPERATORS

Operators are used to form expressions. The following operators are defined in LPL in

decreasing precedence order:

```
+ - ~ # $       unary plus and minus, not, cardinality, base set
IN              membership of an element within a set
( )             expression nesting
^   %  ?<  ?>   power, modulo, smaller, larger
/  *            division, multiplication
<ind-op>        all indexed operators (see below)
-  + &          binary plus, minus, string concat
<rel-ops>       relational operators (see below)
AND  NAND       binary and operators
OR  NOR         binary or operators
-> <- <-> XOR ..    implication, rev. impl., equivalence,
                    exclusive OR, set range
:=              assignment operator
, |             a comma or a bar, for an expression list
```

Example:

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

which will be interpreted as:

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

The parentheses may be used to change the operator's precedence, as in:

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

The operators have the following meaning (where *x* and *y* are arbitrary expressions):

```
+ x                 return x   (unary plus)
- x                 return -x  (unary minus)
# s                 return cardinality of a set s
i IN s              return the position of element i within set s
                    or 0 if i is not in s
~ x                 return 1 if x=0, else return 0
( x )               nesting x, return x
'x'                 return string x
<numb>              return the number <numb>
<id>                return the value of the identifier id
x ^ y               return y-th power of x, error if x<0
x % y               return x modulo y
x ?< y              return the smaller of x and y
x ?> y              return the larger of x and y
x / y               division (error if y=0)
x * y               multiplication
x - y               subtraction
x + y               addition
x & y               string concatenation of x and y
x >= y              return 1 if x>=y, else 0 (str or num compar.)
x <= y              return 1 if x<=y, else 0 (str or num compar.)
x > y               return 1 if x>y, else 0  (str or num compar.)
x < y               return 1 if x<y, else 0  (str or num compar.)
x <> y              return 1 if x<>y, else 0 (str or num compar.)
x = y               return 1 if x=y, else 0  (str or num compar.)
x AND y             return true if x and y are both true
x OR y              return true if at least one of x or y is true
x XOR y             return true if either x or y is true
x<->y               return true if x XOR y is false
x->y                return true if x is false or y is true
x<-y                return true if x is true or y is false
x NAND y            return true if at least one of x or y is
                    false
x NOR y             return true if both x and y are false
x := y              return the value of x (side-effect: assigns
```

```
                        the value of y to x)
x , y                   return y (side effect: evaluates both: x , y)
```

Examples:

```
2^4+2                   =     18
3=2                     =     0  (=FALSE)
3<2-6                   =     1  (=TRUE)
1 or 6-6                =     1  (=TRUE)
~(0 and 1+1)            =     1  (=TRUE)
a:=3 , 4                =     4  (side effect: assign 3 to a)
```

There is no explicit TRUE or FALSE Boolean value. Like in C, the numerical values 1 (or any value different from zero) means TRUE and 0 means FALSE. This means that an expression such as *a<>0* can always be written simply as *a*. Since *a* can be a real, this rises a problem of precision. All numerical comparison operators on reals are considered within LPL as operators with a small range. *a<>0*, for example is interpreted as TRUE (=1) if and only if *a* is within the range [-ε,ε], where ε is a small number (typically, ε=1E-8).

The IN operator tests, whether a particular element is within a specific set. The first argument must be a local index (see §5.5) or a bounded index; the second argument must be an unbounded index (see binding).

Example: Suppose the following entities are defined:

```
SET i := / 1:10 /;   -- the set {1,2,3,4,5,6,7,8,9,10}
SET j := / 2:4 /;    -- the set {2,3,4}
PARAMETER a{i};
```

then the following expressions

```
SUM{i | i IN j} a[i]   -- sum all a[i], such that i is an element in j
SUM{j} a[j IN i];      -- the same
```

both sums up the three following values: `a[2] + a[3] + a[4]`.

The relational operators are: '>=' '<=' '>' '<' '<>' and '=' . They compare numeric or alphanumeric values. If one argument is an index and the other evaluates to a number then the index is interpreted as a position (see §5.5ff) within the set. If one argument is a string, the index is interpreted as element-name and an alphanumeric comparison takes place. Example:

```
VARIABLE x{i,j | i<j};    -- if position i is less than position j }
VARIABLE y{i,j | i<'AS'}; -- for all i, such that the element-name is
                          -- alphabetically before 'AS'
```

The first statement declares a variable for every tuple (i,j), if the element of i has a position which is less than the position of the element of j (all sets are ordered in LPL). This, actually, declares the upper right triangular matrix of x{i,j}, if i and j have

the same numbers of elements.

The second expression declares a variable for every tuple (i,j) such that the name of i (the element within the set i) comes alphabetically before 'AS'.

3.11. INDEXED OPERATORS

The operators SUM, PROD, FORALL, EXIST, MAX, MIN, ARGMAX, ARGMIN, COL, ROW, ATLEAST, ATMOST, EXACTLY, FOR, (indexed-)AND, OR, XOR, NOR, and NAND are called index-operators. They are defined in LPL as follows:

```
SUM{i} a[i]           return the sum of vector a over index i
PROD{i} a[i]          return the product of vector a over index i
FORALL{i} a[i]        same as: AND{i} a[i]
EXIST{i} a[i]         same as: OR{i} a[i]
MIN{i} a[i]           return the smallest a[i] within the vector a
MAX{i} a[i]           return the largest a[i] within the vector a
ARGMIN{i} a[i]        return the position of the smallest a[i]
ARGMAX{i} a[i]        return the position of the largest a[i]
COL{i} a[i]           expands a write/read horizontally
ROW{i} a[i]           expands a write/read vertically
ATLEAST(k){i} a[i]    return true, if at least k of all a[i] are
                      true
ATMOST(k){i} a[i]     return true, if at most k of all a[i] are
                      true
EXACTLY(k){i} a[i]    return true, if exactly k of all a[i] are
                      true
AND{i} a[i]           return true if all a[i] are true
OR{i} a[i]            return true if at least one a[i] is true
XOR{i} a[i]           return true if exactly one a[i] is true
NAND{i} a[i]          return true if not all a[i] are true
NOR{i} a[i]           return true if all a[i] are false
FOR{i} (A)            return A (side-effect: executes A #i times)
{i} x[i]              return x[#i], side effect: run through i
```

These operators have the same meanings as the corresponding sum and product operators ($\sum$ and $\prod$), the 'min' and 'max' function over an index in mathematics, and the $\forall$- and $\exists$-operator in the predicate logic etc. COL and ROW are only used for the Input and Report Generator (see §10). ARGMIN and ARGMAX have the same meaning as MIN and MAX, but they return the position of the smallest or largest element instead of the element itself. ATLEAST, ATMOST, and EXACTLY are logical operators to formulate expressions such as “at least *k* of all elements in a vector are zero”. They return TRUE or FALSE. FOR is an iterator like for in C or Pascal to define loops. The five operators AND, OR, XOR, NOR, and NAND are binary operators, but they also can be used as index-operators. The index-operators are always followed by an index-list (see §5). Examples:

```
SUM{i} 1              evaluates to n, if n is the number of
                      elements of set i (same as: #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
MAX{i} a[i]           return the largest a[i]
MIN(i | a>0) a[i]     return  the smallest of all a[i] greater zero
```

```
EXIST{i} a[i]          tests, whether any a[i] is not zero. If yes,
                       it evaluates to the first position with
                       a[i]<>0 else it return 0
FORALL{i} a[i]         return 1 if all a[i]<>0 else it return 0
AND{i} a[i]            return TRUE if all a[i] are TRUE
PARAMETER a{i,j,k};
VARIABLE x{i,j | FORALL{k} a[i,j,k]} ;
```

The last expression means: There exists a variable x{i,j} for every {i,j}-tuple, 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 also be written as:

```
VARIABLE x{ i,j | AND{k} a[i,j,k] } ;
```

or by explicitly mention all elements in k:

```
VARIABLE x{i,j | a[i,j,1]<>0 AND a[i,j,2]<>0 AND ... AND a[i,j,#k]<>0 } ;
```

where #k is the number of elements in k.

The EXIST index-operator has a similar meaning. Its underlying operator is the OR operator:

```
PARAMETER a{i,j,k} ;
VARIABLE x{ i,j | EXIST{k} a[i,j,k] } ;
```

The last expression means: There exists a variable x(i,j) for every {i,j}-tuple, if for all k at least one a[i,j,k] is not zero. This could have been written as:

```
VARIABLE x{ i,j | OR{k} a[i,j,k] } ;     -- is the same
```

or by explicitly mention all elements in k:

```
VARIABLE x{i,j|a[i,j,1]<>0 OR a[i,j,2]<>0 OR ... OR a[i,j,#k]<>0};
```

where #k is again the number of elements in k.

Hence, the OR operator -- used as index-operator -- is exactly the same as the EXIST operator and the AND operator is the same as the FORALL operator.

Note that the XOR, NOR, and NAND operators can be used as binary or as index-operators.

3.12. FUNCTIONS

The following algebraic functions are defined in LPL (where *x*, *y* and *z* are arbitrary expressions):

```
NOW                    return the actual Date/Time
ABS(x)                 return the positive value of x
ARCTAN(x)              return the ArcTan of x
BREAK(x)               return x (side-effect: leaves a FOR/WHILE
                       loop)
CEIL(x)                return the smallest integer greater than x
FLOOR(x)               return the greatest integer smaller than x
TRUNC(x)               return x truncated to an integer
SIN(x)                 return the sinus of x
```

```
COS(x)                return the cosines of x
ARCTAN(x)             return the arc tangens of x
LOG(x)                return the natural logarithm of x, error if
                      x•0
EXP(x)                return e^x
SQRT(x)               return the square root of x, error if x<0
RND(x,y)              return an uniform random value in the
                      interval [x,y]
RNDN(x,y)             return a normal distributed value with mean x
                      and standard deviation y
IF(x,y,z,...)         return y if x is true , else return z
SORT(a)               return 1, (a must be a parameter) side-
                      effect: a is sorted in a decreasing way.
SPLIT(id,id1,char)    return 1, slits each id1 (string) into a list
                      to id, the split character is char
POSSTR(x,y)           return the position of substring x within y
                      (zero is x is not a substring in y)
SUBSTR(x,y,z)         return the substring of string x at position
                      y with length z (null string of not possible)
ORD(x)                return the ASCII-code of x (x is a char).
WHILE(x,y)            return y (side-effect: executes y as long as
                      x is true)
```

The IF is evaluated as follows: If *x* is TRUE, then *y* is returned else *z* is returned. If *z* is omitted then the third parameter is considered to be zero.

Examples:

```
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
```

The **IF** can be used also with more than three arguments. For example, the expression:

```
IF(a,b,c,d,e)
```

returns *b* if *a* is TRUE, else it returns *d* if *c* is TRUE, else it returns *e*. It is like a switch-statement in C: every second expression (beginning with the first one) is a condition to check: the first that is true is fixed and the subsequent expression is evaluated and its result is returned.

The **SORT**(a) function is somewhat special. Its return value is always 1, but the side-effect is to sort the index of *a*.

Example: Let *b{i}* be a parameter index over *i*.

```
set i := /1:10/;
parameter b{i} := [2 3 5 1 7 8 4 6 0 9];
parameter a{i} := b[i];
parameter dummy{i} := sort(a);
```

First the parameter values of b are copied to a. Then a is sorted. However the result is NOT a list of values from b, it is the indexes of b. Hence a{i} is:

10 6 5 8 3 7 2 1 4 9 in this order

because 9 is the 10th element within b, 8 is the 6^{th} element in a, etc.

The **SPLIT**(a,b,c) function return s 1, but its side-effect is to split strings b into a. Example:

```
set i := /1:5/;
string parameter b{i} := [‘1,2,3’ ‘4,,5’, ‘6’ ‘8,9,10’ ‘3,4’ ];
parameter a{i,k={1:3}} := split(a,b,’,’);
```

The result is a two-dimensional array a:

```
parameter a:= [1 2 3 , 4 0 5 , 6 0 0 , 8 9 10 , 3 4 0];
```

Note that of a is numeric then the split product are automatically cast into numbers.

3.13. COMMENTS

Three kinds of comments are defined in LPL. The first two kinds are skipped by the compiler, the third is read.

(1) A comment can be inserted between tokens anywhere in the model. They are delimited by the symbols (* and *). Example:

```
(* This is a comment *)
(* comment (*  nested comment *)  ends first comment *)
```

The (* ... *) comments can be nested. This kind of comment is just skipped by the compiler.

(2) A second kind of comment -- useful for short remarks -- is restricted to one line. All character beginning by a -- (double dash) up to an end-of-line is considered as a comment. Example:

```
PARAMETER a;   -- here is a short comment
```

(Note that the -- comment is not interpreted as ordinary comment in Format B tables and in text files read by a READ statement.)

(3) There is another comment enclosed within double quotes " ... " called comment attribute. It is read and memorized by the compiler. Note that these comments can span over several lines. Like strings, they can contain the same non-printable characters. The length is not limited.

3.14. OVERVIEW OF ALL TOKENS

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

```
+                   unary or binary plus operator
-                   unary or binary minus operator
*                   multiply operator
/                   division op., begin/end data format B
&                   string concatenation operator
%                   modulo operator, read/write format specifier
^                   power operator
<                   less operator
>                   greater operator
=                   equal operator, match-operator in database
                    communication
(  )                to bracket index-lists, to change operator
                    precedence in expressions
[  ]                to bracket applied index-lists, lower/upper
                    bound specification
```

```
{  }                    to bracket an index-list
' ... '                 string: to enclose alias names of elements,
                        and string literals
" ... "                 to enclose a comment attribute
|                       data tables delimiter for multi-declaration
                        and expression selector in index-lists
.                       decimal point, or default value in data
                        tables
,                       to list items in sets, data tables, index-
                        list operator, index-list sep., in format A
                        and B
:                       define operator, write format delimiter
;                       terminal delimiter of a statement
#                       cardinality operator
$                       set transform operator
~                       NOT operator
@                       date/time data
```

Tokens consisting of more than one character are:

```
--                      begin a one-line comment
(*                      begin a comment
*)                      end comment
<>                      not-equal operator
<=                      less or equal than operator
>=                      greater or equal than operator
:=                      assign operator
->                      logical implication
<-                      reverse implication
<->                     logical equivalence
?<                      smaller-than operator
?>                      larger-than operator
<Reserved words>        meaning explained in context
<Identifiers>           user defined entity
<Numbers>               to define numeric data
<Dates>                 to define a date
<Strings>               to define a string
```

3.15. EXPRESSIONS

The different tokens, such as identifiers (parameter-, variable- and index-names, etc.), numbers, dates, strings, operators and functions are used to form expressions used in various contexts of the model code. Expressions written in the LPL language are very close to ordinary mathematical notation or expressions of other programming languages, such as Java. There are particularities because of the indexed operators absent from typical programming languages:

- The $\sum$ and the $\prod$ symbol used in mathematics to sum or multiply terms over indices is replaced by the reserved word SUM and PROD.
- The $\forall$- and $\exists$-operator in predicate logic are replaced by the reserved words FORALL and EXIST.
- Indices are not subscripted as in a_{ij}. Braces are used instead to enclose the index-list and the index-names are separated by commas as in *a{i,j}* and *a[i,j]* (instead of a_{ij}).

A well-formed expression always evaluates to a numerical value or to a string. If the expression is a Boolean expression, it evaluates to 1 (or another non-zero) for TRUE and 0 for FALSE. Therefore, the expression '1=2' evaluates to 0 and is interpreted as FALSE. Parameters (numerical data) 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 element (set are always considered as ordered). If indices are used in an expression, they must be bound (see §5 binding) to an index in a previous index-list (except the second argument of the IN operator). Examples:

```
4*7+7^2              is a single expression that returns 71
4^6/c                is a single expression, if 'c' is just a
                     single parameter
SUM{i} a[i]          is a single expression, since it returns only
                     one value (the sum of all a[i])
a[i,j] + c[j]        is an indexed expression, i,j must be bound
                     before (it returns a two-dimensional table)
a[i] + SUM{i}c[i]    is another indexed expression
```

Expressions are used in two different contexts within an LPL model code:

- in every statement to define and assign data (also to define constraints)
- in the index-list to limit the tuples by a condition

Examples:

```
PARAMETER composed{i} := a[i] + SUM{i} c[i];  -- assign data
. . . {i,j,k,m | a[i]>b[j] OR c[k]<d[m] }     -- in an index-list
CONSTRAINT r: x+y-23*z + 78;                  -- define a constraint
WRITE  : a+b ;                                -- write an expression
```

The comma operator is a list operator the left term is evaluated then the right term is evaluated (always in this order) and the value of the second is returned. Example:

```
PARAMETER b := a:=23 , a+c      -- assigns 23 to a and returns a+c (to b)
PARAMETER a := (b:=78 , b+6);   -- a is 84 (side effect: b:=78)
```

The assign operator in expression produces a side effect, it assigns a value to an entity. An example on how to combine the comma with the assign operator (using also a FOR loop) is the following LPL program to calculate the greatest common divisor of the two numbers 994009 and 96709 (Euclid's algorithm):

```
PARAMETER a := 994009; b := 96709; t;
WRITE: FOR{i={1:10}} IF(b=0,BREAK(a),(t:=a,a:=b,b:=t%b));
```

This feature allows the modeler to implement algorithms as expressions in LPL.

4. STRUCTURE OF AN LPL MODEL

The overall syntax of an LPL model is as follows:

```
MODEL <ModelHeader>
  <statement list>
END
```

The statements can be classified into 6 *declarations*, 13 *instructions* and the empty statement. They can be in any order. The 6 declarations are: The *set*, *parameter*, *variable*, *constraint*, *unit*, and *model* declaration. The 13 instructions are: *solve*, *read*, *write*, *check*, *option*, *variant*, *if*, *while*, *for*, *internal-proc-call*, *external-proc-call*, *model-call,* and the *assignment* instruction.

The syntax of all statements (declarations, definitions and instructions) is similar and consists of a sequence of at most 20 *attributes*: The *pretype*, *type*, *genus*, *name*, *index*, *expression*, *range*, *subject-to*, *unit*, *if*, *priority*, *probability*, *to*, *from*, *alias*, *quote*, *comment*, *default*, *string* and the *freeze* attribute. Not all statements contain all attributes. Each statement ends with a semicolon. For example in:

```
INTEGER VARIABLE x ALIAS X "Quantity to transport" [0,10] ;

     INTEGER                  is the type attribute
     VARIABLE                 is the genus attribute
     x                        is the name attribute
     "Quantity to transport"  is the comment attribute
     ALIAS X                  is the alias attribute
     [0,10]                   is the range attribute
```

Each statement always begins with one of the first four attributes (a *pretype*, *type*, *genus*, or a *name* attribute), it then follows an optional *index* attribute and the other attributes can be in any order. We now list all statement attributes.

4.1. THE STATEMENT ATTRIBUTES

Each statement consists of a sequence of the following attributes with the exception of the *if*, *for*, *while* statement – which have a special syntax.

4.2.1. THE PRETYPE ATTRIBUTE

This attribute consists of the **SEMI** or **ASSUMTION** keyword and are only used for the variable declaration, to specify if a variable is semi-continuous or an assumption (for assumption based models only).

```
SEMI VARIABLE x [2,10]; -- a semi-continuous variable with range 2 to 10.
SEMI INTEGER  y [3,5];  -- a semi-continuous integer variable with
                        -- the possible values are  0, 3, 4, and 5.
```

```
ASSUMPTION BINARY a;     -- an assumption variable
```

SEMI defines a semi-continuous or a semi-continuous integer variable. A semi-continuous (integer) variable can have value zero or values in a range [a,b]. For semi-continuous variables the range [a,b] is continuous, for semi-continuous integer variables the range [a,b] only contains integer values between a and b.

4.2.2. THE TYPE ATTRIBUTE

This attribute consists of one of the keyword **REAL**, **FREE**, **INTEGER**, **BINARY**, **DATE**, **STRING**, or **DISTINCT**. It defines the type of a declaration. If no type attribute is used, the value domain of a declaration is double (8-byte floating point) This corresponds to **REAL**. The keyword **FREE** stands for a free real variable (the lower bound is not zero but minus infinity), **INTEGER** stands for integral numbers. If it is used for variables it declares (discrete) integer variables. **BINARY** declare the values to be zero or one. Used in variables it declares logical variables (0-1 variables). **BINARY** overrules the *range* attribute. **DISTINCT** is applicable for indexed variables. It says that all variables must be integer and different from each other. LPL automatically generates the corresponding constraints. **DATE** is basically the same as **REAL**, however the input and output are dates. **STRING** declares the entity to be a sequence of characters (variable cannot be of type **STRING**). The lengths of the string do not need to be declared. Strings are allocated dynamically.

```
INTEGER PARAMETER a;      -- only integral values are allowed for a
BINARY VARIABLE b;        -- b is a 0-1 variable
DISTINCT VARIABLE x{i};   -- all variables are discrete and distinct
STRING PARAMETER c;       -- c is a string parameter
```

4.2.3. THE GENUS ATTRIBUTE

This attribute consists of one of the keyword **SET**, **PARAMETER**, **VARIABLE** **CONSTRAINT**, **UNIT**, **MODEL**, **MAXIMIZE**, **MINIMIZE**, **QUERY**, **OPTION**, **READ**, **WRITE**, **CHECK**, **IF**, **WHILE**, **FOR**, **EMPTY**, **FREEZE**, **UNFREEZE**. It tells what kind of statement it is.

The **SET**, **PARAMETER**, **VARIABLE CONSTRAINT**, **UNIT**, **MODEL** are for the six declarations: The *set*, *parameter*, *variable*, *constraint*, *unit*, and *model* declaration.

```
SET i;                  -- declares a set named i
PARAMETER a := 2;       -- declares a parameter a and assigns 2 to it
VARIABLE x; y;          -- declares variable x and y
CONSTRAINT c : x+y>=2; -- defines a constraint named c
UNIT meter;             -- defines a unit meter
```

If several consecutive statements declare the same genus then it is not necessary to repeat the genus attribute over and over again (see above the declaration of the y variable). The model declaration is special in the sense that it can contain a complete model (see §4.5).

```
MODEL submodel;         -- declares a submodel within another model
  <statement list>
END
```

Models can be recursively declared within other models (see § 4.3.6).

The keywords **MAXIMIZE**, **MINIMIZE**, **QUERY, OPTION**, **READ**, **WRITE**, **CHECK**, **IF**, **WHILE**, **FOR**, **EMPTY**, **FREEZE**, **UNFREEZE** are to start the *solve, option*, *read*, *write*, *check*, *if*, *while*, *for*, and *internal-proc-call* instruction. These statements always must start with these keywords.

```
MAXIMIZE obj: x+y;       -- solve statement named obj (maximizing)
MINIMIZE this: x-y;      -- solve statement named this (minimizing)
QUERY q1: x OR y;        -- solve statement for logic models (query)
OPTION solver:=cplex65;  -- instruction option: "use solver cplex65"
READ FROM MyFile : a;    -- read data from file MyFile
WRITE a,b,c;             -- write data a, b, and c to some file
CHECK : a>=b;            -- check for expression a>=b
IF a>12 THEN … END       -- if instruction
WHILE a<>b DO … END      -- while loop instruction
FOR {i} DO … END         -- for loop instruction
EMPTY MyModel;           -- internal-proc-call EMPTY
```

(The other three instructions (*external-proc-call*, *model-call*, and the *assignment* statement) begin with a user-defined identifier, which is the name attribute.)

4.2.4. THE NAME ATTRIBUTE

This attribute consists of a user-defined identifier. Every *set*, *parameter*, *variable*, *constraint*, *unit*, *model* declaration and every *solve* and *option* instruction has a unique user-defined name. It is used for reference in expressions and other parts of the code.

```
MINIMIZE this: x-y;  -- solve (minimized) statement name is: 'this'
VARIABLE x;          -- declares a variable with the name x
```

4.2.5. THE INDEX ATTRIBUTE

This attribute consists of a list of set names separated by commas and surrounded by braces. It is to define tables of values (vectors, matrices or higher dimensional tables) (see chapter 5).

```
SET i; j;          -- declares two sets
VARIABLE x{i,j};   -- declares a matrices x of variables
```

These previous five attributes – if used – must be in this order. The next 16 attributes can be in any order within a statement.

4.2.6. THE EXPRESSION ATTRIBUTE

This attribute defines or assigns an expression that represents the value(s) of the declared entity. It can also be a table of values. Example:

```
PARAMETER a{i} := b+c;    -- declaration of a and assigns values
PARAMETER d{i} :  b+c;    -- definition of d
```

```
CONSTRAINT h : x+y >=10;  -- defines a constraint h
```

This attribute consists of an assignment or definition operator followed by an arbitrary expression. The *assignment operator* is := and the *definition operator* is : (a colon).

4.2.7. THE RANGE ATTRIBUTE

This attribute consists of an expression (containing exactly one comma) surrounded by brackets. It defines a lower and upper bound on a numerical entity. The lower bound and upper bound are especially for variables: they are translated as bounds for the solver. LPL translates the expression automatically to lower and upper bounds of variables at the moment of solving.

```
PARAMETER a [3,5];       -- a must be in the range 3 to 5
VARIABLE x  [3-2,10+12]; -- lower bound of x is 1 upper bound is 22
```

The bounds are defined only for numerical entities otherwise they are ignored. Variables without a range indication have range [0,∞]. Parameters and free variables have the default range [-∞,∞]. Binary variable automatically have range [0,1].

4.2.8. THE SUBJECT-TO ATTRIBUTE

This attribute begins with the two keywords **SUBJECT TO** extended by a list of identifiers of constraints and of models. It is using only in a *solve* instruction. The list contains the constraints that must be included or excluded from the solving.

Example:

```
MINIMIZE obj: x+y  SUBJECT TO modelA, ~modelB, const1, ~const2;
```

This instruction says that all constraints in model *modelA*, but none in model *modelB* and the constraint *const1*, but not the constraint *const2* must be taken into account for the minimization of x+y.

4.2.9. THE UNIT ATTRIBUTE

This attribute begins with the keyword **UNIT** followed by a unit expression in bracket. It allows the modeler to add a measure to the identifier. The unit expression is much like any expression except that its syntax is very limited. It can only contain: unit identifiers, numbers, unary plus and minus, the binary multiply and divide operators. Units are explained in §8. Example:

```
UNIT meter;           -- defines a basis unit of meter
UNIT second;           -- defines a basis unit of second
PARAMETER length UNIT [meter];
VARIABLE velocity UNIT [meter/second];
```

4.2.10. THE IF ATTRIBUTE

This attribute begins with the keyword **IF** followed by an expression. It is applicable for the *constraint* declaration and the *read* and *write* and a *solve* instruction.

If applied to a *read* or *write* statement, then the entire instruction is ignored at execution time if the condition is false. Applied to a *constraint*, it means that the constraint has no effect if the condition is false. An absent *if*-attribute is interpreted as true. Applied to a *solve*-statement, it is executed or not depending on the Boolean expression of the if-attribute.

4.2.11. THE PRIORITY ATTRIBUTE

This attribute begins with the keyword **PRIORITY** followed by an expression. For integer and binary variables one can add a priority. This priority is passed over to the MIP solver whenever possible, where the way of branching is affected. Higher priority means that the variable is branched on earlier. Example:

```
BINARY VARIABLE x{i} PRIORITY a[i];
```

4.2.12. THE PROBABILITY ATTRIBUTE

This attribute begins with the keyword **PROB** followed by an expression. For assumption variables in probabilistic argumentation systems, one can add the probabilities for each assumption. The syntax is:

```
ASSUMPTION BINARY x{i} PROB a[i];
```

4.2.13. THE TO ATTRIBUTE

This attribute begins with the keyword **TO** followed by an expression, which return a string (a file name). It is used only in the *write* instruction to define the file name to which the instruction should write (see §9). Example:

```
WRITE x TO 'file.txt';         -- writes x to file 'file.txt'
STRING PARAMETER a := 'text'; -- declare a string and assign some
WRITE a TO a & '.txt' ;        -- write a to file 'text.txt'
```

4.2.14. THE FROM ATTRIBUTE

This attribute begins with the keyword **FROM** followed by an expression, which return a string (a file name). It is used only in the *read* instruction to define the file name from which the instruction must read (see §9). Example:

```
READ FROM 'file.txt' : a;  -- opens and read data a from this file
READ FROM 'db,tab' : … ;   -- reads data from database db and table tab
```

4.2.15. THE ALIAS ATTRIBUTE

This attributes begins with the keyword **ALIAS** followed by at most two unique identifiers separated by a comma. Each declaration has a unique identifier (the *name* attribute). However, sometimes it is useful to have more than one name for the same entity. The alias attribute allows the modeler to declare at most two additional names: the alias names. Normally the alias names are shortcuts which can be used in complex expressions later on in the model. Example:

```
VARIABLE ProductQuantity ALIAS X;
CONSTRAINT C : ... + X + ...;
-- instead of: CONSTRAINT C : ... + ProductQuantity + ...;
```

The user does not need to make compromises on the name length between the expressiveness and the shortness of a name. He can use both, short and long names for the same object.

The alias attribute is also useful for sets. Often a set name is used several times in the same expression but should be separately identified as in:

```
SET location;
PARAMETER link{location,location};
```

A better formulation might be

```
SET location ALIAS i,j;
PARAMETER link{i,j};
```

This formulation avoids introducning extra local indices later on, if the first and second indices in *link* have to be referenced separately.

4.2.16. THE QUOTE ATTRIBUTE

This attribute consists of a single quoted string. It is used in the *read* and *write* instruction to define the data format (see §9). It is also used in the objective function for defining various attributes (see §7.3).

4.2.17. THE COMMENT ATTRIBUTE

This attribute consists of a double quoted string. Each statement may contain this attribute. It gives a brief description of the statement. These comments are remembered by LPL and can be recalled later on. Example:

```
PARAMETER Price{i,j} "Price of commodity i in country j" ;
```

4.2.18. THE DEFAULT ATTRIBUTE

This attribute begins with the keyword **DEFAULT** followed by a number. It declares a default value for the entity, that is, the value used for a dot or for the values not assigned to the entity. If no *default* attribute is used, the default is zero.

```
PARAMETER a DEFAULT 1;    -- a has a default value of 1
PARAMETER b ;             -- b has a default value of 0
```

Only numerical defaults are allowed.

4.2.19. THE STRING ATTRIBUTE

This attribute begins with the keyword **STRING** followed by a unique user defined identifier. It is used only in *set* declarations to assign (longer) element names to each element entry. Example:

```
SET i STRING iName;
```

which is basically the same as

```
SET i;
STRING PARAMETER iName{i};
```

However, the first notation links *i* and *iName* uniquely. The *iName* elements (which is generally a more explicit name for an element) can be printed instead of the elements of *i*. In an expression the *iName* element can be accessed by *i.sname* instead of *i.ename*.

4.2.20. THE FREEZE ATTRIBUTE

This attribute consists of the keyword **FREEZE**. Applied to *set* and *parameter* declarations make them immune against **EMPTY** or *snapshot read* instructions. Example:

```
MODEL m;
  PARAMETER a := 4 FREEZE;  -- keep that value
  PARAMETER b := 5;         -- volatile
  EMPTY m;                  -- delete all data within m (except a)
  WRITE a,b;                -- b returns the default (0), but a is 4
  READ FROM 'snapshot.sps'; -- values for a does not change either
END
```

Applied to variables their values are frozen against solvers (a solver does not change their value because they are defined as fixed bounds). Applied to constraints, it means to ignore the constraint for the solver.

4.3. THE STATEMENTS

The 19 different statements are now briefly explained.

4.3.1. SET DECLARATION

The *set* statement declares or defines an *index* or a *relation* (see compound set) used in the model. An index is also called a *set* in LPL. A set consists of a list of items (called *elements*). An element may be an integer, a range of integers or a string with a

define syntax. Sets may be indexed too, and they are called *compound sets*. The *set* statement begins with the reserved word **SET** followed by zero or several index declarations or definitions. (Sets are explained in §5.) Example:

```
SET
  i;                  -- declares an index (set) called 'i' (no data)
  h := / 1 2 /;       -- declares a set h with its elements '1' and '2'
  j := /1:10/;        -- declares a set j with its elements '1' to '10'
  Seasons := / spring, summer, autumn, winter / ;
                      -- declares a set Seasons with four elements
  IndexedSet{i,h};    -- a list of (i,h) tuples
```

4.3.2. PARAMETER DECLARATION

The *parameter* statement declares or defines the data (numerical or alphanumerical) used in the model. They are normally numerical values (real or integer). Single parameters or multi-dimensional tables (vectors, matrices or n-dimensional arrays) may be defined. The dimensions are determined by the indices, which have been declared before in the *set* statement. The data may be put in predefined table-formats or they may be numerical expressions. The reserved word **PARAMETER** heads a *parameter* statement and it is followed by zero or several parameter declarations or definitions (see §6). Example:

```
PARAMETER
  a;                -- declares a single parameter a (no data)
  b := -4.678;      -- defines a single parameter b and assigns a value
  c{i};             -- declares a vector c where i is an index
                    -- the length of the array depends on the number
                    -- of the elements of i
  d{i,j};           -- declares a 2-dimensional numerical data array d
                    -- (a matrix) with index i and j
                    -- the maximal length of the matrix is the Cartesian
                    -- product over ixj)
  e{i} := b*c[i]; -- define e from other parameters (or entities)
```

`i` and `j` are index names, which must have been *declared before* in the model.

The structure `{i}` or `{i,j}` is called index-list. An index-list declares over which sets a parameter, a variable, or a constraint runs. Index-lists are also used together with index-operators. Index-lists are explained in more detail in §5. The data table formats are explained in §6.

4.3.3. VARIABLE DECLARATION

The *variable* statement declares or defines all variables used in the model. It has exactly the same structure as the *parameter* statement except that it is headed by the reserved word **VARIABLE**. Variables are discussed in §7. Example:

```
VARIABLE X;       -- declares a model variable X
VARIABLE y{i};    -- declares a vector of variables y, the length of
                  -- the array is the cardinality of set i
VARIABLE z{i,j};  -- two-dimensional array of variable z
```

For assumption based models, **VARIABLE** is preceeded by the keyword

ASSUMPTION (and then **VARIABLE** can be omitted). If **REAL ASSUMPTION** are defined, the solver is GAUSS (GLOG) otherwise it is ABEL (ALOG).

4.3.4. CONSTRAINT STATEMENT

The *constraint* statement declares or defines all constraints of the model. They are declared exactly in the same way as parameters are, which are defined by expressions, except that they are headed by the reserved words **CONSTRAINT**. A definition of a constraint assigns an arithmetical expression to the declared constraint separated by a definition operator. More information on this statement is provided in §7. Example:

```
CONSTRAINT Res: x + y - 23*z < 12;     -- defines a constraint 'Res'
CONSTRAINT q{i,j} : x[i,j] + y[i] = 0; -- two-dimensional array
                                       -- of constraint
```

4.3.5. UNIT DECLARATION

The *unit* statement allows the user to define units such as meter, kilogram, inch, yard, etc. They are used to measure numerical entities, to check their commensurability if compared or used in expressions, and to scale automatically commensurable values. Basic units (see elementary units) are just declared and derived units depend on other units (basic or derived once). The *unit* statement begins with the reserved word **UNIT** followed by the unit declarations and definitions. Example:

```
UNIT
  meter;                      -- a basic unit 'meter'
  mile     := 1800*meter;     -- a derived unit 'mile'
  sec;                        -- another basic unit
  day      := 86400*sec;      -- another derived unit
  velocity := mile/day;       -- and still another
```

What is a basic unit and what is a derived unit is entirely decided by the user. They may depend on the circumstances. Units can only be defined once in a model. They are explained in §8.

4.3.6. MODEL DECLARATION

A LPL model can be a hierarchical structure, since sub-models can be declared within models. Sub-models are like parameterless functions: they can be executed by calling them. Models nested within other models have their own name space, which means that:

1 The same name can be reused in another models without a name conflict,

2 The identifier is exportable using the dot-notation (see below).

Example:

```
MODEL m;                    -- line 1
  PARAMETER a; b; c;        -- line 2
  MODEL mm;                 -- line 3
    PARAMETER a; bb; cc;    -- line 4
  END                       -- line 5
```

```
    MODEL nn;                  -- line 6
      PARAMETER d;             -- line 7
    END                        -- line 8
END                            -- line 9
```

Two models (model *mm* and *nn*) are nested within the model *m*. The parameters a, b, and c on line 2 are visible within the model *m* (from line 2 to line 9) which means that they are also visible within its submodels. They are even visible outside model *m* and can be accessed by the dot notation in an expression as:

```
... m.a ... m.b ... m.c
```

The parameters *a* and *bb* on line 4 are defined in model *mm*, i.e. they are only visible inside the model *mm* which extends from line 4 to 5. They can, however be accessed from model *m*, for instance, through a dot-notation `mm.a`, and outside model *m*, through the --notation `m.mm.a`.

The path of model identifiers separated by a dot from the nested models inwards determines how the corresponding entity is accessed from outside.

The two parameters *a* (on line 2 and 4) have the same name which means that the visibility of *a* (line 2) has a "hole" from line 3 to 5. Using the identifier *a* within the model *mm* refers the parameter *a* (line 4) and not *a* (line 2). How could *a* (line 2) be referenced within model *mm*? By using the dot-notation `m.a` within the model *mm*.

The concept of MODEL is important not only for structuring the information hierarchically but also for hiding and encapsulate a piece of knowledge and to decompose the entire structure in manageable modules.

Models can be declared *after* their call instruction within other models. This is called forward referencing.

4.3.7. SOLVE STATEMENT

The *solve* statement declares or defines objective functions (if any) of the model. They are declared exactly in the same way as constraints, which are defined by expressions, except that they are headed by the reserved words **MINIMIZE**, **MAXIMIZE** or **QUERY**. A definition of an objective assigns an arithmetical expression to the declared objective separated by a definition operator. More information on this statement is provided in §7. Example:

```
MAXIMIZE obj : x+y;                    -- statement to maximize x+y
```

4.3.8. READ STATEMENT

The *read* statement defines the data input of the model. The model data can be read from files in many formats. It begins with the reserved word **READ** followed by several attributes such as a device name to read from and an indication how and what

to read. Example:

```
READ FROM 'text.txt' '%2:table' : a , b , c ;
```

This instruction reads three read-tokens from the textfile *text.txt* beginning with the second occurrence of **`table`** within the file and assigned the result to the model entities *a*, *b*, and *c* (note that *a,b,c* are three identifiers in the model). The *read* statement is explained in §9.

4.3.9. WRITE STATEMENT

The *write* statement defines the output of the model. A mask within the *write* statement (the comment attribute) defines, *how* the output has to be formatted and the expression of the *write* statement specifies *what* to output. The comment attribute is used as mask. Example:

```
WRITE 'This is the mask with two place holder: 1) %4s 2) %6.2f end'
      : 'xyz', 12.56
```

This statement uses the layout of the mask to write the output. It begins with the reserved word **`WRITE`** followed by a mask, a definition operator and an expression. The expression in the previous example is another single quoted string and a number. It outputs the mask while filling in the place holders by the expression parts. In the last example, it will output the following line:

```
This is the mask with two place holder: 1)  xyz 2)  12.56 end
```

The *write* statement may also be used independently of any mask to write tables to the output file. It is explained in §9. Example:

```
WRITE a;
```

4.3.10. CHECK STATEMENT

The *check* statement is useful to check the data consistency within the model. It begins with the reserved word **`CHECK`**. Example:

```
CHECK This{i} : a[i]>1 and a[i]<100;
CHECK "check a+b<0 has failed" : a+b < 0 ;
```

The first check tests, whether all a[i] are between 1 and 100. The second simply checks, whether a+b<0. If the check fails, the run is aborted with an error message. Hence, the three following statements check exactly the same condition:

```
CHECK This{i,j}: a[i,j] > 1 ;     -- all a[i,j] must be greater than one
CHECK {i,j} :  a[i,j] > 1 ;
CHECK : FORALL{i,j} (a[i,j] > 1) ;
```

4.3.11. OPTION INSTRUCTION

This instruction is used to predefine various global parameters of a model: the solver used, a random seed for generating randomized data and others. The different options are explained in §10. Example:

```
OPTION r := 10;                              -- set the random seed to 10
OPTION path := 'c:/lpl/models;d:/models;'; -- sets the directory path
OPTION solver := xpressSol;                  -- sets the solver to Xpress
```

4.3.12. VARIANT INSTRUCTION

This instruction allows one to execute a list of submodels at this point of execution, we call them *variant points*. The list is given in an APL parameter **@VAR**. If nothing has been indicated in the APL parameter, then the default list is executed. Example:

```
VARIANT va 'subModel1+subModel2';
```

In the previous instruction the variant is called **va**. The default sub-models to execute are **submodel1** and **submodel2** in this order. This can be override by an APL parameter as follows:

```
@VAR=va(submodel3+submodel4+submodel5)
```

In the later case, the three submodels **submodel3**, **submodel4**, and **submodel5** are executed at the variant point **va** instead.

Each model has a *default variant point*. This point in a model is before the first instruction that is neither a declaration (*set*, *parameter*, *variable*, *constraint*, *unit*, and *model* declaration), nor a *read* nor an *option* statement. The default variant point is like a nameless variant point. At this point one can also execute a list of submodels indicated by an APL parameter. Example:

```
@VAR=(submodel1+submodel2)
```

In this previous case, the submodels **submodel1** and **submodel2** are executed at the default variant point. If no APL @VAR parameter has be defined, a submodel called **data** will be executed automatically at the default variant point, if it exists.

4.3.13. IF INSTRUCTION

This instruction is for branching. It begins with the keyword **IF** followed by a (Boolean) expression. This is followed by a **THEN** keyword. After that any number of statements follow, an optional **ELSE** clause and ending with an **END** keyword. Example:

```
IF a>b THEN
  <statement list>   (to be executed if a>b is true)
ELSE
  <statement list>   (to be executed if a>b is false)
END
```

Note that no declarations can be placed in the statements lists within an *if* statement.

4.3.14. WHILE INSTRUCTION

The *while* instruction implements loops. It begins with the keyword **WHILE** followed by a (Boolean) expression. This is followed by a **DO** keyword and a sequence of statements that ends with a keyword **END**. Example:

```
WHILE a>b DO
  <statement list>    (to be executed as long as a>b is true)
END
```

No declaration can be placed in the statement lists within a *while* statement.

4.3.15. FOR INSTRUCTION

The *for* instruction is another loop statement. It loops over sets. It begins with the keyword **FOR** followed by an *index* attribute. This is followed by a **DO** keyword and a sequence of statements that ends with a keyword **END**. Example:

```
FOR{i} DO
  <statement list>     (to be executed for all elements in set i)
END
```

No declaration can be placed in the statement lists within the *for* statement.

4.3.16. INTERNAL-PROC-CALL INSTRUCTION

The four different internal-proc-call instructions begin with the keyword **EMPTY**, **FREEZE**, **UNFREEZE** followed by a list of identifiers, and **ADDCONST** followed by a constraint specification. The first is called the *empty* instruction. It clears the data store of various entities. Example:

```
EMPTY ModelY, a, b;  -- clears all the data in ModelY, the entity a and b
```

The second and third are the *freeze* and *unfreeze* instruction. It can be used to "freeze" and "unfreeze" the value of various entities. Example:

```
VARIABLE x; y;
PARAMETER a:=2;
FREEZE x, y;  -- fixes the values of the variables x and y
FREEZE a;     -- fix the value of the parameter a
UNFREEZE x;   -- now unfix the value of variable x
FREEZE MyModel;  -- freeze the constraints in MyModel
```

Freezing a parameter or a set means that its value will not be modified by a *snapshot read* or an EMPTY instruction. It can be modified by an assignment or another reading instruction. Constraints and solve statements can also be "activated" (unfreeze) or "inactivated" by the instruction which means that they are not send to the solver:

```
CONSTRAINT MyConstraint: x+y; -- define a constraint
FREEZE MyConstraint;          -- inactivates it
```

```
UNFREEZE MyConstraint;        -- activates it
```

Freezing/unfreezing a model entity means to freeze/unfreeze all constraints within a model and within all its submodels.

The *addconst* instruction has the same syntax as a constraint – except for the beginning keyword, which is **ADDCONST** instead of **CONSTRAINT**. This instruction allows adding constraints at runtime. They are used in the context of row-generation and the 'keep' option in the solve statement.

4.3.17. EXTERNAL-PROC-CALL INSTRUCTION

Procedures form two standard libraries, called **Sys** and **Draw** can be called from within a LPL model. See §10.7. Example:

```
Sys.Call('myProgram.exe parameter');  -- executes an external program
Draw.Ellipse(1,2,3,4,0);              -- draws an ellipse
```

4.3.18. MODEL-CALL INSTRUCTION

For running *MyModel* within another (main) model one just can write the instruction:

```
MyModel;
```

4.3.19. ASSIGNMENT INSTRUCTION

Sets, *parameters* and *variables* can be assigned and reassigned at any time after the declaration. The assignment can also take place at the same statement of the declaration. Example:

```
PARAMETER b;      -- declare b (n data)
b := 10;          -- now assign a value to b
b := 11;          -- reassign another value to b
PARAMETER a := 2; -- declare a parameter a and assign a value
SET s ;           -- declare a set s (no data)
s := / 1:10 /;    -- s was declared to be a set
```

Sets and *parameters* can be *defined*. The difference between ***assignment*** and ***definition*** is important. Data can be assigned at various time points of the execution; a definition, however, takes place once and only once. Example:

```
PARAMETER a:=1; b:=2;    -- declare and assign a and b
PARAMETER c : a+b;       -- a definition (c is ALWAYS a+b)
b:=20;                   -- reassign b
```

This example assigns a value 1 and 2 to *a* and *b*. The parameter c then is defined. This cannot be re-assigned. Its value will always be *a*+*b* (at the line 2 its value is 3). If later in the code, *b* is assigned to be 20 (as in the example above in line 3), then automatically *c* has a new value of 21.

4.3.20. THE EMPTY STATEMENT

The empty statement consists of a single semicolon. It has no function except it ends a sequence of declarations. For example, a sequence of *set* declarations can be written without repeating every time the keyword **SET**. So, we may write

```
SET i; j;
```

Instead of

```
SET i; SET j;
```

An empty statement ends the declaration list and a subsequent declaration must begin with the corresponding keyword (in the example **SET**):

```
SET i;; SET j;  -- second SET keyword is mandatory
```

4.4. THE DOT NOTATION

As indicated in §3.4, identifiers can be extended by a dot and another identifier, called qualified identifiers:

```
Identifier { '.' Identifier }
```

This is also called the dot-notation. The purpose of this notation is the following. If an identifier is followed by a dot and one of the following words in the list in an expression occurs, then the corresponding attribute is returned from the expression. For instance, if X is a variable then `X.nam` returns the string "X" since "X" is the name of X. As another example, `X.comment` returns the comment attribute of X. The following words after the dot can occur. They are:

Notation	or	returned value	return type (**S**tring or **N**umberic)
X.ptyp	X._01	pretype attribute of X	S
X.type	X._02	type attribute of X	S
X.genu	X._03	genus attribute of X	S
X.name	X._04	name attribute of X	S
X.indx	X._05	index attribute of X	S
X.expr	X._06	expression attribute of X	S
X.cond	X._07	condition attribute	S
X.rang	X._08	the range/subject-to attr.	S
X.uni	X._09	unit of X as string	S
X.prio	X._10	if/priority/probability att.	S
X.to1	X._11	to/from attribute	S
X.ali1	X._12	first alias of X	S
X.ali2	X._13	second alias of X	S
X.quot	X._14	quote attribute	S
X.comm	X._15	comment attribute	S
X.defa	X._16	default attribute	S
X.stri	X._17	string attribute	S
X.frez	X._18	freeze attribute	S
X.asgn	X._19	assign/define operator: (' ': none, '0' for table assignment, '1' for :=, '2' for table definition, '3' for : (definition))	S

```
X.namx     X._20   dot notated name attribute of X              S

X.rhs      X._21   right hand side value of constraint X        N
X.lhs      X._22   left hand side value of constraint X         N
X.lrhs     X._23   same as:  X.lhs-X.rhs                       N
X.low      X._24   the lower bound value of X                  N
X.upp      X._25   upper bound value of X                      N
X.dual     X._26   dual value or reduced price of X            N
X.lran     X._27   lower sensitivity range in LPs of X         N
X.uran     X._28   upper sensitivity range in LPs of X         N
X.enam     X._31   element name (X must be a set)              S
X.snam     X._32   element string name                        S
X.intn     X._33   the internal name                          S
X.extn     X._34   the external name                          S

X.stat     X._35   solver status [0..7] of model X            N
X.prty     X._36   problem type [0..15] of model X            N
X.time     X._37   elapsed time of a model run in msecs       N
```

All attributes and their corresponding dot notation can be used in expressions. Note that `X._21` to `X._34` are indexed if the entity is indexed. Example:

```
SET i:=/1:10/; PARAMETER a{i} := i; b{i} :=i*i;
               PARAMETER X{i} [a+1,b+10] DEFAULT 23;
```

Now, the following terms can be used in an expression:

```
...+ X._24[1] + ...   -- adds 2 to the expression (since a[1]+1 is 2)
...- X.upp[3] + ...   -- subtracts 19 from the expression (b[3]+10 = 19).
```

4.5. HOW IS A MODEL PROCESSED?

An LPL code is processed as follows by the LPL compiler:

1 Parse the code completely,

2 Run the main (top) model.

The LPL code is a hierarchical structure and a model within a model can be defined hierarchically as seen above. The main model is the model declared as the top model of the hierarchical structure.

Running a model means to execute all the statements in the sequence in which they are written in the code. Submodels are executed by calling them explicitly by a *model-call* instruction. Models can be declared after their use. There is a *forward reference mechanism* of the LPL parser, which allows using model call identifiers to be defined later on in the source code.

5. INDICES

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

5.1. INTRODUCTION

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 (set) of elements, say

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

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 in a matrix form – is as follows:

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

In LPL, an index is also called a set, or even index-set. This is justified by the fact that a set in LPL may not only contain a list of elements but also compound sets or relations. An item of a set is called element. From the point of view of database theory, an index defines a domain of elements. Therefore, the term domain may also be used. It may be confusing to give the same entity four different names: index, set, index-set, and domain. The justification is the following: Although they have the same meaning in LPL modeling, their functional meaning is quite different:

- Index is used in the mathematical context of *running though* some operators like in the expression $\sum_{i=1}^{n} a_i$, which is nothing else than an abbreviation of $a_1 + a_2 + \ldots + a_n$.
- Set is a fundamental notion in mathematics. There may be operations like "for each element x in the set S ($x \in S$) do something," or even more interesting "for all elements x in S with the property $P(x)$ ($\{x \in S | P(x)\}$) do something."
- Index-set is a mix up of index and set to emphasize their close links.
- Domain is a fundamental notion in database theory. Databases are (sparse) tables running through domains. Operations like select a subtable or join two

tables are important manipulations. They can be viewed as set manipulations.

Unlike in mathematics, where the elements are a range of positive integers, but like in databases, LPL also uses strings to designate elements.

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-set named 'i'
SET  i := / 1,2,3,4 /;  -- define the index-set 'i' now
```

A set is *declared* by writing just an identifier that designates the set followed by a semicolon. A set is *defined* by an identifier followed by an assignment operator and a list of elements optionally separated by commas. The list of elements are surrounded by two '/'. The number of elements in a set is not limited. The number of elements a set contains is called its cardinality. The order of the elements within a set is, in general, not important. LPL, however, also allows one to refer to the position of an element within the set. In this case, the order of the elements 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 element called position. The position of the first element is 1 and the position of the last element is the cardinality of the set.

5.2. ELEMENTS AND POSITION

The syntax of elements has already been explained in §3.9. Element names are always considered as strings, even if they are written as numbers. Hence the set

```
SET IntSet := /1, 2, 3, 5/; -- 4 elements: '1', '2', '3' and '5'.
                            -- The commas are optional
```

contains four strings as names, although they are written as integers. Note that the position of an element within a set is not the same as the element itself, since the element is a string and its position is an integer. In the following set

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

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

Identifiers may be more descriptive than integers to represent elements. Example:

```
SET seasons := / spring summer autumn winter / ;
```

The set *seasons* contains four elements: *spring*, *summer*, *autumn* and *winter*.

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

```
SET  IntegerRangeSet := /1:10/;   -- contains 10 elements
```

This is the same as the following:

```
SET  IntegerRangeSet := / 1 2 3 4 5 6 7 8 9 10 / ;
```

A colon must separate the two limits. The limits are included in the elements list. No space is allowed before or after the colon.

5.3. RELATIONS (COMPOUND SETS)

Sets can also be indexed like an parameter or variable. In this case, they declare or define a tuple-list. Example:

```
SET location := / NY BO LA /;          -- a set of plant locations
SET links{location,location} := / NY BO , NY LA , LA NY /;
```

location is a simple set. *links* is the allowed list of connections between the locations, which is a subset of the Cartesian product (*location*×*location*). This is called a compound set. Relations are a powerful mean to define multidimensional sparse subtables.

5.4. INDEX-LISTS

Indices are used in LPL to define multidimensional objects such as parameters, variables, and constraints. 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 dimension (or the arity) of that object.

In LPL, the subscripts are written differently: they are separated by commas and surrounded by braces. So x_{ij} would be written as x{i,j} in LPL. The {i,j}-part is called index-list. Index-lists are used to define parameters, variables, constraints, and relations. They are also used together with index-operators in LPL. Example:

```
PARAMETER A{i,j};          -- 2-dimensional data matrix
VARIABLE X{i,j,k,l,m};     -- 5-dimensional variable
CONSTRAINT R{i};           -- 1-dimensional constraint
 ... SUM{i,j,k} ...        -- 3-dimensional summation
```

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

```
SET  i := / 1:3 / ;
SET  j := / 1:2 / ;
PARAMETER a{i,j};
```

'{i,j}' is an index-list, '[3,2]' is a tuple of it, The complete tuple-list in lexicographic order is: { [1,1] [1,2] [2,1] [2,2] [3,1] [3,2] }. Its cardinality is 6(=3x2).

Sets, parameters, variables, and constraints can be indexed by just appending an index-list to the corresponding identifier. This defines or creates simultaneously many individual objects (parameters, variables or constraints). The number is the cardinality of the appended index-list. Every time an indexed object is used in LPL, one may think of a loop statement, which is executed in the background in lexicographic order of the tuple list. But, in general, the user need not care about that; LPL does it automatically. Example:

```
SET  i := / 1:10 / ;
PARAMETER a{i} := b[i] + 2 ;
```

This assignment defines a parameter *a* over the set *i*. The statement, however, assigns not *one* value, but ten values *simultaneously*, since the cardinality of the index-list '{i}' is 10. One may think of the parallel execution of the following ten statements

```
a[1] := b[1] + 2 ;
a[2] := b[2] + 2 ;
a[3] := b[3] + 2 ;
a[4] := b[4] + 2 ;
a[5] := b[5] + 2 ;
a[6] := b[6] + 2 ;
a[7] := b[7] + 2 ;
a[8] := b[8] + 2 ;
a[9] := b[9] + 2 ;
a[10] := b[10] + 2 ;
```

Of course, on a sequential machine these statements are executed sequentially (normally in the mentioned lexicographic order).

There are, however, situation where the user must know in which order an indexed expression is evaluated. This can be shown by the simple shifting up and down of tables by one position. Example:

```
PARAMETER
  a{i} := a[i+1];
  a{i} := a[i-1];
```

Since the expression is evaluated sequentially in lexicographic order (i from the first to the last element), the first expression does a down-shift within the table: the first element is lost, the last will be zero (the default of *a*), and all others are shifted down. Note that LPL does not generate “a out of index bound” error or something the like. Accesses to element outside of bounds or inexistent elements inside the bounds generate the default value (normally zero).

The second expression will place a zero at the first place and shift this up to the end,

such that the whole table will be filled up with zero. The whole table is lost. So be careful, when using the same identifier at the left and right-hand-side of the assign operator.

Every index-operator must also be followed by an index-list. Again, the cardinality of the index-list determines how many times the operator is executed. Example:

```
SET   i := / 1:10 / ;
SET   j := / 1:20 / ;
PARAMETER  a := SUM{i,j} b[i,j];
PARAMETER  c{i} := SUM{j} b[i,j];
```

The SUM operator, in the third assignment, sums 200 (10x20) terms and the sum is stored in a single parameter *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 index-list or in different index-lists in the same expression. This produces some ambiguity in the expression evaluation. Suppose, the following statement was written:

```
PARAMETER 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 local indices. A local index is an identifier which replaces a (global) index-name. It is used locally in an expression, much like a local variable in a programming language. The local index precedes the index it replaces in the index-list and is followed by an equal sign or the reserved word IN. Example:

```
PARAMETER a{d1=i,j,d2=i} := b[d1,d2,j] + 2 ;
PARAMETER a{d1 IN i,j,d2 IN i} := b[d1,d2,j] + 2 ;  -- is the same
```

d1 and *d2* are two local indices for *i*, which is replaced in the subsequent expression. Local indices *must* be added to index-lists if ambiguity arises. But they *can* be used everywhere as well. Some modeler may even prefer to use local indices systematically. Example:

```
PARAMETER c{index1}   := SUM{index2} b[index1,index2];
                         -- without local indices
PARAMETER c{i=index1} := SUM{j=index2} b[i,j];
                         -- with local indices
PARAMETER c{i IN index1} := SUM{j IN index2} b[i,j];
                         -- is the same
```

5.5. INDEX-LISTS WITH CONDITIONS

Every index-list can be extended with a condition beginning with the '|' character before the right brace. Example:

```
... { i,j,k | i=k AND a[i]<>12 } ...
```

This index-list is the same as the mathematical expression

$$\{i \in I, j \in J, k \in K | (i = k) \wedge a_i \neq 12\}.$$

The condition after the character '|' can be any legal (Boolean) expression. If the condition evaluates to zero (FALSE), it means that the specific tuple is not selected and must be discarded. This limits the tuple list and the resulting tuple list is a subset of the complete tuple list. Example:

```
VARIABLE  X {i,j | a[i,j]} ;
```

This statement declares a variable for every tuple of '{i,j}', such that the corresponding a[i,j] is not zero. If a{i,j} is a sparse table, x{i,j} also will be a sparse table. Subsequent use of the variable *X* discards automatically all non-existent tuples. Therefore, a subsequent expression like 'SUM{i,j}X[i,j]' may produce fewer variables than the cardinality of the complete SUM-index-list.

Another mathematical example: Suppose, the following 5 equations *R* are defined as following:

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

They can be formulated in LPL as (note the use of a local index *k*)

```
SET    t := / 1:5 / ;
CONSTRAINT  R{t} :  x[t]  =  y[t] + SUM{k=t | k<t} a[k]*z[k];
```

Also relation-names can be used in index-lists. If one needs to sum up over all links defined previously as:

```
SET location := / NY BO LA /;          -- a set of plant locations
SET links{location,location} := / NY BO , NY LA , LA NY /;
VARIABLE x{links};                     -- declared over a relation
PARAMETER a{location,location};        -- defined over the complete
tuple list
```

then this can be done in the following way:

```
... SUM{links}  x[links] ...
```

But there may be a problem: suppose one needs to sum up all a[i,i] over *links* only. It could be done in two different ways as following:

```
... SUM{i=location,j=location | links[i,j]}  a[i,j] ...
```

or

```
... SUM{links[i,j]}  a[i,j] ...
```

The first way is by using two local indices for the same set (*location*), running through the complete tuple list (full Cartesian product *location*x*location*), and selecting only the *links*. The second way is more efficient: the program runs only through all *links*. But now there are two local indices to identify them in a[i,j]. Local indices for the basic sets of relations are appended to the relation-name within the index-list and surrounded by brackets. In this case, the number of local indices must correspond to the declaration of the relation.

One can mixed them up too: If the modeler must sum *x* and *a* over links, then it can be done by

```
... SUM{k=links[i,j]}  ( a[i,j] + x[k] ) ...
```

where *k* is a local index for *links*, whereas *i* and *j* are two local indices for *location*.

5.6. APPLIED INDEX-LISTS AND BINDING

The construct

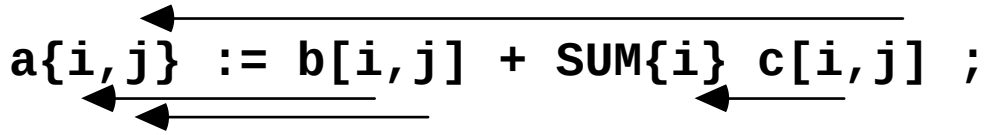
```
[ index , index , ... ]
```

used before, is called *applied index-lists*. Applied index-lists are appended to indexed entities in expressions. An applied index-list can be any expression surrounded by brackets. Example:

```
PARAMETER a{i,j} := b[i,j] + SUM{i} c[i,j];
```

[i,j] following *b* and *c* are applied index-lists.

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



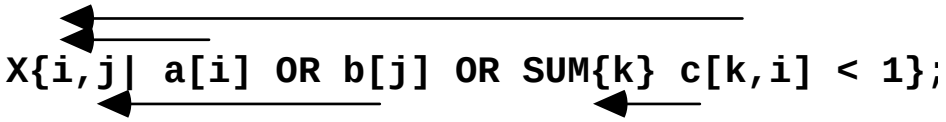
The index *i* has been used two times in an index-list (after *a* and after *SUM*). This can produce an ambiguity for the *i* in the applied index-list after *c*: It is unclear which of the previous *i*'s it is bound to. The LPL compiler interprets this automatically as shown by the arrows. The rule is: bind indices to the closest one in the syntax tree.

A specific binding can be overruled by the user, if local indices are used. They can eliminate any doubt about binding. Hence, the last example could have been written as (using local indices *k1*, *k2*, and *k3*):

```
PARAMETER a{k1=i,k2=j} := b[k1,k2] + SUM{k3=i} c[k3,k2];
```

This eliminates any doubt about the binding of *i* after *c*. If local indices are used in a systematic way then applied indexed-list contain only local indices. This may be “cleaner” and more transparent. But the user is entirely free to use local indices regularly or not. In an ambiguous expression they *must* be used, in all other they *can* be used.

Indices in a condition within the index-list must also be bound. Example:



Applied index-lists (or parts of them) may be dropped, if there is no doubt how the indices have to be bound. Therefore the last two examples can be abbreviated as:

```
PARAMETER a{i,j} := b + SUM{i} c;
VARIABLE  X { i,j | a OR b OR SUM{k} c<100 );
```

This shortens the expressions, but it might be more cryptic.

Every index used in an expression must be bound to an identical index in a previous index-list as has been explained just before, *except* it is indexed itself. In that case, their basic indices must be bound. Example:

```
SET
  i; j;                    -- two basic sets
  k{i,j};                  -- an indexed set (relation)
PARAMETER
  a{i,j} := k + i ;        -- note : k is not bound directly, but i and j
are
  a{i,j} := k[i,j] + i ;   -- the same without dropping the applied index-
list
```

An index in an applied index-list can also be replaced by an element of that index in quotes or by a number, which indicates the position of an element. Example:

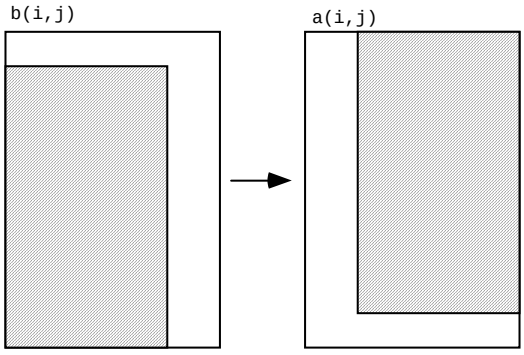
```
PARAMETER b{i,j} :=  / ...data here... /;
PARAMETER a{i}   :=  b[i,'J4'] ;
PARAMETER a{i}   :=  b[i,4] ;
```

J4 must be an element of set *j*. '4' means "put here the fourth element from *j*".

An index in an applied index-list may be an arbitrary expression. Example:

```
PARAMETER a{i,j} := b[i+3,j-4] ;
```

The next figure shows how the table *a* is copied from the table *b*. The shaded regions are data block assigned through the last statement. The white space is filled with the default value.



Circular access can be generated by a wrap around function. The *n*-th wrap around element of *i* and the *m*-th previous wrap around element can be accessed as

```
PARAMETER a{i,j} := b[(i-1+n)%#i+1 , (i-1-m+#i)%#i+1 ] ;
```

6. DATA IN THE MODEL

All numerical values or strings used in an LPL model are called data. Data can be added directly to an algebraic expression, as in '43 + 3/7' or '5*x'. Most data, however, are assigned to a parameter in the Parameter statement, which is headed by the reserved word PARAMETER followed by an identifier and an assignment operator. Then follows the data in a predefined format or as an expression. Example:

```
PARAMETER  a := 10;
PARAMETER  b{i,j,k} := 12;
```

The first declaration assigns the value 10 to the identifier *a*. It can from now on be used in an expression replacing the value 10. The second declaration assigns 12 to every tuple of *b*. (It is certainly not very useful to fill up a whole three dimensional tables with one number, but this is only for illustration.) It is much like a loop statement in a programming language. In Pascal, this statement may be represented as the following sequence of instructions, where *m*, *n*, and *p* are the cardinalities of the three sets *i*, *j*, and *k*:

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

Data do not need to be numerical; they can be strings. Strings are declared within the parameter statement by adding the reserved word STRING as type attribute. Example:

```
STRING PARAMETER
  a{i} := [ 'One' 'Two' 'Three' ];
  text :=  ['A whole phrase might be assigned to a string variable'] ;
```

(Note in the second example, that even a simple string declaration must be surrounded by “[” and “]”.) Normally, strings are read from files by the Read statement to be used in the Report Generator (Write statement). The manipulations of them within the model are limited to string comparison (see Section 4.7). Basically, they are used to write tables with the Write statement.

Data can be added directly to a LPL model in three ways: two table formats and the assignment by expressions. The first is called Format A and is a very simple format for lists (or vectors) of small sizes. The second is called the Format B. It is a powerful representation of sparse multi-dimensional tables.

6.1. FORMAT A

Suppose there are two sets defined as

```
SET i:=/1:3/; j:=/1:2/;
```

then one can declare a (two-dimensional) numerical matrix as follows:

```
PARAMETER a{i,j} := [ 10 20 . , 100 . 300 ];
```

For each tuple of parameter *a*, a value is entered in lexicographic order surrounded by brackets ([,]). Optional commas can be used to separate the data. This format is useful for small data lists. A dot means the default value. It is important to note that the set must be defined before a format A table can be entered, since the number of values between [and] is the cardinality of the full Cartesian product of its index-list.

This format can also assign string to an indexed string entity.

```
STRING PARAMETER s{i,j} := [ 'one' 'two' '' , 'four' '' 'six' ];
```

6.2. FORMAT B

The table format B is a powerful format specification to define sparse multidimensional tables. It can be used to specify sets, relations, numerical tables, or tables containing strings. It consists of a unique syntax with several options: (1) the list-option, (2) the colon-option, (3) the transpose-option, (4) the template-option, and (5) the multiple-table-option.

The table specification begins with a / and end with a /.

```
SET k  := // ;           -- k is the empty set
PARAMETER v{k} := // ;   -- v is an empty numerical vector
```

6.2.1. THE LIST-OPTION

For sets, the elements are just listed sequentially between the slashes. For a numerical data vector the corresponding elements *and* its values must be listed sequentially as in:

```
SET k := / one two three four /;              -- / {element} /
PARAMETER v{k} := / one 1  three 3  two 2 /;  -- / {element Data} /
```

There is no need to list the elements of set k two times. One could also just declare the set k and assign the data as well as the elements within the declaration of v as follows:

```
SET k;
PARAMETER v{k} := / one 1  two 2 three 3  four . /;
```

Since the fourth entry is not defined for the vector v, a dot is used instead. This defined four elements for the set k and three elements for the vector v.

For a numerical (two-dimensional) matrix c with rows i and columns j, the following declaration can be used

```
SET i  := / row1  row2  row3 /;
SET j  := / col1  col2 /;
PARAMETER c{i,j} := / row1 col1 1  row1 col2 2  row2 col2 4
                      row3 col2 6 /;
```

A comma can separate the entries. The declaration of c can also be written as

```
SET i; j;
PARAMETER c{i,j} := / row1 col1 1 , row1 col2 2 , row2 col2 4 ,
                      row3 col2 6 /;
```

Now suppose, just relations (Boolean tables instead of numerical tables) are needed. A one-dimensional table w is defined as a set, by listing its elements

```
SET k    := / one two three four /;
SET w{k} := / one two three /;
```

The vector w is nothing else then a subset of k. The advantage of declaring w{k} instead of

```
SET w := / one two three /;
```

is to constrain the elements of set w to the elements of k and to check this requirement automatically.

A two-dimensional relation b can be defined correspondingly as

```
SET i; j;
SET b{i,j} := / row1 col1  row1 col2  row2 col2  row3 col2 /;
```

Just leave out the numerical values. The elements of a relation are called tuples. In our example (*row1 col2*) is the second tuple within b, b contains therefore 4 tuples, a subset of the full Cartesian product over $i_X j$.

The declaration of string vectors is similar. They are declared as follows:

```
SET k;
STRING PARAMETER z{k} :=
    / one 'string one' two 'string two' three . four 'string four' /;
```

A two-dimensional string table is declared similarly as

```
SET i; j;
STRING PARAMETER c{i,j} :=
    / row1 col1 'row1 col1'  row1 col2 .  row2 col2 'row2 col2'
      row3 col2 'row3 col2' /;
```

Higher-dimensional (sparse) tables can be declared similarly as

```
SET i; j; k;
SET e{i,j,k} := / i1 j1 k1 , i2 j1 k2 , i3 j2 k3 , i3 j2 k4 /;
STRING PARAMETER   g{i,j,k} :=
   / i1 j1 k1 '1' , i2 j1 k2 '2' , i3 j2 k3 '3' , i3 j2 k4 '4' /;
PARAMETER f{i,j,k} :=
```

```
    / i1 j1 k1 1 , i2 j1 k2 2 , i3 j2 k3 3 , i3 j2 k4 4 /;
```

These declarations define 3 three-dimensional 3x2x4 tables, a relation e, a numerical table f, and a string table, g containing only 4 entries.

We summarize: Sparse tables can be declared just by listing the element tuples in any order. If the table is numerical or if it contains strings, then the corresponding data must be inserted right after the tuples.

6.2.2. THE COLON-OPTION

Sometimes tables are built of blocks of smaller tables. It is convenient to have an option that allows the model-builder to specify the subtables separately.

Suppose, the following sparse 14x14 numerical matrix is given (where a dot denotes an non-existent element).

	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14
r1	.	.	.	.	.	.	.	.	.	1	2	3	4	5
r2	.	.	.	.	.	.	.	.	.	6	.	8	9	10
r3	1	1	1	1	1	.	.	.	.	11	12	.	14	15
r4	.	.	.	.	.	.	.	.	.	.	.	.	.	7
r5	.	.	.	.	.	.	.	.	.	.	.	.	.	7
r6	.	.	.	.	.	.	.	4	4	4	.	4	4	7
r7	.	.	.	.	.	.	.	44	44	44	44	.	44	7
r8	.	.	.	.	.	.	.	.	.	.	.	.	.	7
r9	.	.	.	.	.	.	.	55	55	55	55	55	55	7
r10	.	.	.	.	.	.	.	.	.	.	.	.	.	.
r11	.	.	.	.	.	.	.	.	.	.	.	.	.	7
r12	.	.	.	.	.	.	.	.	.	.	.	.	.	7
r13	.	.	.	.	.	13	14	.	.	.	.	.	.	7
r14	.	.	.	.	.	12	12	.	.	.	.	.	.	7

The most simple way to specify this matrix using LPL data table formats would be to explicitly list all entries different from zeroes (or the default) -- as we have seen above in the list-option -- as following:

```
SET rows; cols;
PARAMETER mat{rows,cols} := /
   r1  c10  1
   r1  c11  2
   r1  c12  3
   .....
   r14 c7  12
   r14 c14 7   /;
```

Another way to specify the matrix is to divide it into blocks or subtables (see the "SubTable" in the syntax specification) defined as following

```
SET rows; cols;
PARAMETER mat{rows,cols} := /
    : c10 c11 c12 c13 c14 :
  r1    1   2   3   4   5
  r2    6   .   8   9  10
  r3   11   12  .  14  15

    : c1  c2  c3  c4  c5 :
```

```
r3   1   1   1   1   1

  : c8  c9  c10 c11 c12 c13 :
r6   4   4    4   .   4   4
r7  44  44   44  44  .   44
r9  55  55   55  55  55  55

  :  c6  c7 :
r14  12  12
r13  13  14

  : (tr) r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14 :
 c14      7  7  7  7  7  7   .   7   7   7   7     /;
```

The subblocks begin with a colon (:) followed by a list of elements of the last set (cols) in any order. A second colon terminates the list. Right after this, an element of the first set (rows) and as many data as there are elements between the two colons follow. This can be repeated. New-line characters are not important within the format. So, the next to the last subblock could also be written as

```
:  c6  c7 :  r14  12  12   r13  13  14
```

An interesting option inside the colon-option right after the first colon is called the transpose-option written as *(tr)*. It just reverses the last and the next to the last index in such a way that elements from the next to the last index are listed within the colons, whereas elements of the last index must be written after the second colon. This is interesting especially for two-dimensional tables, but is also valid for higher dimensional tables. The effect is obvious, if one looks at the last subblock in the matrix defined above.

The whole matrix could be considered as one block and one may also write the full matrix as:

```
SET rows; cols;
PARAMETER mat{rows,cols} := /
   :  c1  c2  c3  c4  c5  c6  c7  c8  c9 c10 c11 c12 c13 c14 :
r1     .   .   .   .   .   .   .   .   .   1   2   3   4   5
r2     .   .   .   .   .   .   .   .   .   6   .   8   9  10
r3     1   1   1   1   1   .   .   .   .  11  12   .  14  15
r4     .   .   .   .   .   .   .   .   .   .   .   .   .   7
r5     .   .   .   .   .   .   .   .   .   .   .   .   .   7
r6     .   .   .   .   .   .   .   4   4   4   .   4   4   7
r7     .   .   .   .   .   .   .  44  44  44  44   .  44   7
r8     .   .   .   .   .   .   .   .   .   .   .   .   .   7
r9     .   .   .   .   .   .   .  55  55  55  55  55  55   7
r10    .   .   .   .   .   .   .   .   .   .   .   .   .   .
r11    .   .   .   .   .   .   .   .   .   .   .   .   .   7
r12    .   .   .   .   .   .   .   .   .   .   .   .   .   7
r13    .   .   .   .   .  13  14   .   .   .   .   .   .   7
r14    .   .   .   .   .  12  12   .   .   .   .   .   .   7  /;
```

The colon-option can also be used for one or higher than two-dimensional tables, or for relations or string tables. Consider first one-dimensional tables. Using the same examples for the vectors v, w, and z as above, they can be declared as

```
SET k;
PARAMETER v{k} := /  : one  two  three  four :
                     1    2      3      .  /;
```

```
SET  w{k} := /  : one  two  three  four :
                  +    +      +     .  /;
STRING PARAMETER z{k} :=
  /  : one          two          three          four :
       'string one' 'string two' 'string three' .     /;
```

This format needs two remarks: 1) since the tables are one-dimensional, the data follow right after the second colon. No further elements are needed. 2) The declaration of w must indicate which elements are in the set w and which are not. The elements not in w are just notified as a dot and the others by any other character or string ('+' is used the subsequent examples). Note that the declaration of w could also be written as

```
SET  w{k} := /  : one  two  three :
                  +    +      + /;
```

The second remark does not change the requirement that the same number of items as elements between the colons must follow the second colon. Note this was not the case in the first example, where one can write

```
SET w{k} := / one two three /;
```

One does not need to state this as

```
SET w{k} := / one + two + three + /;  -- faulty declaration
```

Although this seems not to be consequent at first sight, it is nevertheless convenient and a little practice may convince the user that the adopted solution is simpler.

Of course, one can adopt the solution that no '+' is needed either in

```
SET  w{k} := /  : one  two  three :
                  +    +      + /;
```

One can write this just as

```
SET  w{k} := /  : one  two  three :
                                  /;
```

without the '+' indication. But in this case, the colon-option is not useful for higher dimensional relations. Consider the relation e (above) which was declared as

```
SET e{i,j,k} := / i1 j1 k1  i2 j1 k2  i3 j2 k3  i3 j2 k4 /;
```

Using the colon-option, the relation e cab be declared as

```
SET e{i,j,k} := / :  k1  k2  k3  k4 :
                 i1 j1  +   .   .   .
                 i2 j1  .   +   .   .
                 i3 j1  .   .   +   .
                 i3 j2  .   .   .   +  /;
```

We see: for higher-dimensional tables the colon-option can also be used. The elements of the last index-set are listed first between the colons. The colon-option is followed of as many elements as the table has dimensions less one in the same order in which they are listed within the index-list {...}.

The transpose-option could also be used in this case. The next to the last index-set is

just exchanged with the last index-set. Hence, the table e can also be declared as

```
SET e{i,j,k} := / : (tr)  j1  j2  :
                i1 k1      +   .
                i2 k2      +   .
                i3 k3      +   .
                i3 k4      .   +   /;
```

Correspondingly, using the colon-option the three-dimensional 3x2x4 numerical table f (see above) is declared as

```
PARAMETER f{i,j,k} := / : (tr)  j1  j2  :
                 i1 k1      1   .
                 i2 k2      2   .
                 i3 k3      3   .
                 i3 k4      .   4   /;
```

6.2.3. THE TEMPLATE-OPTION

The colon-option is a powerful method to partition a sparse multi-dimensional table into non-sparse subblocks *of the same dimension as the original table*. Sometimes it is useful to partition the table into subblocks of *lower* dimension. To do this, the modeler uses the template-option. Consider again our 14x14 matrix. The matrix can be viewed as a list of slices of (one-dimensional) row vectors. The matrix can be declared as:

```
SET rows; cols;
PARAMETER mat{rows,cols} := /
   [r1,*]  c10 1  c11 2  c12 3  c13 4  c14 5
   [r2,*]  c10 6  c12 8  c13 9  c14 10
   ... some rows are cut ...
   [r13,*]  c6 13  c7 14  c14 7
   [r14,*]  c6 12  c7 12  c14 7   /;
```

A slice begins with a template, that is, by a left bracket, or parenthesis a list of elements or stars separated by commas, and terminates with a right bracket or parenthesis. The number of elements or stars in the template must correspond to the number of indices used in the indexlist {...}. The number of the stars indicates the dimension of the slice. By default -- that is without any template -- it is supposed that the dimension of the slice is the same as the original table. This corresponds to a template containing stars only. In our 14x14 matrix the default template is [*,*]. It is not needed to write the default template, but it is not an error to write it either.

A template such as [r1,*] means that it follows a slide (or a subtable) of one dimension (since one star is used in the template). The first index (rows) is bound to the element r1 for the whole slice and the second (cols) is free. Hence, we need only to list elements of the free index-sets together with the corresponding value for numerical tables. For relations, only the element tuples of the free index-sets are listed in arbitrarily order. Therefore, if mat{rows,cols} is a relation, one can declare it as:

```
SET rows; cols;
SET mat{rows,cols} := /
   [r1,*]  c10   c11   c12   c13   c14
```

```
[r2,*]  c10   c12   c13   c14
... some rows are cut ...
[r13,*]  c6   c7   c14
[r14,*]  c6   c7   c14    /;
```

Several templates and slices can be repeated in arbitrary order. The four-dimensional 2x3x4x5 sparse table d{i,j,k,m} defined without templates as

```
SET i; j; k; m;
PARAMETER d{i,j,k,m} := / i1 j1 k1 m1 1
                        i1 j1 k1 m2 2
                        i1 j1 k2 m1 3
                        i1 j2 k2 m3 4
                        i2 j1 k3 m4 5
                        i2 j3 k3 m5 6
                        i2 j3 k4 m1 7 /;
```

can also be declared as

```
PARAMETER d{i,j,k,m} := / [i1,j1,*,*] k1 m1 1  k1 m2 2  k2 m1 3
                        [*,*,*,*]   i1 j2 k2 m3 4  i2 j1 k3 m4 5
                        [i2,j3,*,*] k3 m5 6  k4 m1 7  /;
```

Another way to define the slices is

```
PARAMETER d{i,j,k,m} := / [i1,j1,k1,*] m1 1  m2 2
                      [i1,*,k2,*]  j1 m1 3  j2 m3 4
                      [i2,*,*,*]   j1 k3 m4 5  j3 k3 m5 6  j3 k4 m1 7 /;
```

The template-option and the colon-option can also be mixed in the same table. Therefore, our four-dimensional table d can be declared as

```
PARAMETER d{i,j,k,m} := / [i1,j1,*,*] : m1 m2 :
                                 k1   1  2
                                 k2   3  .
                        [i2,*,*,*]     : m1 m4 m5 :
                                 j1 k3     .  5  .
                                 j3 k3     .  .  6
                                 j3 k4     7  .  .
                        [*,*,*,*]   i1 j2 k2 m3 4   /;
```

The template-option together with the colon-option is a powerful method to break a sparse, multi-dimensional table down into blocks (sub-tables) of different dimensions. And the syntax is straightforward and simple:

- If the sparse table contains just some unrelated tuples, the list-option is an appropriate mean,
- If the table can be broken down into homogeneous, two-dimensional sub-tables, the colon-option is your choice.
- If the table can be broken down into sub-tables of any lower dimensional tables, the template-option is the right choice.
- If the sub-tables of lower dimension are two-dimensional, then the colon-option and the template-option can be combined.
- If a transpose representation is needed, the transpose-option within the colon-option is helpful.

But there is more to come.

6.2.4. THE MULTIPLE-TABLE-OPTION

Sometimes several *similar* tables, that is with the same dimension and sparcity, are used to model the data. Suppose, one uses three sets j, k, and m. Furthermore, there are 5 three-dimensional 2x2x5 sparse tables: a relation i{j,k,m}, three numerical tables a{i} (note that i is considered a three-dimensional relation), b{j,k,m}, and c{j,k,m}, as well as a string table d{j,k,m}. Suppose they are defined as following

```
SET j; k; m;
SET i{j,k,m} :=       /      [j1,*,*] : m1  m2  m3  m4  m5 :
                                    k2    +   +   +   +   +
                                    k3    +   +   +   +   +
                             [j4,*,*] : m1  m2  m3  m4  m5 :
                                    k2    +   +   +   +   +
                                    k3    +   +   +   +   +
                             [j4,*,m6]  k4  k5  k6 /;
PARAMETER a{i}:=      /      [j1,*,*] : m1  m2  m3  m4  m5 :
                                    k2    1   2   3   4   5
                                    k3    6   7   8   9  10
                             [j4,*,*] : m1  m2  m3  m4  m5 :
                                    k2    .  12  13  14  15
                                    k3   16  17  18   .  20
                             [j4,*,m6]   k4 21  k5 22  k6 23  /;
PARAMETER b{j,k,m} := /     [j1,*,*] : m1  m2  m3  m4  m5 :
                                    k2   -1  -2  -3  -4  -5
                                    k3   -6  -7   .  -9 -10
                             [j4,*,*] : m1  m2  m3  m4  m5 :
                                    k2  -11 -12 -13 -14 -15
                                    k3  -16   . -18 -19 -20
                             [j4,*,m6]  k4 -21  k5 -22  k6 -23  /;
PARAMETER c{j,k,m} := /     [j1,*,*] : m1  m2  m3  m4  m5 :
                                    k2   31  32  33  34  35
                                    k3   36  37  38   .  40
                             [j4,*,*] : m1  m2  m3  m4  m5 :
                                    k2   41  42  43   .  45
                                    k3   46  47  48  49  50
                             [j4,*,m6]  k4 51  k5 52  k6 53  /;
STRING PARAMETER d{j,k,m} :=
                         /   [j1,*,*] : m1  m2  m3  m4  m5 :
                                   k2    a   b   c   d   e
                               k3   aa  bb  cc  dd  ee
                             [j4,*,*] : m1  m2  m3  m4  m5 :
                                   k2  aaa   . ccc  dd  eee
                                   k3 aaaa bbbb  . dddd eeeee
                             [j4,*,m6]  k4 xxx  k5 yyy  k6 zzz  /;
```

All five tables are similar. They are indices over the same sets and their sparcity is almost identical. Therefore, one can merge them together. The format B allows this by adding a list of identifiers right after the first slash enclosed by two bars. The identifiers are considered as names of tables of the same dimension. If they aren't, because they have been declared before, an error will indicate this. The five tables defined above can be declared in a compact way as follows

```
SET i; j; k; m;
PARAMETER a{i}; b{j,k,m};
STRING PARAMETER d ;
SET i{j,k,m} := / |a b c d|
                      [j1,*,*] : m1  m2  m3  m4  m5 :
                            k2    1   2   3   4   5
                                 -1  -2  -3  -4  -5
                                 31  32  33  34  35
                                  a   b   c   d   e
                            k3    6   7   8   9  10
                                 -6  -7   .  -9 -10
                                 36  37  38   .  40
                                 aa  bb  cc  dd  ee
```

```
[j4,*,*] : m1  m2  m3  m4  m5 :
      k2    .  12  13  14  15
           -11 -12 -13 -14 -15
            41  42  43   .  45
           aaa   . ccc  dd  eee
      k3    16  17  18   .  20
           -16   . -18 -19 -20
            46  47  48  49  50
          aaaa bbbb  . dddd eeeee
[j4,*,m6]  : k4  k5  k6 :
             21  22  23
            -21 -22 -23
             51  52  53
            xxx yyy zzz  /;
```

Several points have to be clarified for the multiple-table-option:

1) Tables can be declared without indicating the dimension, as it is the case for i and d. The table *a* has the dimension of *i*. But i might be a simple (static) set or a relation. Hence, there is no need do know the concrete dimension at this point. From the declaration "STRING *d*", one can only deduce that it represents a single string or a table of strings. Only at the point of the assignment the dimension is fixed. The declaration of b was given together with the indexlist {j,k,m}. But again, the explicit dimension is unknown. All one knows at this point, is that b must have at least the dimension of 3, since {j,k,m} indicates a Cartesian product of triples. But *j* or *k* or *m* might be themselves relations of multiple dimensions.

2) All these pending dimensions are resolved at the heading of the last declaration: *i* is said to be of dimension of {j,k,m}, so is *a* and *b* -- which is confirmed -- *d* gets now the same dimension and *c* is just newly created as a numerical table of the same dimension at this point. Of course, since the sets *j*, *k*, and *m* will be assigned with elements at this moment, their dimension will be fixed to one.

3) The data are listed in a natural way. If the colon-option is used, the slices of data are listed in the order of the table names declared within the two bars. So first come the data slice for *a*, then the data slice for *b*, then the data slice for *c* and finally the slice for *d*. But where are the data of *i*? Strictly speaking, the data slide of *i* should be listed first, as it were the case if *i* would have been declared independently (see above). But here another simplification at the cost of stringency has been adopted: If multiple tables are declared at the same place and if the first table is a relation, then *its* data can just be left out and they are all assigned as if all were '+' (or different from a dot). A little reflection shows that this simplification makes sense. Normally, a relation is an important entity within a model, and several data tables can be defined over the same tuple list as the relation. Hence, the relation can be implicitly defined through its data tables.

4) If no colon-option is used for a multiple table declaration, then the tuples are listed as in the single table declaration. Each element tuple is now followed by as many data as table names are declared within the two bars. As an example, the user might

declare the five tables as following

```
SET i{j,k,m} := /                | a   b   c   d |
                       j1 k2 m1   1  -1  31   a
                       j1 k2 m2   2  -2  32   b
                       j1 k2 m3   3  -3  33   c
                       ... here more to come...
                       j4 k4 m6  21 -21  51   xxx
                       j4 k5 m6  22 -22  52   yyy
                       j4 k6 m6  23 -23  53   zzz  /;
```

Again as in the single table declaration, it is not needed to explicitly assign a data element to the table *i*. Every listed tuple is assigned to *i* implicitly. Therefore, the following table declaration for *i* is perfectly correct:

```
SET i{j,k,m} := /
                       j1 k2 m1
                       j1 k2 m2
                       j1 k2 m3
                       ... here more tuples to come...
                       j4 k4 m6
                       j4 k5 m6
                       j4 k6 m6  /;
```

The reader has seen a complex and somewhat arbitrary example of the multiple-table-option. This might give the impression that this option is not very useful in practical data modeling. But the multiple-table-option is already useful in simply defining several vector over the same index. Here is an example. Suppose the modeler has a set *i* and three numerical data vector as following

```
SET   i     := / i1    i2    i3     i4     i5     i6     i7  /;
PARAMETER  a{i} := [  .     1   2.4  -6.7     8      .      .  ];
PARAMETER  b{i} := [  2     .   -23  5.56     .      .      8  ];
PARAMETER  c{i} := [  1     0    .   10.3     .      .      .  ];
```

The four statements are a correct form to represent the vectors in LPL. But there is a more concise form using the multiple-table-option:

```
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       .   /;
```

It is remarkable that all these options to define sparse multi-dimensional tables can be specified with the simple syntax of the table format B.

6.2.5. THE STAR INDEX-SET

In the context of the Format B, another feature is useful: the star index. One can define a table without using index-names, as follows:

```
SET p "products";
PARAMETER cdata{p,*} := /
```

```
            :price  quantity:
apple        1.2      234
banana       1.3      123 /;
```

This code will generate automatically a new nameless set for '*' which contains the elements 'price' and 'quantity'. More than one star can be used.

The table *cdata* can be extended later on by making further assignments to it as follows:

```
PARAMETER dummy: {p} (cdata[p,'total'] :=
                              cdata[p,'price']*cdata[p,'quantity']);
```

The star index set now contains three elements: 'price', 'quantity' and 'total' and the two-dimensional table *cdata* contains now three columns. Stared indexsets can also have a name for later reference: In this case, one has only to add an identifier just after the star as in:

```
PARAMETER cdata{p,*name};
```

'name' is now the name of the stared index set. The only difference between stared and not-stared indexes is that stared indexes do not need to be declared in the model.

6.3. ASSIGNMENT/DEFINITION THROUGH EXPRESSIONS

Any simple or indexed expression can be assigned to a parameter. The *assignment operator* is the token **:=** (a equal sign followed by a colon). Example:

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

It is also possible to ***define*** an expression using the *define operator* **:** (a colon). This means that the values always correspond to the expressions (without the need to reassign the entities defined (see also § 4.4.18).

7. VARIABLES AND CONSTRAINTS

7.1. VARIABLES

All unknowns used in an LP model are called model variables or just variables. Each variable used in an LPL model must be declared in the *variable* statement. Variables are declared in exactly the same way as parameters except that a variable declaration is headed by the reserved word **VARIABLE**. Example:

```
VARIABLE
  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,m};      -- a four dimensional variable 'z'
  w { i | i<>5 };   -- with a condition
```

Like parameters, variables may also have default values, lower and upper bounds, integer or binary type, or a unit specification. Lower and upper bounds on variables directly produce a bound constraint in the output, and an *integer* attribute produces a mixed integer model. Variables and parameters are almost the same: they have both numerical values, which can be used within any expression. Example:

```
VARIABLE
  y UNIT 1000*pound ;  -- a unit specification
  INTEGER x;           -- integer variable x
  w   [1,10];          -- bounds and integer type
```

However, there are subtle differences (1) the variables are assigned under the control of an external solver, (2) one can assign a value to a variable, but this value is considered as start value for a solution process; (3) One can even assign an expression to a variable that contains itself other variables, in this case it is considered as a constraint. Example:

```
VARIABLE  x;
PARAMETER a := x;
VARIABLE  y := 3;
VARIABLE  z : x + y;
```

In the declaration and assignment of the parameter a, the value of x (whatever it is at the actual moment) is assigned to a. In the declaration of y, y gets a start value of 3 that will be modified by the solver. In the definition of z, a constraint **z=x+y** is generated.

If the defined variable is a binary then we have four different possibilities:

```
BINARY VARIABLE  x: a*y+2 <= 6;
BINARY VARIABLE  x -> a*y+2 <= 6;
BINARY VARIABLE  x <-> a*y+2 <= 6;
BINARY VARIABLE  x <- a*y+2 <= 6;
```

The first and second lines are exactly the same. On the right of the four operators, one may have any expression containing other variables. In all cases, this is interpreted as a (logical) constraint. Again, if the right-side expression does not contain variables then it is interpreted as a starting value for the variable (zero or one).

7.2. CONSTRAINTS

The *constraint* statement contains the constraints and the objective function of a model. The reserved word **CONSTRAINT** heads the definition, followed by a constraint identifier, a colon (the definition operator), and an expression containing variables. Example:

```
CONSTRAINT t : x-y;
CONSTRAINT r : x+y^2 = 2;
CONSTRAINT re{i} :  lo(i) <= xe[i] - SUM{j} ye[i,j] <= up[i];
MAXIMIZE ma : x-y+z;
```

Constraints are automatically of binary type. If the expression is not Boolean (as the first "x-y" then it is interpreted as being greater or equal to zero. Hence, the first example above is the same as:

```
CONSTRAINT t :  x-y >= 0;
```

The same variable may be used several times in the constraint. LPL takes care of this automatically and will reduce the expression (supposing the expression is linear otherwise no reduction is done). Example:

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

will be translated by LPL as

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

A constraint can be made inactive by adding the keyword **FREEZE** as an attribute. An inactive constraint does not produce any output to the LPO-file and to the solver, only the active do. An inactive constraint can be made active by an *unfreeze* instruction. Example:

```
CONSTRAINT
  r1 : x+y+z <= a;
  r2 : x-y-z > b;
  r3 FREEZE : 2*x - y -1 < d;
  ....
UNFREEZE r3; -- reactivate the constraint r3
```

7.3. THE OBJECTIVE FUNCTION

The objective function may be defined first as inactive constraint entity as in

```
CONSTRAINT profit FREEZE:  x + y + z;
```

Later on, one may maximize the profit by writing

```
MAXIMIZE obj : profit;
```

This *solve* statement generates a model instance, calls a solver and reads the solution back to LPL (see §10.2.3) in the meantime the execution of the model run is stopped.

One can also directly define the maximizing function as

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

By default, *all* active constraints within the model are sent to the solver as well as all constraints defined in submodels called by a variant instruction. An alternative is using the *subject-to* attribute, in which one can select a list of constraints and submodel constraints:

```
MAXIMIZE profit : x + y + z  SUBJECT TO mod1,~mod2,~const1;
```

This solve statement takes all constraints of model *mod1*, but excludes all constraints of model *mod2*, and also ignores the single constraint *const1*.

The objective function can contain a switch within single apostrophes:

```
minimize obj 'mip+lp': x+y+z+a;
minimize obj 'lp': x+y+z+a;
minimize obj 'keep': x+y+z+a;
```

The string ‘lp’ means to solve the relaxed LP. ‘mip+lp’ means to solve the mip, then fix all integer and binary variables and finally solve the corresponding lp. The ‘keep’ option says to keep the solver open, so one can add constraints on the fly (with the *addconst*-instruction) in order to speed up a row-generation scenario (works actual only with CPLEX). The solver will be closed automatically at the end of a run.

MAXIMIZE or MINIMIZE statement may be used several times within the same model. Between two optimizing stage, several variables may be fixed or unfixed using the *freeze* and *unfreeze* instruction. This is useful for multi-stage modeling. Constraints may be activated or inactivated using the same keywords between two optimizing stages. Another way to activate and inactivate constraints is the IF attribute: For example:

```
CONSTRAINT c IF a>=b : x+y=9;
```

The constraint *c* is active if the condition *a>=b* is true, otherwise the constraint is not considered (same as frozen), that is, it is not sent to the solver.

LPL automatically recognizes the problem type if it runs a model. They are:

```
0    NONE  no variables or constraints are defined
1    LS    linear system (no max or min objective)
2    iLS   integer linear system
3    LP    linear program
4    MIP   mixed integer (linear) problem
5    QP    quadratic problem (obj has x'Qx, Q semi-definite)
6    iQP   quadratic integer problem
```

```
7    QCP   quadratic constraint as x'Qx, (convex constraints)
8    iQCP  quadratic constraints with integer vars
9    NLS   non-linear system
10   iNLS  non-linear integer system
11   NLP   non-linear optimization problem
12   iNLP  non-linear integer opt. problem
13   PERM  permutation problem
14   ALOG  assumption logic problem (solver is ABEL)
15   GLOG  GAUSS assumption problem (solver is GAUSS)
```

LPL checks whether all constraints are linear. If they are, it returns LS, iLS, LP, or MIP. It also checks whether at least one optimization function exists in the model if not, it returns LS, iLS, NLS, or iNLS. If the objective function is quadratic and all constraints are linear, LPL returns QP, with integer variables it is iQP. If some constraints have quadratic expressions, then it is a QCP, and with integer variables it is an iQCP. If some variables are discrete, it returns iLS, MIP, iNLS, or iNLP.

PERM is a special problem type (see §10.2.3) and can be "solved" using the integrated TABU heuristic solver. PERM problems are models in which

1 Only one variable is declared as: `DISTINCT VARIABLE x{i} [1,#i];`

2 Only an optimization function without any other constraints is declared.

The problem is to find a permutation *x* which optimizes the objective. Many problems in scheduling can be formulated as permutation problems. LPL supports this model class with an integrated heuristic TABU solver.

7.4. LOGICAL CONSTRAINTS

Logical constraints are supported. They are translated by default into a MIP formulation. Suppose one wants to impose the following constraint to an otherwise mathematical model, where *x* is a variable of type real:

```
VARIABLE x [0,100];
CONSTRAINT R : x>=20 or x<=10;
```

All variables that are used in logical constraints must be bounded explicitly. Since *R* is not a mathematical formulation of a model constraint, an LP/MIP-solver would not be able to solve the model. Therefore it is translated automatically by LPL into a set of pure mathematical constraints. The constraint *R*, for example, would be translated into the following two constraints where *d* is a newly introduced 0-1 variable.

```
CONSTRAINT R1: x >= 20*d;
CONSTRAINT R2: x + 90*(1-d) <= 100;
```

One can see that the two mathematical constraints (*R1* and *R2*) are the same as *R*, by the following reasoning: Suppose d=1, then it follows from *R1* that $x \geq 20$ and from R2 that $x \leq 100$. On the other side, if d=0 then it follows from *R1* that $x \geq 0$ and from *R2* it follows that $x \leq 10$. Therefore, *d* is a “switch” for *x* between the two intervals [0,10]

and [20,100]. (Note that the upper bound (100) on *x* is important and necessary).

Table 1 summarizes all logical operators that are defined in LPL and that can be used in the formulation of logical model constraints. Of course, all operators can also be used in Boolean expressions that are evaluated immediately (which do not contain model variables) as in

```
PARAMETER a{i,j};
VARIABLE X{i,j | ATLEAST(3) {i} a[i,j]};
```

This declaration of the variable *X* is perfectly correct. It means that the variable *X* is declared for every (i,j)-tuple, such that at least three of a row *i* in the (known) data matrix a_{ij} are different from zero.

Note that the operators AND, OR, XOR, NOR, and NAND can be used as binary operators as well as index-operators. As an example, "*AND{i} a[i]*" simply means the same as "*a[1] and a[2] and ... and a[n]*". Furthermore, "*x AND y*" can also be written as "*AND(x,y)*". It should also be noted that the *AND{}* has the same meaning as *FORALL{}* and the *OR{}* is the same as *EXIST{}*.

EXACTLY, ATLEAST, and ATMOST are index-operators with a slightly different syntax. The reserved word is followed by an integer surrounded by parentheses. The expression

```
ATMOST (4) {i} a[i];
```

means that "at most 4 out of all *a[i]* should be true (=non-zero)". If this is the case, then the expression returns true, otherwise it returns false.

Operator	Alternative formulation	Interpretation
	(x and y are any logical sub-expression containing variables)	
unary operators		
~x		x is false
binary operators		
x AND y	ATLEAST(2) (x,y)	both (x and y) are true
x OR y	ATLEAST(1) (x,y)	at least one of x or y is true
x XOR y	EXACTLY(1) (x,y)	exactly one is true (either ... or)
x -> y	~x OR y	x implies y (implication)
x <- y	x OR ~y	y implies x (reverse implication)
x <-> y	(x -> y) AND (y -> x) ~(x XOR y)	x if and only if y (equivalence)
x NOR y	~(x OR y) ~x AND ~y ATMOST(0) (x,y)	none of x and y is true (at most none is true)
x NAND y	~(x AND y) ~x OR ~y ATMOST(1) (x,y)	at most one is true (at least one is false)
indexed operators		
AND{i} x[i]	ATLEAST(#i){i} x[i]	all x[i] are true
OR{i} x[i]	ATLEAST(1){i} x[i]	at least one of all x[i] is true
XOR{i} x[i]	EXACTLY(1){i} x[i]	exactly one of all x[i] is true
NOR{i} x[i]	ATMOST(0){i} x[i]	none of all x[i] is true
NAND{i} x[i]	ATMOST(#i-1){i} x[i]	at least one of x[i] is false
FORALL{i} x[i]	ATLEAST(#i){i} x[i]	all x[i] are true

`EXIST{i} x[i]  ATLEAST(1){i} x[i]`	`at least one of all x[i] is true`
`ATLEAST(k){i} x[i]`	`at least k of all x[i] are true`
`ATMOST(k){i} x[i]`	`at most k out of all x[i] are true`
`EXACTLY(k){i} x[i]`	`exactly k out of all x[i] are true`

Table 1: logical operators in LPL

LPL also allows one to introduce predicate variables. They are simply declared as variables of type BINARY such as

```
BINARY VARIABLE MyPredicate{i};
```

To link the predicate with the rest of the otherwise mathematical model, an expression can be attached to the **predicate**. Suppose that a predicate *P* is introduced into the model with the meaning that it is true, if another (real) variable *x* is strictly between its lower (*l*) and upper (*u*) bounds. The following declaration introduces this predicate *P* and the real variable *x*, and links the predicate to the (real) variable.

```
VARIABLE x [l,u];              -- quantity x of product i
BINARY VARIABLE P -> l<=x<=u;   -- product i is manufactured
```

This declaration expresses the logical condition $P \rightarrow l \leq x \leq u$. This means that if *P* is true then *x* is strictly between the lower and upper bound.

It is also possible to link a predicate to an arbitrary, mathematical expression, such as

```
VARIABLE x [lx,ux]; y[ly,uy];
BINARY VARIABLE Q -> (x>a) or (y<b);
```

The declaration of the predicate *Q* expresses the logical condition $Q \rightarrow ((x > a) \text{ or } (y < b))$. Predicates defined in this way, are automatically translated to model constraints containing 0-1 variables by the LPL compiler.

Several LPL examples can be found in the model library.

8. UNIT STATEMENT

Most quantities in models are measured in units (dollar, hour, meter, horsepower etc.). In physics and other scientific, technical and commercial applications, using units of measure has a long tradition. It increases the reliability and the readability of calculations. Furthermore, explicit mention of units can give the compiler additional checking power. This reduces the number of syntax errors, and can let the compiler do the job of automatic unit conversion and scaling.

Units are declared and defined in a UNIT statement where the elementary units are declared as unique identifiers, and the derived units are defined through their commensurateness relationship (unit expression). An elementary unit is simply a unit that is not itself dependent on other units. Derived units, on the other hand, are based on other units and expressed by a unit expression. Two units are *commensurable* if their unit expression is the same except by a numerical factor. An example is:

```
UNIT
  gram;                    -- an elementary unit called gram
  Mile; inch; year;        -- three other elementary units
  kilo := 1000;            -- derived and dimensionless
  kg := kilo*gram;         -- derived and compound, but commensurable to
gram
  SquareMile := Mile*Mile;          -- a derived unit
  acceleration := inch/year/year;  -- another one
```

Units and unit expressions can be attached to every entity (except for SET and READ entities for which they are ignored). This is done by adding a unit attribute to the corresponing entity. An example is:

```
PARAMETER  weight UNIT [kg];                    -- unit of weight is kg
INTEGER VARIABLE   cars UNIT [1000] [0,100];    -- unit of cars is in
1000
CONSTRAINT  r  UNIT [12*kg];                    -- r is in dozen of
kilogram
```

Two entities are *commensurable* if their attached units are commensurable. So *weight* and *r* are commensurable, but *cars* and *weight* are not.

Expressions also carry a unit expression. Expressions consisting of identifiers (entity names) carry the same unit as identifiers, expressions consisting of numbers must be extended by a unit expression, encluded within brackets. Example:

```
.... +  600[hour/day]  - ....
```

All other expressions carry the unit of their operation combination. Most operators carry the same unit as their arguments. For example, the expression "*a*+*b*" carries the same unit as *a* or *b* (*a* and *b* must be commensurable). On the other hand, "*a***b*"

carries the same unit as the unit of *a* times the unit of *b*. The operators that change the unit are as follows:

```
times: unit(a*b) = unit(a)*unit(b)
divide: unit(a/b) = unit(a)/unit(b)
modulo: unit(a%b) = unit(a)/unit(b)
expo: unit(a^b) = unit(a)*unit(a)*....b times (b must be an integer)
sin: unit(sin(a)) = dimensionless
cos: unit(cos(a)) = dimensionless
sqrt: unit(sqrt(a) = sqrt(unit(a))
```

If the unit check (OPTION unitCheck) is turned on, the LPL compiler does the following:

- Checks whether two subexpressions are commensurable
- Scales the data if needed

8.1. EXAMPLE

Suppose the following piece of model is given:

```
UNIT dollar;
PARAMETER a UNIT [dollar] := 10[dollar];
  b UNIT [10*dollar]      := 100[dollar];
  c UNIT [100*dollar]     := a+b;
  a1 UNIT [dollar]        := /10/;
  b1 UNIT [10*dollar]     := /10/;
  d UNIT [100*dollar] DEFAULT 2;
WRITE a ; b; c; a1; b1; d;
WRITE xx UNIT [1/10*dollar] : a1+b1+2[dollar];
END
```

A basic unit *dollar* is declared. The parameter *a* is measured in *dollar*. The assigned expression `10[dollar]` means that *10 dollars* is assigned to *a*. The parameter *b* is measured in *10 dollars*. The assignment `100[dollar]` means that *100 dollars* is assigned to *b* independent of the measurement of *b*. Since numbers in expressions do not have units, they must be extended by a unit expression. This is done by `10[dollar]`, because an expression like `a:=10` would be illegal. Since *a* is measured in *dollar* assigning `10` (10 what?) does not make sense. However, `a:=10[dollar]` does perfectly make sense. This can be seen using another operator apart of ":=": The expression `a+10` is illegal, because *a* is measured in *dollar* but 10 has no unit, hence they are not comparable. We must write `a+10[dollar]`.

Parameter *a* and *b* are written using their units. Hence, `WRITE a` outputs 10 and `WRITE b` also outputs 10 (because 100 dollar is 10 times *10 dollar*). The parameter *c* is measured in *100 dollar*. Since *a+b* is 110 dollars, this outputs 1.1 (since 110 dollars are 1.1 times *100 dollar*).

Let's now turn to the parameters *a1* and *b1*. The numbers within /.../ (see data Format B) are interpreted differently than numbers in expressions. The declaration `a1 UNIT dollar :=/10/;` means that *a1* is measured in *dollar* and the data 10 is assigned to *a1*. This data is measured in *dollar*. The difference can now be seen in the declaration

`b1 UNIT 10*dollar:=/10/;`. *b1* is measured in *10 dollar* and 10 (of *10 dollar*) is assigned to *b1*, which gives 100 dollars (like *b*). Note the subtle and important difference. In *b* the number to assign is embedded within an expression, therefore it must carry its own unit to make the expression commensurable. In *b1* the number is data and is considered to come from outside. Therefore, its unit is determined by the units of the entity. Numbers within the Format A and B as well as numbers read from data tables via a *read statement* are considered as data and are scaled accordingly.

Numbers of DEFAULT values also are considered as data. Hence, the declaration `d UNIT 100*dollar DEFAULT 2;` means that *d* is by default *200 dollars* not *2 dollars*. But writing *d* by the command `WRITE d;` outputs 2 since *d* is measured in *100 dollar*.

The WRITE instruction can have its own UNIT of measurement as is seen in the instruction `WRITE xx UNIT 1/10*dollar : a1+b1+2[dollar];` This means: "add *a1* (10 dollars) and *b1* (100 dollars) and 2 dollars (giving 112 dollars) and write the result in *1/10 dollar* (which is 1120)".

8.2. SUMMARY OF THE SEMANTIC

1 Numerical data assigned to an entity are considered as numbers measured in the unit of the entity. So, "PARAMETER a UNIT km/sec:=/23/;" means *a* is *23 km/sec*.

2 Numerical data within the Format A and B as well as numerical data read by the READ statement and the DEFAULT value are considered as data in this sense.

3 Numbers within expressions are not considered as data and must be extended by their own unit of measurement (exa.: "...+ 10[dollar] + ..."). (Note that the IF operator has 2 or three arguments, if 2 are used then the third is considered to be zero. Using units, one must explicitly write the third argument even if it is zero. Exa.: "...+ 10[dollar] + IF(x>0, 10[dollar], 0[dollar]) + ..." . Note also that the second and third argument must be commensurable, but neither must be commensurable to the first argument which specifies the condition.)

4 While writing the entities they are written in the unit of the entity (exa.: "PARAMETER b UNIT 10*dollar :=100[dollar]; WRITE b;" will output 10.

5 Writing expressions require to extend the WRITE statement with a commensurable measurement to the assigned expression.

6 Conditions as in "x{i,j | Condition}" or in "IF(Condition,a,b)" need not be commensurable to surrounding expressions. The same is true for applied Indexlists as in "...+ a[AppliedIndexList] + ...".

The use of Units can be studied by the model **`tutor04.lpl`**.

9. INPUT AND REPORT GENERATOR

Two powerful statements (READ/WRITE statement) give the user a tool to read data from and write data to (1) plain text files, (2) snapshots or (3) databases. Together, they define the *Input and Report Generator* of LPL.

9.1. THE READ STATEMENT

The Read statement reads data from (1) text files, (2) snapshots or (3) databases. Its overall syntax is:

```
READ [ IList ] [ 'Format' ] [ FROM SExpr ]  [ : Expr ] ;
```

(1) Reading from text file is done sequentially and token-wise. A token is any sequence of characters having the same syntax as the elements (see §3.9).

A sequence of delimiters is just like one delimiter character with the exception of tabs. Several successive tabs in a text file are interpreted by LPL's READ statement as token of an empty string. The modeler indicates how and what to read from the text file using an expression. A good example to begin is the model **`tutor08.lpl`** together with the data file **`tutor.txt`**.

The syntax of the From-attribute begins with the reserved word FROM followed by an expression returning a string that must by a legal and existing filename.

Example:

```
READ FROM 'MyFile.dat' ;           -- opens 'MyFile.dat' for input
READ FROM 'a:/subdir/file.dat' ;   -- indicates a path
```

The block attribute (‘Format') defines a region (a block) within the text file to read. Without any block indication the file is considered as one single block and is read entirely in one Read statement (to an EOF mark). If there is a block attribute, it indicates the block to select from the file. The block must be marked by a user-defined string at the beginning of a line: it is called the *block-delimiter*. The block-delimiter must be an identifier or a single quoted string. A Read statement containing such a block instruction will place the read pointer at the beginning of the line after the block-delimiter, and reads until the next block-delimiter or an end-of-file is encountered. If the block-delimiter occurs several times within the file, the different blocks are numbered beginning with one and can be read with separate Read statements indicating just this number.

Example: Suppose the text file to read is “MyFile.dat” and the is “Table” and the text

file is divided into 3 such blocks (such like the tutor.txt file).

The instruction

```
READ FROM 'MyFile.dat' ;
```

opens the file *MyFile.dat* and all subsequent READ statements will read from this file, until another device name is used. The instruction

```
READ '%1:Table';
```

defines the block delimiter. The block attribute is a string having the following syntax:

```
'c [<BlockNumber>] [: <BlockDelimiter>]'
```

It begins with a character c. If c is the character % then the read is done token-wise in the same syntax as elements. If c is a different character (for example ;) then the tokens are split using just that characters. The character c is followed by <BlockNumber> which can be a number or an identifier, followed by a colon and by a block-delimiter string. If <BlockNumber> is an identifier then this identifier must be defined in the model and evaluate to a number. The instruction

```
READ '%2:Table' : <read-expr>;
```

places the read head after the second occurrence of the block-delimiter and read from the file whatever is define by <read-expr> until the next block-delimiter. If the block-delimiter does not occur within the file, then the reading head is place at the end of the file and nothing is read. The instruction

```
PARAMETER ID := 10;
STRING PARAMETER myFile := 'myFile.dat';
READ '%ID:Table' FROM myFile: <read-expr>;
```

places the read head after the tenth occurrence of the block-delimiter 'Table' and read from the file 'myFile.dat' whatever is define by <read-expr> until the next block-delimiter.

All subsequent Read statements do no longer need to define the file name or the block-delimiter. They can simply be written as:

```
READ '%2' : .... ;
READ '%3' : .... ;
READ '%1' : .... ;
```

The blocks can be read in any order. If no block number is given, the first block is read. Blocks can be read from different files in any order. In this case, the filename, the block number, and the delimiter must be added in each READ statement.

The following example reads data first from file *f1.dat* beginning with the third occurrence of '##' and ending with a subsequent '##'. Afterwards, data are read from file *f2.dat* beginning at the first '##' and ending at the next '##'. Finally, another block

is read from file *f1.dat* beginning with the second occurrence of '-----' and ending with the next '-----'.

```
READ FROM 'f1.dat' '%3:##'    :  ..... ;
READ FROM 'f2.dat'            :  ..... ;
READ FROM 'f1.dat' '%2:-----' :  ..... ;
```

A delimiter must occur at the beginning of a new line within the file.

Finally, the Read statement contains an expression, which says how and what to read from a file block. The expression is an ordinary LPL expression with some constraints: it may only contain the following operators and arguments:

, (comma) : to list identifiers in a specific order to be read and assigned

COL : index-operator to read tokens repeatedly on a line

ROW : index-operator to read repeatedly lines

set, parameter, and variable identifiers

the string ‘lf’ do skip a line

Furthermore, index-list of ROW and COL can only contain basic sets (no compound sets). ROW cannot be nested either.

The block to read is divided into read-tokens as defined above. Comments of the form (* ... *) are skipped by the read statement. The tokens are read in a sequential order and are assigned to the identifiers in the order indicated by the read-expression. Example:

```
PARAMETER a; b; c;    -- declare three numerical entities
STRING PARAMETER d;   -- declare a string entity
READ : a,d,c,b ;      -- read four tokens and assign them in this order
```

This reads four tokens and assign them to the parameters and strings *a*, *d*, *c*, and *b* in this order. If the file contains the following text to read:

```
23  34          text  56
```

then *a* gets the numerical value 23, *d* gets the string value '34', *c* gets zero since 'text' is converted to zero, and *b* will be 56. If required, the token is converted into numeric data. But no error will indicate a failure in this conversion. Furthermore, the read text does not need to correspond to the read expression. Suppose the Read statement above reads the line: **23 34 text**

or the line: **23 34 text 56 6789**

In the first case, the last identifier *b* does not get a value, in the second case, the additional text '6789' is ignored.

The real power of the Read statement comes from the two index-operators **COL** and **ROW**. They are needed to repeat and to synchronize the reading within the block of read-tokens. Example:

```
SET i;
PARAMETER a{i};
READ : ROW{i} ( i , a ) ;
```

This statement reads repeatedly the first two tokens on each line and assign the result to *i* and *a[i]*. If the read block is

```
bean   230
corn   0
rye    4571
```

then the set *i* will contain the three elements *bean*, *corn*, and *rye* and the integers will be assigned to the parameter *a*, as if the modeler had defined them within the LPL model as follows:

```
SET i := / bean , corn , rye  /;
PARAMETER a{i} := [ 230 , 0 , 4571 ];
```

The **ROW** operator also synchronizes the reading, since it begins to read on a new line, independently of the numbers of tokens on a single line present in the file. Therefore the following block will produce the same result (as above):

```
bean   230  here are other tokens
corn
(* mais 23    this line is skipped because it is a comment *)
rye    4571  456
```

The third and subsequent tokens on line 1 and 4 are ignored and the missing token on line 2 does not disturb the reading process, and *a[2]* does not get a value.
While the **ROW** operator extends readings over lines repeatedly, the **COL** operator repeats reading tokens on the same line up to an end-of-line character (eoln). The following instruction has the same effect as the 'READ : ROW ...' instruction above:

```
READ : COL{i} ( i, a[i] ) ;
```

if the data in the file are organized in one line as follows:

```
bean   230  corn   0  rye    4571
```

The **COL** operator reads two tokens repeatedly up to an eoln. The following Read statement combines the two operators and reads a two dimensional table as well as the elements for both sets

```
SET rows; columns;                  -- declare two sets
PARAMETER table{rows,columns};      -- declare a two-dimensional table
READ : COL{columns} columns ,       -- read the first line
       ROW{rows} ( rows , COL{columns} a[rows,columns] ) ;
```

And here is the block to be read as defined in the text file:

```
          A  B  C  D  E  F
bean      1  2  3  4  5  6
corn      7  8  9 10 11 12
rye      13 14 15 16 17 18
```

The first part of the expression "*COL{columns} columns*" reads the first line and assigns the tokens to the set *columns*. The next part of the expression "*ROW{rows} (rows,X)*" reads two elements on the following lines repeatedly, the first are the element-name of *rows* and the second (*X*) is another expression "*COL{columns} a[rows,columns]*" containing one token to read repeatedly on the same line up to an eoln and assigns the tokens to the corresponding *a[i,j]*. The resulting assignment is:

```
SET columns := / A B C D E F /;
    rows    := / bean corn rye  /;
PARAMETER a{rows,columns} = [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18];
```

Relations (see compound sets) can also be "read" using the Read statement. Suppose, the following sets are defined

```
SET i; j; compound{i,j};
```

Now suppose, that the file contains the tuples [i,j] each on a line as follows:

```
i1 j1
i2 j1
i2 j2
i2 j3
....
```

The read statement

```
READ : ROW{i,j} ( i , j ) ;
```

reads the data correctly. But the *compound{i,j}* relation remains empty. There is a simple way to assign also a relation through a Read statement

```
READ : ROW{i,j} ( i , j , compound ) ;
```

Since *compound[i,j]* is a relation and since *i* as well as *j* has been read already, there is no further need to read another token to define a tuple of the relation *compound*. LPL simply assigns the tuple to the indexed set *compound[i,j]* without advancing the file pointer.

(2) Reading from snapshots is another way to input data into the model. Snapshots are binary files generated by LPL previously by writing snapshots (see below). A snapshot is a file containing all data (including variable values) generated at a specific moment of a model run. If a model is solved after a lengthy calculation, for example, the user can generate a snapshot. Loading (or reading) this snapshot later on will restore exactly the same data within LPL without solving the model again.

Reading a snapshot will delete all data within LPL and storing the data from the snapshot. To read a snapshot the instruction is as follows:

```
READ FROM 'mySnap.sps';
```

The extension '.sps' is important. LPL will recognize a snapshot by this extension of the filename. READ FROM+ and READ FROM* are used in the context of reading multiple snapshots.

(3) Reading from databases is explained below.

One can specify whether the READ will be executed or not. In the following READ, the statement will be executed if *a>=b* otherwise it is ignored:

```
READ FROM 'MyFile.dat' IF a>=b : ...;
```

9.2. THE WRITE STATEMENT

A Write statement writes output to (1) text files, (2) snapshots, or (3) databases. The syntax is similar to the Read statement:

```
WRITE [List] [ IList ] [ 'Format' ] [ TO SExpr ]  [ : Expr ] ;
```

(1) Writung to text files is a powerful way to generate tables. The simplest way is write tables with a predefined format. It only uses the reserved word **`WRITE`** followed by a list of identifiers. This identifier may represent a data table, a variable (indexed or not), a constraint, a set or a string. Example:

```
WRITE a, b;
```

Another way is to output an expression as follows:

```
WRITE{i} : a[i]-b[i];
```

This generates a one-dimensional table of the given expression.

The TO attribute: By default, the Write statement write to the NOM-file -- the file with the same name as the LPL-file but with extension ‘nom’. The user does not need to open or close output files. At the first WRITE-statement, the NOM-file is opened for writing, overwriting eventually an existing NOM-file. In the process of a compilation all WRITEs are written sequentially to this file as long as the modeler does not use a TO attribute as follows:

```
WRITE TO 'NewFile.txt';  -- opens and creates the file ‘NewFile.txt’ and
                         -- erases an old one if it exists
```

After this instruction, LPL closes the previous NOM-file and creates and opens a new file for writing the subsequent outputs.

The new file is treated as write file to which new output is *appended* by default, but

this can be changed by **TO+** or **TO-** as follows:

```
WRITE TO+ 'NewFile.txt';   -- opens NewFile.txt for appending
WRITE TO  'NewFile1.txt';  -- also opens NewFile1.txt for appending
WRITE TO- 'NewFile2.txt';  -- open NewFile2.txt overwriting its content
```

If LPL was launched with an 'a' parameter then all **WRITE TO** are treated as "open for appending" regardless whether the minus sign was used or not. At the end of the compilation (or after an error occurred) the last output file is closed.

A format attribute can be added to the write statement to overrule the format. This also determines the layout of the data. The format attribute is a single quoted string with the following syntax:

```
"%" ["-"] [width] ["." prec] type
```

A format specifier (for numerical data) begins with a % char. After the % come the following, in this order:

An optional left justification indicator, ["-"]
An optional width specifier, [width]
An optional precision specifier, ["." prec]
The conversion type character, **type**

The following list summarizes the possible values for **type**:

%, ‘ ‘ (A double ‘%%’ or ‘% ‘) Escape sequence for %.

d Base-10 decimal. The argument must be an integer value. The value is converted to a string of decimal digits. If the format string contains a precision specifier, it indicates that the resulting string must contain at least the specified number of digits; if the value has less digits, the resulting string is left-padded with zeros.

u Unsigned base-10 decimal. Similar to *d* but no sign is output.

x Hexadecimal (16-base integer). The argument must be an integer value. The value is converted to a string of hexadecimal digits. If the format string contains a precision specifier, it indicates that the resulting string must contain at least the specified number of digits; if the value has fewer digits, the resulting string is left-padded with zeros.

o Octal (8-base integer). (not in Delphi, but in Java5)

b,B Format for the value of Boolean: “true” or “false” (not in Delphi).

c Outputs a single char, outputs ‘?’ for value >255. (not in Delphi).

f Fixed 10-base float. The argument must be a floating-point value. The value is converted to a string of the form "-ddd.ddd...". The resulting string starts with a minus sign if the number is negative. The number of digits after the decimal point is given by the precision specifier in the format string—a default of 2 decimal digits is assumed if no precision specifier is present.

e Scientific 10-base float in exponential notation. The argument must be a floating-point value. The value is converted to a string of the form "-d.ddd...E+ddd". The resulting string starts with a minus sign if the number is negative. One digit always precedes the decimal point.The total number of digits in the resulting string (including the one before the decimal point) is given by the precision specifier in the format string—a default precision of 15 is assumed if no precision specifier is present. The "E" exponent character in the resulting string is always followed by a plus or minus sign and at least three digits.

g General 10-base float. The argument must be a floating-point value. The value is converted to the shortest possible decimal string using fixed or scientific format. The number of significant digits in the resulting string is given by the precision specifier in the format string—a default precision of 15 is assumed if no precision specifier is present. Trailing zeros are removed from the resulting string, and a decimal point appears only if necessary. The resulting string uses fixed point format if the number of digits to the left of the decimal point in the value is less than or equal to the specified precision,

and if the value is greater than or equal to 0.00001. Otherwise the resulting string uses scientific format.

n Base-10 floating point number with locale-dependent thousand separator. The argument must be a floating-point value. The value is converted to a string of the form "-d,ddd,ddd.ddd...". The "n" format corresponds to the "f" format, except that the resulting string contains thousand separators. (not in Java).

m Money. The argument must be a floating-point value. The value is converted to a string that represents a currency amount (locale-dependent(. (Note for Delphi users: The conversion in Delphi is controlled by the CurrencyString, CurrencyFormat, NegCurrFormat, ThousandSeparator, DecimalSeparator, and CurrencyDecimals global variables or their equivalent in a TFormatSettings data structure. If the format string contains a precision specifier, it overrides the value given by the CurrencyDecimals global variable or its TFormatSettings equivalent.) (not in Java).

s String. The argument must be a character, a string, (or a PChar value, in Delphi). The string or character is inserted in place of the format specifier. The precision specifier, if present in the format string, specifies the maximum length of the resulting string. If the argument is a string that is longer than this maximum, the string is truncated.

z The fractional decimal part is transformed to the nearest rational nominator / denominator number of at most the number of digits specified in the precision specifier in the denominator. (only in LPL).

t All date/time formats (see below).

For all floating-point formats, the actual characters used as decimal and thousand separators are obtained from the LOCALE information of the operating system.

For the date/time type *t*, a second character follows – this is the standard that has been adopted also by Java 5. These date/time types are similar to but not completely identical to those defined by GNU **`date`** and POSIX **`strftime(3c)`**.

The following conversion characters are used for formatting time*:*

H Hour of the day for the 24-hour clock, formatted as two digits with a leading zero as necessary i.e. `00-23`. `00` corresponds to midnight.

k Hour of the day for the 24-hour clock, i.e. `0-23`. `0` corresponds to midnight.

I Hour for the 12-hour clock, formatted as two digits with a leading zero as necessary, i.e. `01-12`. `01` corresponds to one o'clock (either morning or afternoon).

l Hour for the 12-hour clock, i.e. `1-12`. `1` corresponds to one o'clock (morning or afternoon).

M Minute within the hour formatted as two digits with a leading zero as necessary, i.e. `00-59`.

S Seconds within the minute, formatted as two digits with a leading zero as necessary, i.e. `00-60` ("`60`" is a special value required to support leap seconds).

L Millisecond within the second formatted as three digits with leading zeros as necessary, i.e. `000-999`.

p Locale-specific morning or afternoon marker in lower case, e.g."`am`" or "`pm`".

The following conversion characters are used for formatting date:

B Locale-specific full month name, e.g. "`January`", "`February`".

b Locale-specific abbreviated month name, e.g. "`Jan`", "`Feb`".

h Same as '`b`'.

m Month, formatted as two digits with leading zeros as necessary, i.e. `01-13`, where "`01`" is the first month of the year and ("`13`" is a special value required to support lunar calendars).

A Locale-specific full name of the day of the week, e.g. "`Sunday`", "`Monday`".

a Locale-specific short name of the day of the week, e.g. "`Sun`", "`Mon`".

- Y Year, formatted to at least four digits with leading zeros as necessary, e.g. `0092` equals `92` CE for the Gregorian calendar.
- y Last two digits of the year, formatted with leading zeros as necessary, i.e. `00-99`.
- d Day of month, formatted as two digits with leading zeros as necessary, i.e. `01-31`, where "`01`" is the first day of the month.
- e Day of month, formatted as two digits, i.e. `1-31` where "`1`" is the first day of the month.

The following conversion characters are used for formatting date/time:

- R Time formatted for the 24-hour clock, same as **`"%tH:%tM"`.**
- T Time formatted for the 24-hour clock, same as **`"%tH:%tM:%tS"`**.
- r Time formatted for the 12-hour clock, same as **`"%tI:%tM:%tS %Tp"`**. The location of the morning or afternoon marker (`'%Tp'`) may be locale-dependent.
- F ISO 8601 complete date formatted, same as **`"%tY-%tm-%td"`**.
- c Date and time formatted.

Examples are given in the **`tutor20.lpl`** model file.

The format string allows the user to define the layout of the specific output. The format string may contain any characters (tabs and new-lines included), and several format specifiers, and may extend over several lines. The Write statement outputs the format string as is by substituting the format specifier. Example:

```
WRITE 'Text: %s \nValue %12.5f  again text: %7s \n'
   :  'ABCDE' , 123456.789876 , XYZ;
```

This instruction will output the following :

```
Text: ABCDE
Value 123456.78988  again text:     XYZ
```

Each subexpression (separated by a comma) is evaluated and the value replaces a format specifier. Hence, 'ABCDE' replaces '%s', '123456.78988' replaces '%12.5f', and so on.

Write expression also can contain the two indexed operators **`COL`** and **`ROW`** in a similar way as in the READ expressions. The **`COL`** operator repeats a format specifier horizontally, whereas the **`ROW`** operator repeats a format specifier vertically. Example:

```
SET i := /P1 P2 P3 P4/;
WRITE "begin %s end"   : COL{i} i;
WRITE "---  %3s ---\n"  : ROW{i} i;
```

The first write statement produces the following output to the NOM-file:

```
begin P1  P2  P3 xP4 end
```

The second write statement produces the following output:

```
---  P1  ---
---  P2  ---
---  P3  ---
```

```
---  P4  ---
```

A good example to study this Write statement is model file **`tutor09.lpl`**.

(2) Writing to a snapshot has the simple syntax:

```
WRITE TO 'mySnap.sps';
```

The extension of the filename is important and must be '.sps'. LPL writes all data (including variable values) to the snapshot file, which can be loaded by a READ later on to restore the LPL data store.

One can specify whether the WRITE will be executed or not. In the following WRITE the statement will be executed if *a>=b* otherwise the WRITE statement is ignored:

```
WRITE a,b  IF a>=b;
```

9.3. DATABASE CONNECTIVITY

The READ and WRITE statement can also be used to read from and write to databases. The interface between LPL and a database is coded in a dynamic link library called *lpldb.dll*. This library must be present.

The general syntax of the READ-statement reading from a database *db* and a *table* is:

```
READ IndexList FROM 'db,table' : Expr ;
```

where *Expr* is an expression of comma separated assignments between an LPL entity and a database field. The match-operator is = . On the left of the = operator, the LPL entity must be written and on the right, the database field (within apostrophes).

Indexlist contains a list of indices. They must also be read from the table. The expression *FROM 'db,table'* indicates the source where *db* is the database filename (or a string parameter name) and *table* is the tablename or a SQL query statement.

The WRITE-statement has almost the same syntax:

```
WRITE IndexList TO 'db,table' : Expr ;
```

The expression *Expr* is similar with the exception that the database field name is on the left hand side of the match-operator (=) and on the right hand side any expression is allowed. Another difference is of whether the data must simply be updated or inserted. To update existing records, one uses the **`TO`** (or **`TO+`** to add records) keyword. If new records must be inserted (clearing completely the table content first), one uses the **`TO-`** construct (the keyword TO with a minus sign). If we use **`TO*`** instead (TO with a star), it means that the corresponding database table and its fields

must be created. The table should not exist before. If we use **TO**** (**TO** with two stars), it means that the corresponding database must be created. If it exists, it is deleted first.

Example: Suppose the following declarations are given:

```
SET
  i ALIAS j STRING iN;
  k;
  S{i,j};
PARAMETER
  A{i};
  B{i,j};
VARIABLE
  X{i,j};
```

Suppose further that the database *MyDB* consists of the following three tables (represented by the following SQL-statements):

```
CREATE TABLE iTable (
  ID INTEGER NOT NULL PRIMARY KEY,
  iName varchar(20),
  A double,
  XX double
)

CREATE TABLE kTable (
  ID varchar(20) NOT NULL PRIMARY KEY,
  XX integer
)

CREATE TABLE ijTable (
  ID INTEGER NOT NULL  REFERENCES iTable (ID),
  ID1 INTEGER NOT NULL  REFERENCES iTable (ID),
  B double,
)
```

Then the following READ statements read from different tables or queries and have the following syntax:

```
PARAMETER para   := 5;
STRING    MyDB   := 'MyDatabase.mdb';

READ{i} FROM 'MyDB,iTable' :
  i  = 'ID',
  iN = 'iName',
  A  = 'A';
READ {i,j} FROM 'MyDB,ijTable' :
   i = 'ID',
   j = 'ID1',
   S = ('ID','ID1'),
   B = 'B';
READ {k} FROM 'MyDB,SELECT * FROM kTable WHERE XX <= :para' :
   k = 'ID';
WRITE {i} TO 'MyDB,iTable' :
  'ID' = i,
  'XX' = SUM{j} X;
```

The first READ statement reads the entities *i*, *iN* and *A* which are matched by the fields *'ID'*, *'iName'* and *'A'* in the database table *iTable*. The number of records gives the cardinality of *i*. The second READ statement reads from a relationship. Since S is a tuple list and does not correspond to a field entry, one has to say how S is to be

matched by a multiple of fields. The third READ reads from a query (a select statement). It is a parameterized query where records only are read with the condition XX<=5 true. The parameter name must be a legal parameter defined before in the LPL code and begins in the query string with a colon. Before reading, this parameter name together with the dot is replaced by the parameter value. Only then is the SELECT statement been sent to the database server.

It is also possible to call parameterized stored procedures. Suppose the database stores a query called **`Query3`** which is defined as follows (where **`XY`** is a tablename, **`tours2`** is a field and **`vari`** is a parameter):

```
SELECT * FROM XY WHERE (((InStr([tours2],vari))>0));
```

Then one can call the query within LPL as follows (**`tour`** is a singleton parameter):

```
read{i} from 'db,Query3@vari=:tour' : …… ;
```

The DB connection feature in LPL is not limited to a specific database provider. One can read/write from/to Mirosoft Access, mySQL, Oracle or even Excel in this way. The provider is specified by a *database connection string* defined as an option (see OPTION in Chap 10.2.7):

```
OPTION db := '...' ;
```

Model examples showing the DB connection are **`tutor18.lpl`**. and **`tutor19.lpl`**.

9.4. GENERATING A DATABASE FROM A LPL CODE

One can generate a complete database just from a LPL model structure. This tool supposes that certain entities in the language can be matched with entities in the database. It is obvious to match SETs in the language with keys in a table, since both must have the property, that the single elements must be different from each other. This guarantees uniqueness. Therefore, every SET defined in a model language code creates a table with a key field. Each SET in the model creates a *basic table* in the DB. Conversely, the primary keys in every 3-norm database table (which does not define a relationship nor has a foreign key) could be used to generate a basis SET in the modeling language code. Furthermore, all entities (parameters, variables, constraints etc.) in the language, which are indexed over one single index, can be seen as an additional data field in the *basic table*, in which the corresponding index-set is the primary key. The converse is true too: Every field (except the primary key field) could be identified as a one-dimensional vector, hence an entity indexed over the set that is matched with the key. However, sometimes it is more convenient to interpret fields as slices in a two-dimensional entity. This is particularly interesting, if one should iterate over field-names in the modeling language code.

LPL generates a SQL script (the SQL-file is a file with the same name as the model file and an extension '.sql') which can be interpreted by any database server and create the corresponding database. To instruct LPL to generate the SQL-file, it (lplc.exe) must be launched with 'q' as a second parameter:

```
lplc tutor08 q
```

This command will generate a file tutor08.sql. (Alternatively, there is a menu item in the tools menu in the lplw.exe.) In addition, a second file is generated (‘tutor08.sq2’) which contains a model in LPL code of the Read/Write statements. This model can then be included in the main model to read/write data from the database.

10. COMPILER DIRECTIVES AND OPTIONS

Compiler directives and the Option statement allow the user to guide the way in which a model is compiled.

10.1. COMPILER DIRECTIVES

A compiler directive in LPL is a comment beginning with the following three characters (*$. They have special meanings for the LPL compiler:

```
(*$I <filename>*)          File include
```

A compiler directive can be placed anywhere within the model where a normal comment is legal.

10.1.1. FILE INCLUDE ($I*)*

The (*$I directive redirects the LPL scanner 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 files. For an example see the **`tutor07.lpl`** model file.

Nested include files are also possible up to a level of 5. The filename must be within single quotes. Example:

```
(*$I 'include.dat' *)     -- in main file
(*$I 'include2.dat' *)    -- in an included file
```

10.2. THE OPTION STATEMENT

An LPL model can also contain *option* statements. They define various directives to control behavior of the LPL compiler, solver or other part of LPL. The syntax is:

```
OPTION <ident> := <value> ;
```

`OPTION` is a keyword, <ident> is an identifier and <value> is the value (numeric or alphanumeric) to be assigned to the option. The following options are implemented:

```
OPTION binding := - (+)     (off/ON)    (index binding)
OPTION unitCheck := + (-)   (ON/off)    (Unit checking)
OPTION solver := <string> | Id          (Solver parameter interface)
OPTION solver_options := <string>       (additional solver options)
OPTION solver_list := <string>          (a list of solvernames)
OPTION path := <string>                 (directory paths)
OPTION workingDir := <string>           (working directory)
OPTION randSeed := <number>             (sets the random seed)
```

```
OPTION formatMask "<string>"          (defines a mask for a WRITE)
OPTION db := <string>                 (a db connection string)
```

10.2.1. APPLIED INDEX-LIST ON AND OFF WHILE PARSING

Applied index-list can be dropped in an LPL model. Hence, expressions such as "SUM(i) a" and "SUM(i) a[i]" are both equivalent, if *a* is declared over *i*. By default, both notations are legal. If, however, the parse option

```
OPTION binding := '+';
```

is used, then the modeler cannot drop the applied index-list and the parser checks of whether an applied indexlist was dropped, if so, a parse error is generated. The instruction

```
OPTION binding := '-';
```

makes dropping again possible. By default applied index-lists can be dropped. The main purpose of this directive is model debugging (see also binding).

10.2.2. UNIT CHECK ON AND OFF WHILE PARSING

Normally, the commensurability of units within expression is checked while parsing the model. By default the parser runs with the unit check option ON:

```
OPTION unitCheck := '+' ;
```

There may be reasons that the model, or parts of it, must be parsed without unit check. The user can then remove unit checking by:

```
OPTION unitCheck := '-' ;
```

10.2.3. THE SOLVER INTERFACE PARAMETERS (SIP)

LPL has a flexible and transparent solver interface. The communication between LPL and an solver is basically determinate by 23 *solver interface parameters* (SIP) (explained below) placed in the single *solver parameter string* (SPS). For later reference, we call these parameters SIP1, SIP2, ..., and SIP22). The SIPs are separated by commas within the SPS. They are defined in an OPTION statement as following:

```
OPTION solver := 'solver parameter string';  (* the SPS *)
```

That is:

```
OPTION solver := 'SIP1,SIP2,...,SIP19,SIP20';
```

The SPS in the OPTION statement can also be an identifier in which case the identifier must have been defined before as a solver. Example:

```
OPTION solver := cplex;
```

In this case, a solver option

```
OPTION solver ALIAS cplex := '......'; -- the SPS for solver Cplex
```

must have been defined before. Note that several solvers can be predefined and are distinguished by the ALIAS name.

The solver interface parameters are (SIP1 to SIP23):

1. Indicates what to do before the solver is called. It is:

```
'' (empty)     nothing to do (the default)
'lpo'          the LPO-file is generated before solving
'mps'          the MPS-file is generated before solving
'lpo+mps'      both LPO- and MPS-file are generated
'equ'          the EQU-file is generated before solving
else           the string is interpreted as a program name to be
               executed as a child process (while the parent waits)
```

Example: If the SIP1 consists of the string **`lpo`**, then LPL generates the LPO-file just before it calls the solver.

2. Indicates what to do after the solver has terminated. It is:

```
'' (empty)     nothing is done (the default)
'sol'          the SOL-file and the DUA-file are read after solving
'lpx'          the LPX-file is read after solving
```

Example: If the SIP2 contsists of the string **`sol`**, then LPL reads the SOL-file just after the solver terminates.

3. The program or the library that is called as solver. It is:

```
'lpl-lp'       LPL's own LP solver is called (the default)
'perm'         LPL's internal permutation-heuristic is called
'cplex*.dll'   The dynamic link library of CPLEX is called
'cplex6*.dll'  The library of CPLEX65 or CPLEX60 is called
'mops*.dll'    The dynamic link library of MOPS solver is called
'mosek*.dll'   The dynamic link library of MOSEK solver is called
'xpress'       The dynamic link library of Xpress solver is called
'xa'           The dynamic link library of XA solver is called
'loqo'         The dynamic link library of the Loqo solver is
               called
'conopt'       The dynamic link library of the ConOpt non-linear
               solver is called
'cfsqp'        The dynamic link library of cfsqp solver is called
'OptQuest'     The dynamic link library of OptQuest solver is
               called
'Internet'     The model is sent to an Internet server
'' or 'nosolver'     nothing is done (no solver is called)
else           the program specified by the parameter is called
```

4. The name of the *solver options mapping file* (SOMF) (explained below),
5. The name of the *solver parameter file (*SPF) (explained below),
6. The content of the solver parameter file (the solver options), the different options are separated by a newline char (\n).

7. The string replacing '%3' within the SIP6, if the model has to be maximized,
8. The string replacing '%3' within the SIP6, if the model has to be minimized,
9. The filename from which LPL has to read the solution (SOL-file),
10. An integer indicating on which physical position on a line in the SOL-file the first character of the variable name is found,
11. An integer indicating on which physical position on a line in the SOL-file the first digit of the value is found,
12. An integer indicating the length of the numerical value of the variable in the solution file,
13. The filename from which LPL has to read the dual values (DUA-file),
14. An integer indicating on which physical position on a line in the DUA-file the first digit of the dual value is found,
15. A substring searched in the SOL-file to indicate that the model is optimal,
16. A substring searched in the SOL-file to indicate that the model is infeasible,
17. A substring searched in the SOL-file to indicate that the model is unbounded,
18. A substring containing the model types (see §7.3), separated by colons, that this solver can solve. An empty substring means that the solver has no restriction.
19. The string replacing '%4' within the SIP6, if the model is a MIP (actually used only in Xpress).
20. The string 'd' or 'dh' or empty. An empty string means that LPL will not generate derivatives or the Hessian, 'd' means to generate derivatives only, 'dh' means to generate derivatives and the Hessian.
21. A solver dependent parameter: (Used actually for Cplex as wait-loop number).
22. A solver dependent parameter: (waiting time in loop in secs).
23. A solver dependent parameter: (display time interval in milli-secs of callbacks)

All parameters (except parameter 8 and 9) may contain the following strings:

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

'%2' : which is substituted by the objective function name

'%3' : which is substituted by SIP7 or SIP8, depending whether the model is to be maximized or minimized.

'%4' : which is substituted by SIP19, if the problem is a MIP-problem.

'%%' : which is substituted by the solver options string defined in a OPTION solver_options := '.....'; statement.

Any other characters or strings are taken literally. (Note that a backslash initialize a non-printable character, see §3.7.)

The parameters 9 to 17 are used to read the solution back to the LPL system. They suppose that the solver writes the solution to a SOL-file and DUA-file. They must contain one variable name per line with its value. All lines not containing a variable name in positions specified in SIP10 is ignored. If a substring specified in SIP15, SIP16, SIP17 is found in the SOL-file, this is interpreted as a solver status (optimal, infeasible, or unbounded).

Examples for SPS for several solvers can be found in the file **`lplcfg.lpl`**.

COMMUNICATION BETWEEN LPL AND A SOLVER

When LPL runs a *solve* statement, the following procedure is called:

1. All constraints are generated (a model instance is created) and the problem type (see §7.3) is automatically detected by LPL.
2. If the problem type is NONE then the procedure exits and no solver is called.
3. Next the SIP1 is processed.
4. The SOL- and DUA-files are erased if they exist and the values of all variables are set to zero.
5. Next the solver options SIP6 are processed, that is, the parameters that are passed to the solver.
    1. all substrings '%0' ... '%3' within SIP6 are substituted as explained above,
    2. SIP6 is merged with the *solver option string* (SOS) define in a "OPTION solver_options := '...SOS...';". If the SIP6 contains the substring '%%', it is substituted by the SOS, otherwise the SOS is appended to the SIP6. (We call the resulting string eSIP6).
    3. SIP4 (if not empty) is interpreted as a filename (the *solver options mapping file* (SOMF)) and all solver options in the eSIP6 are substituted by a corresponding entry in the SOMF file (see below the SOMF file),
    4. SIP5 (if not empty) is interpreted as a filename (*solver parameter file (*SPF)) and the modified eSIP6 string is written to this file.
6. The solver is called as specified by the SIP3.
7. The SIP2 is processed.

Example 1: Suppose a model contains the two following OPTION statements:

```
OPTION solver := cpl;
OPTION solver_options := 'timelimit 60';
```

The solver cpl must have been defined before (typically in the file `lplcfg.lpl`) as following:

```
OPTION solver ALIAS cpl := ',,cplex65.dll,cplex.prm';
```

Note that all SIPs are empty except the third and the fourth for this solver. The model is handed over to the solver directly within the memory. LPL does this automatically. The solution is retrieved by LPL directly within the memory also. SIP3 says to call the DLL library of *cplex65.dll* (www.ilog.com) as solver in step 6 above. SIP4 is needed to substitute the solver options (see below, the SOMF) to work with the library.

Example 2: Suppose a model contains the two following OPTION statements:

```
OPTION solver := mopsSol;
OPTION solver_options := 'XMXLPT=1\nXMXMIN=1\n';
```

The solver MOPS (www.mops.fu-berlin.de) must have been defined before (typically in the file **`lplcfg.lpl`**) as following:

```
OPTION solver ALIAS mopsSol  := 'mps,sol,mops.exe,,XMOPS.PRO,\
XFNMPS=\'%1.mps\'\nXMINMX=%3\nXFNLPS=\'%1.sol\'\nXFNIPS=\'%1.sol\'\n\
XOUTLV=3\n,\'max\',\'min\'\
,%1.sol,12,25,15,%1.sol,89,solution,infeas,unbound,LS:iLS:LP:MIP';
```

SIP1 is 'mps'; hence, at step 3 the MPS-file is created. In step 5 the solver options are processed. Suppose the model name is `xyz.lpl`, it is to be maximized, and the objective name is `obj` then the SIP6 will be translated into the string:

```
XFNMPS='xyz.mps'\nXMINMX=max\nXFNLPS='xyz.sol'\nXFNIPS='xyz.sol'\nXOUTLV=
3\n
```

Next it is merged with the SOS string':

```
XMXLPT=1\nXMXMIN=1\n
```

SIP4 is empty, hence this string is left unchanged. SIP5 is ' XMOPS.PRO '. Hence the default configuration file XMOPS.PRO (the solver parameter file (SPF)) is created with the content of:

```
XFNMPS='xyz.mps'
XMINMX='max'
XFNLPS='xyz.sol'
XFNIPS='xyz.sol'
XOUTLV=3
XMXLPT=1
XMXMIN=1
```

These are the solver parameters that will be read from the MOPS solver. Next in step 6, the solver specified in SIP3 is called, that is, the program `mops.exe` is executed. Next in step 7, SIP2 is 'sol'. Therefore, the file specified in SIP10 is read as a SOL-file and the file specified in SIP13 is read as a DUA-file.

THE-FILE

The *solver option mapping file* can be used to substitute solver options for any specified solver. An example is the file CPLEX11.PRM used for the CPLEX11 dynamic library. As an example, the option “tilim 60” will be substituted with “1039

d 60". This later will be interpreted by the LPL to CPLEX interface. The corresponding line in the CPLEX11.PRM file is:

```
TILIM                1039 d  Global time limit
```

The line is separated into four parts from left to right: (1) the name as used in LPL's eSIP6 (option solver part—see above) (here: TILIM), (2) the substitute (here: 1039), (3) an attribute (here: 'd'), this is used in the LPL to CPLEX interface to call the right routine: possible values are 'i' (for integer values), 'd' for doubles, 's' for string values). (4) the rest of the line, which is a comment for the user only. The four parts must be separated by at least one blank. The first part is not case-sensitive.

The above entry will call CPLEX's routine *CPXsetdblparam(env,1039,60)*. For more detail see the CPLEX library and the CPLEX parameter manuals.

The SOMF-file can also be used for other solvers (for example *xpress.prm* for the Xpress solver).

THE SOLVER STATUS

When a solver was called, LPL will return the status of the problem as:

```
0   'NOT SOLVED'    no solver was called
1   'UNBOUNDED'     the problem is unbounded
2   'INFEASIBLE'    the problem is infeasible
3   'ABORTED'       the solver was aborted
4   'TROUBLES'      the solver had problems to solve the problem
5   'HEURISTIC'     the solution is not necessary optimal
6   'NORMAL'        the solver terminated normally (not necessary
                    with an optimal solution)
7   'OPTIMAL'       an optimal solution was found
```

The solver status is communicated to LPL by the means of several solver interface parameters (SIP15-17). The solver status (as number 0-7) can also be returned from LPL by the expression within the model:

```
<ModelName>.stat ;
```

where <ModelName> is the model name that has been solved.

THE HEURISTIC SOLVER

LPL comes with an integrated heuristic solver for certain problems (PERM), called *permutation problems*. A *permutation problem* is defined as following: Let π be a vector of the n numbers in the range [1,n]. The objective is to find the permutation(s) π_i over all $n!$ permutations which minimizes a certain function f:

$$\min_{\pi} f(\pi_i)$$

This problem has many applications. Four solvers are directly integrated: (1) a heuristic based on the tabu search method, (2) a heuristic based on local search; (3) a random search solver; and (4) a full enumeration solver. The tabu search solver is

useful in finding good solution to many problems. The local search solver looks in the neighborhood repeatedly until it finds a local minimum and then stops. It has been shown to be useful for some problems. The random search solver is not a solver to use when a good solution is to be searched. It is rather to analyze the problems. It generates a sample of permutation and calculates the objective function for each permutation. It gathers the minimum, the maximum found and returns the means as well as the standard deviation (in the LOG-file). The full enumeration solver enumerates all permutations and returns the optimum. Needless to say, that this is only useful for very small problems.

The solvers are invoked by one of the following lines:

```
OPTION solver := tabuSol;
OPTION solver := randSol;
OPTION solver := locaSol;
OPTION solver := enumSol;
```

These options should only be used for permutation problems. LPL recognizes automatically, when a model is a permutation problem. Therefore, one must communicate this within the LPL model by one of the four options. The modeler, however, should be careful not to use the heuristic solver for problems that are not permutation problems.

The user can configure, in a limited way, these four solvers through the SIP6. For example, for the TABU solver the SIP6 is:

```
0:60:30000:17:100:20:8:1:1
```

These are 9 numeric parameters separated by colons are passed to the heuristic solver. The nine parameters are:

- Number indicating which solver to use (0=tabu, 1=rand, 2=local, 4=enum),
- Time limit in seconds (60),
- Number of maximal iterations (30'000)
- Length of TABU list (17) (only for tabu)
- The number of iterations at the beginning after which the search of the neighborhood is switched to an exhaustive search of all $O(n^2)$ neighbors (intensive search) (100) (only tabu)
- Switch to a slightly randomized solution if the solution does not improve after this number of iterations (20) (only tabu)
- The number of random exchanges to generate the slightly randomized solution of the last parameter (8)
- Indicate whether to use a randomized solution to start the search or not (0=not randomized, 1=randomized) (1),

- Indicates whether to be mute or to show immediately whether an improved solution has been found (0=to be mute, 1=yes write them).

10.2.4. DIRECTORIES AND FILE PATHS

The paths to all directories used in LPL are collected in the *directory list*. All files (even DLLs) are searched in this list of directories and in the following order. The paths in the list are separated by semicolons.

The *directory list* is built (in this order) from

1 The *working directory* or the *current directory*,
2 The *directory of the executable*,
3 The *lplpath directory list*,
4 The *user-defined directory list*.

(The directory list is displayed in the <LPL : Options> window of the Windows version of LPL or at the beginning of a compilation when the compiler switch 'ww' is used.)

The **working directory** is normally the directory specified in the first program parameter where the model file is stored. If no model file is specified, it is the directory of the executable or the current directory from where the executable was launched. The working directory can be changed within LPL using the option

```
OPTION workingDir := '<workingDirectoryName>';
```

(In the Windows version the working directory is also changed through a open file dialog box; however the SaveAs dialog does not change it.) All intermediary files generated by LPL are saved in the working directory (except the files in the WRITE statement specified by their own path).

The **directory of the executable** is the directory where the executable (lplw.exe or lplc.exe or the application which uses the lpl.dll) is located.

The **lplpath directories** is set by the environment variable 'LPLPATH'. Add the line:

```
set LPLPATH := <my lplpath directory list>.
```

to the autoexec.bat file and reboot the machine; or (in Windows 2000/XP) add interactively the environment variable.

The **user-defined directories** are specified by the option

```
OPTION path := '<List of directories separated by ';'>'
```

Example (note that the backslash is an escape character):

```
OPTION path := 'c:/lpl;c:/lpl/models;c:/solver/cplex';
```

The following indicate how LPL looks for files:

- The file lplStat.txt are only written to the directory of the executable.
- All other files are read and searched in the directory list (in the order of the list), and they are written to the working directory by default.

10.2.5. RANDOM NUMBER INITIALIZATION

The built-in random generator can be initialized with this option:

```
OPTION randomSeed := 1;     -- sets the random seed to 1
```

10.2.6. FORMAT MASKS FOR WRITES

The option

```
OPTION formatmask ALIAS mask1 ' <content of the mask> ';
```

defines a format mask for one or several subsequent WRITE statements. In a subsequent WRITE statement one then can use this mask as:

```
WRITE '@mask1' : .... ;
```

If the mask in a WRITE statement begins with a @ character then it points to a format mask definition and takes that mask.

10.2.7. DATABASE CONNECTION STRING

This option allows the modeler to specify a *connection string* that is necessary to connect to the specific database system. The connection string specifies the information needed to connect to a database. Examples are given in the file **`lplcfg.lpl`**.

10.3. THE FILE *LPLCFG.LPL*

If the file **`lplcfg.lpl`** exists in the LPL-path, then this file is compiled *before* the model file. The file **`lplcfg.lpl`** *is treated as an include-file after the MODEL declaration at the very top of each model file*. The user can define different options (for example solver interface parameters) which are available in every model.

10.4. COMPILER SWITCHES

The LPL compiler can be called with three parameters. The first is a filename (a model in LPL syntax). The extension of the filename must be *lpl*, but is not needed when launching LPL. The second parameter is an optional string of at most 8

characters as follows to instruct LPL how to compile – the compiler switches. (The third is explained in the next section, §10.5).

```
lplc <modelfile> [CompilerSwitches] [APL]
```

CompilerSwitches can be empty or can contain any characters. The following characters, however, modify the default behavior of the compiler:

```
'a' : The NOM-files are never overwritten, but they are appended.
'c' : The solver is not called.
'd' : Debugging information is written to the BUG-file.
'dd': more info to the BUG-file
'e' : Generates the EQU-file
'ee': Generates the EQU-file with long names (eventually)
'eee': Generates sparse EQU-file (only variables with value<>0)
'f' : Create a comment file (modelfile.eng) after parsing
'g' : Given a model with randomized data (RND()), run it for a
              number of times with different random initialization
'h' : No WRITE statement is executed
'i' : Generates the INT-file
'j' : Generates the INT1-file
'l' : Generates the LPO-file
'm' : Generates the MPS-file
'p' : Generates slack variables for all constraints while parsing,
      Adds a new minimizing function: minimize all slacks.
'q' : Generates a SQL-script and a (LPL) MODEL file.
'r' : Only run the model (do not parse it anymore)
's' : Only parse the model, but do not run it
'ss': Only parse, strips off all comments, stores a new file
'sss': Only parse, strips off all comments, stores a new file, encrypt
't' : Generates a partial LATEX-file
'T' : Generates a complete LATEX-file
'u' : The constraints are not generated and a solver is not called
'v' : Write output to file lplStat.txt.
'vv': Write output also to file lpllog.txt.
'w' : Generates more output during compilation.
'ww': Generates even more output during compilation.
'x' : The configuration file lplcfg.lpl is ignored while parsing.
'y' : <not documented>
'z' : Compile all LPL models in a given directory (only lplc.exe)
'1','7' : Level of translation of logical constraints into 0-1
              constraints.
any other character does not modify the default behaviour.
```

The order of the characters does not matter. Note, however, that there are interdependencies between the switches. The switches ‘ss’ and ‘sss’ need to be explained further: Using the switch ‘ss’, parses the model, strips all comments out and stores the model source in the LPL-file then exits.

The switch ‘sss’, in addition of stripping all comments, replaces all identifiers in a way that the model cannot be read anymore by a user, but LPL still can run it. It is made to hide the knowledge modeled in the model to others. At the same time a CRP-file is generated (a file with the same name as the model but with extension `.crp`), which maps the real names with the encrypted names.

The switch ‘p’ is interesting when a model is infeasible. Running it with this switch will reveil the infeasibilities eventually.

10.5. THE ASSIGNED PARAMETER LIST (APL)

One can call LPL (lplc.exe, lpls.exe, lplw.exe) with a third parameter: the assigned parameter list (APL). The parameter list must have the following format:

```
ID=value,ID=value, ...
```

ID can be any non-indexed PARAMETER (or STRING PARAMETER) name within a LPL model or it can be one of the strings '**solver**', '**solver_options**', '**@VAR**', '**@RL**', '**@DOC**' , '**@ID**', '**@IP**', or '**@IP1**'.

- If *ID* is a model parameter defined in the model, its value is assigned exactly in the same way, as if it where defined within the model. For example, if within the model *xx.lpl* a parameter *aa* is declared as:

  ```
  PARAMETER aa;   -- with no value assigned
  ```

  and one calls the LPL compiler with

  ```
  lplc xx.lpl - aa=10
  ```

  then the parameter within the model gets the value 10 after a parse of the model *xx.lpl*. Several parameters can be assigned in this way. The assigned parameter list can also be set with the `lpl.dll`, just call it as: **LPLsetP(12,'aa=10')**.
- If *ID* is '**solver**' then the corresponding solver will be used to solve the problem. If, for example, one uses

  ```
  lplc xx.lpl - solver=glpkSol
  ```

  then the glpkSol will be used. This supposes that 'glpkSol' was defined within `lplcfg.lpl` or the model.
- If *ID* is '**solver_options**' then the corresponding solver options will be used to solve the problem. If, for example, one uses

  ```
  lplc xx.lpl – "solver=cplex,solver_options=timelimit 10"
  ```

  then the solver cplex will be used with a time limit of 10 secs. This supposes that 'cplex' was defined as a solver within `lplcfg.lpl` or the model as well as the file *cplex.prm* is available (see SOMF-file).
- If *ID* is '**@VAR**' then a list of submodels are executed at the different variant points of execution. The *value* must begin with a variant name, followed by submodel names in parentheses. The submodel names must be separated by a `+´. If more than one variant is defined then they must be separated by a semicolon. The variant name can be omitted, in which case the submodels are executed at the *default variant point*. Example:

  ```
  lplc xx.lpl - @VAR=va1(model1+model2);va2(model3+model4);(model6)
  ```

When LPL is called with the previous APL parameter, then at the variant point **va1** the models **model1** and **model2** are executed, at the variant point **va2** the models **model3** and **model4** are executed, and at the default variant point the model **model6** is executed.

In any case, without a default variant parameter, a submodel called **data** is executed at the default variant point if it exists.

- If *ID* is '**@RL**' (RandomLoop) is used together with the compiler switch 'g'. One can choose the number of randomized solutions. For example the call to:

```
lplc xx.lpl g @RL=23
```

  This command will run the model for 23 times each time with another random seed. This supposes also that some data within the model are generated randomly.
- If *ID* is '**@DOC**' then *value* must be a filename, (If the filename begins with a '*' then this character will be replaced by the LPL model filename). The file must contain the documentation part of the model (see 11.6). In this case the actual documentation will be replaced with the documentation in this file after a parse. In this way, one can generate multiple language documentations for a model.
- (The three strings '**@ID**', '**@IP**', or '**@IP1**' are for internal use only.)

10.6. MODEL DOCUMENTATION

The compiler switch 't' or 'T' will generate a documentation file in LaTeX (called TEX-file): 't' generates a partial LaTeX file that can be included into another LaTeX document using TeX's `\include{…}` command, and 'T' generates a standalone LaTeX file that can be translated into a PDF or HTML document using free software.

The model documentation in the LPL source code is part of the model source code. It consists of *comment attributes* and *documentation comments*.

1. *Comment attributes* are strings within quotes (**"……"**) and have already been explained in chapter 4.
2. *Documentation comments* are text of an unlimited number of lines enclosed within **(**……**)** in the model code. Leading spaces and ***** (a star sign) on all lines within the documentation comment are ignored. For example:

```
(** The purpose of this model component is
  * to optimize the profit. The variable
  * is the quantity of the product … .
  *)
MODEL OptProfit;
  ……
```

  A documentation comment can be attached to every entity. It is normally placed in the source code *before* the formal declaration of the entity (as in the example

above: *before* **`MODEL OptProfit …`**). It can also be placed after if no other entity follows. This allows one to place the eventually large documentation comment for the main model *after* the formal model. Each documentation comment can contain any LaTeX specification and commands (that make sense in a particular context) because the model documentation is automatically processed by the TeX/LaTeX typsetting system.

The documentation comment in LPL may also contain one the three following @-specifications.

1. `@Here is text@` : The part "**Here is text**" will be typesetted as is in a type-font style, typically as: `Here is text`. The user must care to place a beginning @ and an ending @. The documentation tool in LPL translates

   `@Here is text@`

   into the verbatim LaTeX code:

   `\verb?Here is text?`
2. **`@.`** (@ and a dot , beginning on a new line): This translates into an enumerated list. The list must end with an empty line.
3. **`@@`** (two @'s on a new line): Includes the formal model in a verbatim and shaded environment.
4. **`@§`** (@ and a '§'): adds the whole model in Math-Mode.
5. **`@!`** (@ and a exclamation mark) adds the LaTeX commands
   `\begin{shaded}{\small\begin{verbatim}` or
   `\end{verbatim}}\end{shaded}` alternatively.

(All specifications could also be realized – of course – by using the appropriate LaTeX commands.)

The translation tool from LPL to a LaTeX code also uses a list of predefined TeX and LaTeX definitions, which are found in the file, called **`lpl.tex`**, distributed with all lpl packages.

To generate a model documentation proceed as follows:

1. Software needed: You'll need LPL and a free LaTeX distribution (p.e. from www.miktex.org , download the minimal package from the Miktex Web-site and install it).
2. Write the documentation in the LPL-code.
3. Generate the TEX-file using lpl's command switch 'T' (where **`mymodel.lpl`** is the LPL model file:

```
lplc mymodel T
```

4. Translate the resulting TEX-file (here `mymodel.tex`) using `pdflatex` in the Miktex distribution. This program is by default installed in directory **`c:\texmf\miktex\bin`**. Hence execute

```
c:\texmf\miktex\bin\pdflatex mymodel.tex
```

5. The result is a PDF-file, that is called **`mymodel.pdf`** in our case.

The whole documentation can also be in a separate file. The file can be read automatically after a parse by the @DOC APL-parameter. The structure of the file is as follows:

1. Each *comment attribute* begins with the two character **##** followed by the entity name to which it belongs. Then follows a blank and the comment itself. For example, if the parameter a is defined as:

```
parameter a "Comment to this parameter";
```

   then the file would contain a line as follows

```
##a Comment to this parameter
```

2. Each documentation comment begins with the two characters **#&** and followed by the entity name to which it belongs. Then follows a blank and the comment itself.
3. Each formatting string in a Write statement begins with the two characters **#!** and followed by the entity name to which it belongs (the entity name is *'W'+nr* , where *nr* is the number of the Write statement. Then follows a blank and the comment itself.
4.

Each comment in the file can contain multiple line the comment ends when an ## , an #& , an #! or an eof occurs. The last newline characters are removed from the comment content.

10.7. CALLABLE FUNCTIONS FROM LIBRARIES

The external-proc-call instructions make a certain number of external procedures available within an LPL model from two libraries: The **`Sys`** and the **`Draw`** library.

10.7.1. THE SYS LIBRARY

The Sys library contains the following function:

```
Sys.Call('notepad.exe file.txt'); -- calls an external program
```

(More library functions will be added soon.)

10.7.2. THE DRAW LIBRARY

The Draw library allows one to draw within LPL. The call of these functions is as follows:

```
Draw.Line(x,y,x1,y1[,c2,w]);       draw an line (default c2=0, w=1)
Draw.Rect(x,y,x1,y1,c1,[c2,w]);    draw a rectangle (def: c2=c1, w=1)
Draw.Ellipse(x,y,x1,y1,c1,[c2,w]);draw an ellipse (def: c2=c1, w=1)
Draw.Ratio(x,y);                   defines the x- and y-ratio (zoom)
Draw.Ratio(x,y,y1);                defines also bottom-left as origin
Draw.Text(t,x,y[,a,h],c3);         write a text to the picture
                                   (default: a=0, h=12)
Draw.setFont(t,h);                 sets the font an dits height
Draw.Density(x,y,x1,y1,p,n,c);     draw a density function of p
Draw.Back(f);                      load and draw a background picture
                                   stored as JPEG- or BMP-file.
Draw.SaveJPEG(f);                  save a JPEG-file of the drawing
                                   space and clear the drawing space
```

The tuple (x,y) gives the coordinate of the top-left of a box, (x1,y1) the bottom-right of the box, c1, c2, and c3 are color numbers (c1 is the fill color and c2 is the pen color and c3 is the font color), w is a pen width, t is a text of up to 255 chars, a is a number for the angle, h is the font height, f is a filename, p is a parameter (variable) defined in the model. The parameters within [and] are optional.

All instructions draw on a single drawing space of 2000x2000 pixels with the top-left point as the origin. The default ratio is (40,40) [`Draw.Ratio(40,40)`]. This means that any (x,y) point is interpreted as (40*x,40*y) pixels. Example:

```
Draw.Line(1,1,2,3,0,1);
```

This instruction draws a line from pixel point (40,40) to (80,120) with color 0 and pen with 1. If a third parameter *y1* in `Draw.Ratio` is used, then the origin is put at the bottom-left point (0,y1) (with y1<=2000) and the space to draw on is (0,0) to (2000,y1). The color are specified as numbers. They can be attributed through the function **`RGB(r,g,b)`**, which assigns a color in the RGB (red,green,blue) color space (with $255 \geq r,g,b \geq 0$). Hence, **`RGB(0,0,0)`** means black (no color), and **`RGB(255,255,255)`** means white (saturated all three colors), and **`RGB(255,0,0)`** means red (red only saturated).

Another way to assign the color is by any positive number or any expression that result in a positive number. The color numbers are assigned as follows:

Grey gradient:	32 to 63
Red gradient:	64 to 91
Green gradient:	92 to 127
Blue gradient:	128 to 159
Yellow gradient:	160 to 192
Magenta gradient:	193 to 223
Cyan gradient:	224 to 255

The numbers 0 to 31 are reserved for a list of saturated colors. Model examples to study the use of LPL's drawing library are **`tutor21.lpl`** and **`tutor22.lpl`**.

10.8. MODEL FILE ENCRYPTION

The LPL files (model files) can be encrypted. The LPL compiler automatically recognizes if a file is encrypted and takes the necessary steps. The user does not intervene in any case. The user can only encrypt a file or decrypt a file using the local popup menu in the *lplw.exe* browser's editor. To encrypt and to decrypt, the user must enter a key (password). Each file in a model project can be encrypted separately and with a different key and can only be decrypted with the same key.

10.9. UNDOCUMENTED FEATURES OF LPL

Undocumented features are extension of LPL on an experimental basis. That is, they can be removed or modified at any time.

1. (LPL4.43h) The $ operator to work with very large data cubes: An example is given by the following model:

```
(* undocumented feature in LPL the $ attribute and the $ function *)

MODEL test;
  SET i := /1:10/; j := /1:5/;
    ij{i,j} := /1 2 , 2 3, 3 4, 4 5 /;
    newij : 1..#ij  "make a basic set out of an indexed set";

  PARAMETER x{ij} := ij; y{newij} := 10*newij;
    a{i,j} := y[$ij];  -- use of $

END

Explain:
The cardinality of i is 10, of j is 5, and of ij is 4.
But the Cartesian product of ij is 50. That is, each element of ij
is mapped to the space from 1 to 50. The four elements (1,2), (2,3),
(3,4) and (4,5) are mapped to the four numbers 2, 8, 14, and 20.
It corresponds to the lexicographic ordering of the four elements in ij.
This can be seen, if you look at the parameter x.
Certainly, x also only contains four elements, but still they are mapped
to the space from 1 to 50.
By introducing a new set newij - with the same cardinality than ij -- one
introduces a new basic set of the same cardinality as ij (four). The
parameter y now, is mapped into the integer space from 1 to 4 (not from 1
to 50, like x).

To make the correspondence between the two sets (ij and newij),
$ can be used as an unary function in an expression,
which makes the transformation between the two mappings (see parameter a).
```

2. Compiler switch 'k' has the effect that SETs are read at once and cannot be extended later. This allows one to read a subset on primary keys of a database without modifying all the READ statements, for example. Suppose the primary key ID of a table contains 10 elements (ten records). An READ only reads 5 records (by a SELECT statement for example), then all derived tables containing the foreign key ID only read the corresponding records containing the 5 elements. No SELECT is

necessary. (For an example see **tutor18.lpl**, run the model with and without the OPTION 'k' and compare.)

3. You can use a parameter in the syntax as follows:

```
PARAMETER M := 10;
SET i : 1..M;
```

Use this only for a single definition of **i**.

4. The keyword ADDM (a function) adds an element to a set at run-time. Use it as:

```
SET i;
……
i := ADDM('1');
……
```

5. **Option compilerSwitch:='wwvv';** sets the compilerSwitches. It is supposed that the user knows what he does! Use with care!

6. Read a range from a EXCEL sheet: One can read a range from an excel sheet as follows:

read from 'xls:file.xls,[sheet1$A1:Z10]' : a;

The file specification must begin with 'xls:' followed by the excel file name. The 'table' is a worksheet name followed by a '$' then follows a upperleft lower right cell. The data are read cell by cell from left to right and top to bottom and assigned to a paramenter 'a' , a set a ect. in a lecicographic order. The modeler is responsible of setting right the dimesions (sets).
This feature is good for quick and dirty reading from Excel.

11. THE LPL RUNTIME LIBRARY

The LPL package comes with a 32-bit DLL (Dynamic Link Library) called **lpl.dll**:
It can only be used under Windows. The DLL allows the user to integrate the complete LPL functionalty into an application written in another language, for example C++. This chapter is an overview on how to use it and gives examples. It is supposed that the reader is familiar with the DLL-programming under Windows or has at least some basic knowledge.

The library exports procedures to access and modify LPL internal structures. It supposes that an LPL-file (a source code written in LPL) has been produced as text file (using a text editor). The package can generate different output-files, for example the LPO-file. It calls automatically a configured solver (see the file `lplcfg.lpl`) and finally, writes the result to a NOM-file, just like the `lplc.exe` or the `lpls.exe` can do. First the exported functions are listed and than several examples are given.

11.1 EXPORTED PROCEDURES

The dynamic link library **lpl.dll** exports the following procedures:

```
procedure LPLsetCallbacks(c:TProc; w:TProcC); stdcall; export;
procedure LPLinitParam; stdcall; export;
procedure LPLinit(Fn:pChar; maxt,maxa,maxr:integer); stdcall;
              export;
function  LPLcompile(opt:pChar):integer; stdcall; export;
function  LPLcompileWithCallbacks(opt:pChar; c:TProc;
              w:TProcC):integer; stdcall; export;
procedure LPLfree; stdcall; export;
function  LPLwhere:integer; stdcall; export;
procedure LPLgetErrMessage(n:integer; var msg:PChar); stdcall;
              export;
function  LPLgetError:integer; stdcall; export;
procedure LPLsetError(n:integer); stdcall; export;
procedure LPLgetP(which:integer; var sP:pChar); stdcall; export;
procedure LPLsetP(which:integer;sP:pChar); stdcall; export;
function  LPLsolve(opt:pChar):integer; stdcall; export;
procedure LPLsaveSnapshot(sP1:pChar); stdcall; export;
procedure LPLloadSnapshot(sP1:pChar; add:integer); stdcall; export;
procedure LPLgenACCESSdb; stdcall; export;

function  LPLgetHandle(name:pChar):LPLhandle; stdcall; export;
function  LPLgetFocusHandle:LPLhandle; stdcall; export;
procedure LPLsetFocus(h:LPLhandle); stdcall; export;
function  LPLisFocus:integer; stdcall; export;
procedure LPLsetFirstEntity(genus:integer); stdcall; export;
procedure LPLnextEntity(genus:integer); stdcall; export;
procedure LPLgetElementList(n,enam:integer; var sP:pChar); stdcall;
              export;
procedure LPLgetAttr(attr:integer; var sP: pChar); stdcall; export;
function  LPLgetGenus:integer; stdcall; export;
```

```
procedure LPLPivotSetParam(which,n:integer); stdcall; export;
function  LPLPivotGetParam(which:integer):integer; stdcall; export;
procedure LPLPivotInit; stdcall; export;
function  LPLPivotX:integer; stdcall; export;
function  LPLPivotY:integer; stdcall; export;
function  LPLPivotData:pChar; stdcall; export;
procedure LPLsetSelect(n,what:integer); stdcall; export;

function  LPLgetValue(n,attr:integer):TREAL; stdcall; export;
procedure LPLgetValueS(n,attr:integer; var sP:pChar); stdcall;
              export;
function LPLgetM(attr:integer):TREAL; stdcall; export;
function LPLgetDependenyFrom(n:integer):LPLhandle; stdcall; export;
function LPLgetDependenyTo(n:integer):LPLhandle; stdcall; export;
function LPLgetDependenyKind(n:integer):char; stdcall; export;
```

They are exported as follows:

```
exports
  LPLsetCallbacks,          -- not in lplj.dll
  LPLcompileWithCallbacks,  -- only in lplj.dll
  LPLinitParam,
  LPLinit,
  LPLcompile,
  LPLfree,
  LPLwhere,
  LPLgetErrMessage,
  LPLgetError,
  LPLsetError,
  LPLgetP,
  LPLsetP,
  LPLsolve,
  LPLsaveSnapshot,
  LPLloadSnapshot,
  LPLgenACCESSdb,

  LPLgetHandle,
  LPLgetFocusHandle,
  LPLsetFocus,
  LPLisFocus,
  LPLsetFirstEntity,
  LPLnextEntity,
  LPLgetElementList,
  LPLgetAttr,
  LPLgetGenus,
  LPLPivotGetParam,
  LPLPivotSetParam,
  LPLPivotInit,
  LPLPivotX,
  LPLPivotY,
  LPLPivotData,
  LPLsetSelect,
  LPLgetValue,
  LPLgetValueS,
  LPLgetM,
  LPLgetDependencyFrom,
  LPLgetDependencyTo,
  LPLgetDependencyKind;
```

The following types are defined as:

```
integer       4 bytes
pChar         pointer to a null-terminated string
TProc         pointer to a parameterless procedure (4 bytes)
TProcW        procedure (s:pChar); stdcall;
LPLhandle     as 4-byte integer  (zero for nil pointer)
```

```
char          1 byte char
```

Certain procedures must be called in a specific order. *LPLinit* must be called first to allocate the memory and to initialize all variables. Next one may call *LPLcompile* or *LPLsolve*. Several calls of these two functions can be made in any order. If the error value at the return time of one of them is different from zero -- which means that an error occurred -- then one can call *GetErrorMessage* and *LPLgetError* to find out more about the specific error. Finally, before leaving the library, one must call *LPLfree* to free the memory again. Certain procedure (p.e. LPLgetP) returns a string, this is marked as "*var x: pChar*". In this case, the user only needs to pass a pointer to a memory location where he allocated space before. The procedure then fills the allocated memory with a zero-terminated string. It is in the clients reponsability to allocate enough space. The procedures are now explained in more details:

LPLsetCallbacks allows the user set the callback (c) and to redirect the output of the LPL messages during the compilation (w).

The procedure (parameter) *c:TProc* is LPL's *callback procedure.* It is executed once at the beginning of each statement while compiling; if *c* is nil, then a default empty callback procedure is executed (nothing is done). The user can hook her own procedure here to allow her to get the control periodically while LPL is compiling (for a typical callBack function see below at *LPLsetError*).

LPL generates many messages depending on whether the compiler switch is '', 'w' or 'ww' are set or not. All messages are handled by the callback procedure *w:TprocC* (second argument of *LPLsetCallbacks*). The user can write her own Write procedure and redirect these messages.

Example: Suppose the user writes the procedure

```
procedure MyWriteToLog(s:pChar); stdcall;
begin writeln(s); end;

procedure MyCallback; stcall; begin end; // do nothing
```

now she assigns these procedures to LPL message stream by:

```
...
LPLsetCallbacks(MyCallback,MyWriteToLog);
...
```

Then all LPL messages will be printed wherever *writeln* writes. The *MyWriteToLog* procedure, however, can be much more complicated. The messages could be written into a LOG-window (like in `lplw.exe`). `lplw.exe`, by the way, uses exactly this method. Note that this procedure is not in `lplj.dll`. Use *LPLcompileWithCallbacks* instead.

LPLinitParam handles the startup (console) parameters of the application that includes **`lpl.dll`** (model name, compiler switches) and passes them to LPL.

Suppose a program *MyProg* (using **`lpl.dll`** as part of its application) is called as:

```
MyProg test.lpl ww
```

then LPL's (1) the model name ('test.lpl') and (2) LPL's compiler switches ('ww') are overwritten by the two parameters. *LPLinitParam* should be called once only at the very beginning even before *LPLinit* .

LPLinit allocates memory and initializes the internal store of LPL. The parameters are as following:

```
Fn          is the modelname. It will overwrite a previously
            assigned modelname. A empty string does not
            overwrite the modelname.
maxt        number of bytes allocated for strings, set elements,
            and texts (default is 20000).
maxa        8*number of bytes for non-zero numerical data
            (default is 20000)
maxr        roughly the length of the LPO-file (default is
            20000)
```

LPL has an automatic allocation mechanism for memory and the user does not need to do something special. Assigning the three parameter **`maxt`**, **`maxa`** and **`maxr`** to 0 – hence calling LPLinit as: **`LPLinit(Fn,0,0,0)`** – is normally the best choice. LPL then will allocate a default amount of memory. If later in the run this turns out to be too small then LPL's memory manager executes an efficient reallocation. Nothing is to be done by the user.

The minimal values of the three parameters are reported at the end of a run in the lpllog.txt file (option wwvv). The user of `lpl.dll` could then copy these thee values into the parameters of the procedure **`LPLinit`** in order to minimize LPL's internal reallocations.

LPLcompile compiles the file defined previously by *LPLinitParam* or in the procedure *LPLinit* as model name using the compiler switches *opt*. The switches are defined in the reference manual (see Compiler Switches).

If *LPLcompile* succeeds, its return value is zero. If *LPLcompile* fails to compile and/or run the model, an error is generated and the return value is the error number. Its message can be returned by the function *LPLgetError*. This number reflects an error message in the text file **`lplmsg.txt`**. The message file must be present to get an error message.

LPLcompileWithCallBacks is the same as *LPLcompile*. However, it comes with two further parameters (the two callback parameters in *LPLsetCallbacks*). This function is only exported from the `lplj.dll`.

LPLfree frees the memory and clears the LPL store completely. It should be called once at the end. *LPLinit* and *LPLfree* should be used in pair. After a call to *LPLfree*, one can again call *LPLinit* to reallocate the memory for LPL (which should be followed by another *LPLfree* at the end).

LPLwhere returns an integer value depending on the state of the LPL compiler:

```
-1: In an error state (after an error occurred)
0 : At startup or after calling LPLfree
1 : After calling LPLinit
2 : After parsing
3 : After running (no model instance created)
4 : After running (model instance created)
6 : while parsing
7 : while running
8 : while constraints generating
9 : while solving
10 and higher: the state is:  "multiple snapshot analysis"
```

LPLgetErrMessage returns the corresponding error message in file `lplmsg.txt`, given its error number. The first parameter is the error number. It must correspond to an error number contained in the first three positions of a line within the file `lplmsg.txt`. The second parameter is the returned message.

LPLgetError returns the last error of a compilation. It is zero, if no error occurs otherwise it is an positive integer from 1 to 999. This number is interpreted as the first three digits on a line in the file `lplmsg.txt`.

LPLsetError can be used to set an error (in particular the error 599 (user abort). But this works only if no error has been occurred prior to this call. Hence, it can be called only once, any other call has no effect. The procedure LPL*setError* is useful in the callBack function (parameter *c* in the procedure *LPLsetCallbacks*). A typical callBack function for LPL is (it is used in the `lplw.exe`):

```
procedure MyCallBack;
begin
  case LPLwhere of
    0: Form.Label2.Caption:='No model';
    1: Form.Label2.Caption:='LPL kernel initialized';
    2: Form.Label2.Caption:='Model parsed';
    3: Form.Label2.Caption:='Model ran';
    4: Form.Label2.Caption:='Model ran/instance created';
    6: Form.Label2.Caption:='parsing...';
    7: Form.Label2.Caption:='running...';
```

```
      8: Form.Label2.Caption:='constraint generating...';
      9: Form.Label2.Caption:='solving...';
    end;
    SouForm.Update;

    if PeekMessage(msg,0,0,0,pm_remove) then DisPatchMessage(msg);
    if flagSet then begin LPLsetError(599); flagSet:=false; end;
  end;
```

This function must be assigned while calling *LPLsetCallbacks*, for example:

```
LPLsetCallbacks(MyCallBack,nil)
```

While compiling/running the model (using *LPLcompile*) this function then checks periodically in which state the LPL compiler is and returns a message to the *Form.Label2.Caption* label. Then it checks Windows events to be hooked on. Finally, it calls *LPLsetError* to abort the compilation, if *flagSet* is set. The Boolean *flagSet* typically is set if the end-user clicks on an ABORT-button.

LPLgetP returns different information pieces from the LPL kernel depending on the first parameter *which*. The returned information is stored in the second parameter *sP*. If *which* is the number as follows then the returns string is given as:

```
1 : the actual modelname
2 : the actual compiler switches
3 : the size of maxa (in LPLinit)
4 : the size of maxt (in LPLinit)
5 : the size of maxr (in LPLinit)
6 : the Filesearch path (all dirs in which LPL looks for a file)
7 : the actual working directory
8 : the LPL version
9 : the directory of the started application
10: the problemtype (if LPLwhere>0)
11: the solver status as string (if LPLwhere>0)
12: the assigned parameter-list (see : APL)
13: the log message
15: the clientID
16: version type: (Free, Professional, Enterprise)
otherwise : sP is the empty string
17: parameter guiding the generation of DB or snapshot
```

The *problem type* and the *solver status* are explained elsewhere.

LPLsetP writes information in the same way as *LPLgetP* gets it. If the parameter *which* is the number as follows then the following information is written by the parameter *sP*:

```
1 : the modelname
2 : the compiler switches
6 : the File search path is extended by sP
7 : the working directory
12: the assigned parameterlist (see : APL)
otherwise : do nothing
17: parameter guiding the generation of a Database (0:all entities,
              1:data (parameters and sets only), 2,3: variable
              only.
```

LPLsolve solves an instantiated model in a LPO-file and saves the result in the LPX-file. It loads a model store in the file <modelname>.LPO, calls the appropriate solver and writes the solution to an LPX-file then exits. The parameter *opt* is the same as for *LPLcompile* (the compiler switches).

LPLsaveSnapshot saves a snapshot of the data stored in the LPL-kernel to a file, called snapshot-file. The parameter sP1 is the filename. It should have filename extention **sps** or **sns**.

LPLloadSnapshot loads an existing snapshot file previously saved with *LPLsaveSnapspot*. The first parameter (*sP1*) is the filename, the second parameter *add* is an integer. If *add* is zero then LPL's data store is cleared and replaced by the data in the snapshot. If *add* is different from zero, then the store is prepared to accept multiple snapshots. If *add* is -1 then a second snapshot is read as a comparative snapshot.

LPLgenACCESSdb generates a new ACCESS database from the actual LPL store of a model. Make sure that the empty database *access.mdb* is accessible in the *directory list* of LPL.

The next procedures work with a focus, that is, an internal pointer to a given entity of the model, and an LPLhandle. LPLhandle must be defined as a 4-byte integer. The focus can be used to retrieve information from the focused entity. The user can set the focus to any entity using the *LPLsetFocus*. While running a model,the focus is set automatically to the actual executing statement. This allows the user to access the attributes of the actual executing statement through the callback Function.

LPLgetHandle gets an LPLhandle to an entity given by a name (argument *name*), if possible. It is Zero if undefined. The argumentr *name* can be in dot-notation. Hence, if a name (say ‘abc’) is defined in a submodel (say ‘mySub’), then the name should be ‘mySub.abc’ (using the dot-notation of the identifiers). Note that (since LPL is case-sensitive) the name must exactly match the case-sensitivity of the declared entity name.

LPLgetFocusHandle returns an LPLhandle to the focus.

LPLsetFocus sets the focus to a gentity given by an LPLhandle (argument *h*), if

possible. While running, the focus is set automatically to the actual executing statement.

LPLisFocus returns true of false of whether the focus is set or not. Suppose that a model does not contain a SET entity, then a call to to **`LPLsetFirstEntity(1)`** cannot set the focus, hence a subsequent call to **`LPLisFocus`** returns FALSE.

LPLsetFirstEntity gets an internal focus to the first entity of a specific *genus*, is possible. The genus is given by an integer with the following meaning:

```
0  : any genus
1  : SET declaration
2  : PARAMETER declaration
3  : VARIABLE declaration
4  : CONSTRAINT declaration
5  : UNIT declaration
6  : MODEL declaration
7  : MAXIMIZE, MINIMIZE, QUERY statement
8  : READ statement
9  : WRITE statement
10 : CHECK statement
11 : OPTION statement
12 : VARIANT statement
13 : IF-(ELSE) statement
14 : WHILE statement
15 : FOR statement
16 : internal proc call
17 : external proc call
18 : model call
19 : Asignment statement
20 : ELSE (part of IF)
21 : END (WHILE,IF,FOR)
22 : END (MODEL)
23 : else (others)
```

Example: A call to **`LPLsetFirstEntity(0)`** sets the focus to the first entity – this must be the top-most MODEL entity of the model, since each model begins with a model entity. A call to **`LPLsetFirstEntity(1)`** sets the focus to the first SET entity within the model.

LPLnextEntity sets the focus the next entity, if possible, within the entity list of a corresponding genus.

A call to one of the three procedure *LPLsetFocus*, *LPLsetFirstEntity*, and *LPLnextEntity* will set an internal *HasFocus* to true or false, depending of whether the call was successful. This value can be returned by *LPLisFocus*.
As an example let us run through the whole model an collect all SET names. The code would be something like the following:

```
Declare : ThisSetName AS String
```

```
LPLsetFirstEntity(1)      //sets the focus to the first SET entity
while LPLisFocus() DO     // check the focus
   ThisSetName = LPLgetAttr(4)  //return the SET's name
   LPLnextEntity(1)             // go to the next SET entity
endwhile
```

LPLgetGenus returns the genus of the *focused* entity. The genus return value is given in the list above in the *LPLsetFirstEntity* definition.

LPLPivotSetParam sets a parameter for the pivot table generation. These parameter should be set before the *LPLPivotInit* function is called. If none is set, then default parameters are used. The function parameter *which* specifies which of the pivot parameter has to be set. The function parameter *n* specifies the value of that pivot parameter. *which* can be:

```
1 : the number of horizontally expanded indexes
2 : the number of excluded indexes
3 : the attribute to be returned (see LPLgetValue function)
4 : The aggregate operator (0:none, 1:sum, 2: count, 3:average)
5 : The element or string name (0:element, 1: string) (second
              parameter of LPLgetElementList)
6 : The sparcity index yes or no (0: no, 1: yes)
7 : The sparcity of the table (0: no, 1: yes) [--does not yet work]
8 : The expression (0-5)
9 : The subtree node (0 to …)
-1: Sort on the n-th row/column (for cols n is negative)
-2: Sort is ascending (n=0) / decending (n<>0)
10-29 : the order of the indexes (permutation)
```

The ordering of the indexes must be a legal permutation beginning at 1; if not LPL will take the default ordering. For example:

```
LPLPivotSetParam(10,2);
LPLPivotSetParam(11,1);
LPLPivotSetParam(12,3);
```

means that the second index should be at the first position, the first index at the second position, and the third index at the third position.

LPLPivotGetParam returns the value of pivot internal parameter. The function parameter *which* specifies which parameter to be returned:

```
1 : the total number of (non-empty) pivot elements
2 : the total number of rows of the pivot table
3 : the total number of colums of the pivot table
4 : the number of colums used for indexes
5 : the number of rows used for indexes
```

LPLPivotInit create an internal data structure for a *focused* entity. It must be called before the next three functions.

LPLPivotX and ***LPLPivotY*** return the (x,y)-position of the current pivot element within a grid beginning with (0,0).

LPLPivotData returns the content of the (x,y) cell. After the data has been returned, this function advances to the next (non-empty) (x,y) cell. LPLPivotX returns -1, if the end of the table has been reached. Example code for representing a pivot table in the grid *Grid*:

```
LPLPivotInit;
while LPLPivotX<>-1 do begin
  x:= LPLPivotX; y:= LPLPivotY;
  Grid.Cell[x,y] := LPLPivotData;
end;
```

LPLsetSelect sets the selection of elements for pivot tables of an index-set. The set must be in the focus. The parameter *n* is the n-th element of the set. If n is zero then the selection is applied to all elements. The parameter *what* is 1, 0 or -1, which means select, unselect or inverse the selection. (By default *all* elements are selected.) Example:

```
LPLsetSelect(0,0);
LPLsetSelect(2,1);
```

Means to unselect all elements from the focused set and then select the second element. Subsequent pivottables that contain this index-set will only display the table with the second element.

LPLgetElementList returns the *n*-th element-name-list or element string list of the *focused* entity. If the focused entity is not indexed then an empty string is returned, if it is a basic set then the n-th element of the set is returned. If it is indexed, then the n-th entry is returned with the element names separated by a '@' character. Example: Collecting all element names of an entity named 'abc' could be done with the following pseudo code:

```
...
integer n := 0; string sP := ' '; list of string Li;
LPLsetFocus('abc');
while sP <> '' do begin
  n := n+1;
  LPLgetElementList(n,0,sP);
  Li[n] := sP;
endwhile;
...
```

If the *n*-th name does not exist, then *sP* is returned as an empty string. The second parameter is *0*, if the element name is asked otherwise the corresponding STRING attribute is returned. Note that since data change in time, the n-th element name list might be different between different calls.

LPLgetAttr returns an attribute of the *focused* entity. The returned string is stored in *sP*. Note that the attribute is returned exactly as it is stored in the source code of an

LPL model. The parameter *attr* can have the following value:

```
1 : the pretype attribute is returned
2 : the type attribute is returned
3 : the genus attribute is returned
4 : the name attribute is returned
5 : the index attribute is returned
6 : the expression attribute is returned
7 : the condition attribute is returned
8 : the range or the subject-to attribute is returned
9 : the unit attribute is returned
10: the if/priority/probability attribute is returned
11: the to/from attribute is returned
12: the first alias name is returned
13: the second alias name is returned
14: the quote attribute is returned
15: the comment attribute is returned
16: the default attribute is returned
17: the string attribute is returned
18: the FREEZE attribute is returned
19: the dot-notated statement name is returned
20: the name attribute as dot-notation is returned

21: the delayed operator is returned
      (' ': not important, '0' for table assignment,
       '1' for :=, '2' for table defs, and '3' for definitions)
22: <sourcefileName> , <line> , <col> of entity
23: the expression attribute as a postfix tree
24: the condition attribute as a postfix tree
25: the range attribute as a postfix tree
26: the unit attribute as a postfix tree
27: the if-attribute as a postfix tree
28: the to/from-attributes as a postfix tree
```

Suppose the following entities have been declared in a LPL model:

```
PARAMETER xx{i,j} UNIT [1000*m];
VARIABLE yy{i,j,k,l | i<>j AND a>8};
```

then the following calls to *LPLgetATTR* :

```
LPLsetfocus('xx'); LPLgetATTR(5,sP);
LPLgetAttr(9,sP);
LPLsetfocus('yy'); LPLgetATTR(5,sP);
```

return the following three strings in *sP*:

```
{i,j}
1000*m
{i,j,k,l | i<>j AND a>8}
```

If the entity does not have the corresponding attribute then an empty string is returned. If the parameter *i* is in the range [21…26], then a string is generated that represents an internal LPL expression structure in postfix-notation (only for advanced users!). The structure of the resulting string is documented elsewhere). An empty string is returned for an empty tree.

LPLgetValue returns a value from the *focused* entity that is at the position *n* within the table of the focused entity. The parameter *attr* can be:

```
20 : the value is returned
21 : the rhs is returned
22 : the lhs is returned
23 : the lrhs (slack) is returned
24 : the lower bound is returned
25 : the upper bound is returned
26 : the dual value is returned
27 : the lower range is returned
28 : the upper range is returned
```

LPLgetValueS is the same as *LPLgetValue*, but it returns a string for the focused entity. The parameter attr can be:

```
30 : the value of a string parameter is returned
31 : the enam is returned (the focus must be a set)
32 : the snam is returned (the focus must be a set)
```

LPLgetM returns a global data from LPL as double. The function returns the following value for the parameter *attr* :

```
35 : solver status is returned [0..7]
36 : problem type [0..15]
37 : elapsed time of a model run in msecs
```

LPLgetDependencyFrom, ***LPLgetDependencyTo***, and ***LPLgetDependencyKind*** are only for advanced users and are not documented.

11.2 USING THE LIBRARY

The library can be used in every programming environment that allows one to load true 32-dll libraries. Several examples are given.

11.2.1 USING THE LPL LIBRARY FROM DELPHI

The three executables **lplc.exe**, **lpls.exe** and **lplw.exe** could be built quite easily using the LPL library *lpl.dll*. The complete source code of *lplc.exe* (using the library) in Delphi would be:

```
program lplc;             //generates the program lplc.exe
  {$APPTYPE CONSOLE}
  const dllLpl = 'lpl.dll';
  type TProcC = procedure(var s:pChar); stdcall;
       TProc  = procedure;

  procedure LPLsetCallbacks(c:TProc; w:TProcC); stdcall; external dllLpl;
  procedure LPLinitParam; stdcall; external dllLpl;
  procedure LPLinit(Fn:PChar;maxa,maxt,maxr:integer); stdcall; external dllLpl;
  function  LPLcompile(opt:pChar):integer; stdcall; external dllLpl;
  procedure LPLfree; stdcall; external dllLpl;

  procedure MyCallBack; begin {write('.');} end;
  procedure MyWrite(var s:pChar); stdcall; begin writeln(s); end;

begin
  LPLsetCallbacks(MyCallBack,MyWrite);
```

```
  LPLinitParam;
  LPLinit('',0,0,0);
  LPLcompile('');
  LPLfree;
end.
```

The application assigns the callback and the LOG-output (*LPLsetCallbacks*), then it reads the parameters (modelname, compiler switches, size of memory allocation) (*LPLinitParam*), allocates memory for the LPL kernel (*LPLinit*), then it compiles and runs the model (*LPLcompile*) and finally cleanup the LPL kernel (*LPLfree*) and exits. Hence, this implements a complete run of an LPL-model. This application can be called as:

```
lplc MyModel ww
```

MyModel is a LPL-file and *ww* are the compiler switches. The two parameters *MyModel* and *ww* are automatically handled and read by the *LPLinitParam* procedure.

11.2.2 USING THE LPL LIBRARY FROM VISUAL BASIC

Here is a complete example with Visual Basic. We suppose that the LPL model file to compile is called “alloy.lpl”. Furthermore, the *lpl.dll* should be in Window's system directory to be found in this code. The complete code of the VB module is as follows:

```
Option Explicit

' Import functions from library lpl.dll

Private Declare Sub LPLsetCallbacks Lib "lpl" (ByVal cl As Long, ByVal wl As Long)
Private Declare Sub LPLinit Lib "lpl" (ByVal Fn As String, _
    ByVal maxa As Long, ByVal maxt As Long, ByVal maxr As Long)
Private Declare Function LPLcompile Lib "lpl" (ByVal pOpt As String) As Long
Private Declare Sub LPLfree Lib "lpl" ()
Private Declare Function LPLwhere Lib "lpl" () As Long
Private Declare Sub LPLsetError Lib "lpl" (err As Long)
Private Declare Sub LPLgetP Lib "lpl" (ByVal n As Long, s As String)

Public AbortBottonClick As Boolean 'this is a boolean set to true by a button
                                   'click (clicking the botton aborts an LPL run)

Private Sub MyWriteLog(s As String)  'LPL WriteToLog callback
  ' Note that s cannot be accessed in VB, so it must be read through LPLgetP 13
  Dim s1 As String
  Dim p As Long
  s1 = String(256, " ")
  LPLgetP 13, s1
  p = InStr(s1, Chr(0)) - 1
  s1 = Left(s1, p)
  Debug.Print s1
End Sub

Private Sub MyCallBack()  'LPL general callBack
  Dim where As Long
  Dim s As String
  where = LPLwhere
  s = Switch(where=-1,"error",where=0, "0", where=1, "LPL initialized", _
    where=2, "parsed", where=3, "run ok", where=4, "run ok", where=5, "5", _
    where=6, "parsing...", where=7, "running...", where=8, "const gen...", _
    where=9, "solving...")
  ' Debug.Print "MyCallback returns: " & s
  If AbortBottonClick Then LPLsetError (599)
  AbortBottonClick = False
End Sub

Public Sub Compile()
  Dim lRet As Long
  LPLsetCallback AddressOf MyCallback, AddressOf MyWriteLog
  LPLinit "alloy.lpl", 0, 0, 0
  lRet = LPLcompile("wvv")
  LPLfree
```

```
End Sub


Sub main()
  AbortBottonClick = False
  Compile
End Sub
```

A call to the routine *Main*() will call the LPL-compiler, generate a MPS-file or a LPO-file (depending on the compiler switches), call the solver, and finally write the result into the NOM-file exactly as from the executable *lplc.exe*. Furthermore, all LPL callback messages are written into the Immediate (debug) window in VB as well as to the LOG-file *lpllog.txt*.

11.2.3 USING THE LPL LIBRARY FROM C++

The following function in C++ shows how to compile and run the model "*alloy.lpl*" using C++. It does it just like the *lplc.exe* program from the LPL distribution. Note that the *lpl.dll* should be in Window's system directory or somewhere that it could be found from this code. The complete code of the C++ module is as follows:
(I am grateful to Andreas Klinkert who has written and tested this code.)

```
//---------------------------------------------------------------------------
// Language: C++, Win32API
// Date: 06.03.06
//---------------------------------------------------------------------------
// Language: C++, Win32API


int CallLplDll(void)
{
    // Define types of function pointers to imported DLL functions:
    typedef void (CALLBACK* LPFNDLL_LPLinit)(LPCSTR, INT, INT, INT);
    typedef INT (CALLBACK* LPFNDLL_LPLcompile)(LPCSTR);
    typedef void (CALLBACK* LPFNDLL_LPLfree)(void);

    // Declare function pointers to imported DLL functions:
    LPFNDLL_LPLinit LPLinit = NULL;
    LPFNDLL_LPLcompile LPLcompile = NULL;
    LPFNDLL_LPLfree LPLfree = NULL;

    // Load LPL DLL:
    LPCSTR lpszDllFile = "lpl.dll";  // Name of DLL file.
    HINSTANCE hDll = ::LoadLibrary(lpszDllFile);  // Get DLL handle.
    if (hDll == NULL) {
            // Error loading DLL.
            return 1;
    }

    // Load DLL functions:
    LPLinit = reinterpret_cast<LPFNDLL_LPLinit>(::GetProcAddress(hDll, "LPLinit"));
    LPLcompile =
        reinterpret_cast<LPFNDLL_LPLcompile>(GetProcAddress(hDll, "LPLcompile"));
    LPLfree = reinterpret_cast<LPFNDLL_LPLfree>(GetProcAddress(hDll, "LPLfree"));
    if (LPLinit == NULL || LPLcompile == NULL || LPLfree == NULL) {
            // Error loading DLL function.
            FreeLibrary(hDll);
            return 2;
    }

    // Execute DLL functions:
    LPCSTR lpszModelFile = "alloy.lpl";  // Name of LPL model file.
    LPCSTR lpszCompileOptions = "ww";
      // Possible options: "", "w", "ww", ... (see LPL reference).
    LPLinit(lpszModelFile, 0, 0, 0);  // Initialize LPL.
    const int nRes = LPLcompile(lpszCompileOptions);
      // Compile LPL model with specified options.
    if (nRes != 0) {
            // Error compiling model.
            LPLfree();
```

```
            FreeLibrary(hDll);
            return 3;
    }
    LPLfree();  // Finalize LPL.

    FreeLibrary(hDll);
    return 0;
}
//---------------------------------------------------------------------------
```

11.2.4 USING THE LPL LIBRARY FROM JAVA

A java program must use the library *lplj.dll* (instead of *lpl.dll*). Both libraries are identically from the functional point of view. Java needs a special interface: the Java-Native-Interface, therefore some functions have different signatures. See file NativeLPL.java for the correct signature. On the base of the class *NativeLPL* and the two interfaces *LogCallback* and *LPLCallback* (as displayed above), one can implement, for example, the complete LPL console compiler (working in exactly the same way as *lplc.exe*) as follows:

```
public class NativeLPLDemo implements LPLCallback, LogCallback {
  public static void main(String[] args) {
    NativeLPLDemo demo = new NativeLPLDemo();
    demo.compile();
  }

    //implement LPL callbacks
    public void callback() { ;} //nothing to do
    public void callback(String message)
      {System.out.println(message);}

    private void compile() {
      NativeLPL.LPLinitParam();
      NativeLPL.LPLinit("",0,0,0);
      NativeLPL.LPLcompileWithCallbacks("",this,this);
      NativeLPL.LPLfree();
    }
}
```

The complete example is stored in the zipped file *javaexam.zip*. To execute the example, do the following.

1) Uncompress *javaexam.zip* to a new folder.
2) Copy *lplj.dll* and *alloy.lpl* into that folder also (if they are not there already).
3) Compile the java files with the command:

   javac –classpath . *.java

4) Execute the class LPLC using the command:

   java LPLC alloy w

The last command is exactly the same as if you executed the program **lplc.exe** as:

lplc alloy w

APPENDIX A: LPL SYNTAX

The complete LPL syntax is presented as the extended Backus-Naur form. The following symbols are metasymbols (are not part of the LPL syntax), unless they are included in quotes (such as "{" or "["):

=	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
( )	determines the order of meta-operations
...	(in "A" | ... | "Z") means «all letters from 'A' to 'Z'»
"	encloses a literal (except """, which means the character ").

All other symbols are part of LPL. Reserved words are written entirely in uppercase letters. The starting symbol is *Model*.

```
Model =          (MODEL | FUNCTION) ModelHeader StatSeq END
ModelHeader =    Id [Attrs] ";"
StatSeq =        { NEntity | Model | IEntity | AssignStat | ForStat | WhileStat | IfStat | Proc }
NEntity =        [Type] [NKeyword] Id IList Attrs ";"
IEntity =        IKeyword [Id] [IList] Attrs ";"
Type =           [ ASSUMPTION | SEMI ] INTEGER | BINARY | DISTINCT | STRING | REAL | FREE | DATE
NKeyword =       SET | UNIT | PARAMETER | VARIABLE | CONSTRAINT
IKeyword =       READ | WRITE | CHECK | MAXIMIZE | MINIMIZE | QUERY | OPTION | ADDCONST
IList =          "{" Index {"," Index} ["|" Expr] "}"
Index =          [Id ("="|IN)] QualId ["[" Id {"," Id} "]"] |
                 Id ("="|IN) "{" "1:" Expr "}" | '*'[Id]
Attrs =          { ALIAS Id ["," Id] | FREEZE | DEFAULT (Number|String) | "[" Expr "]" | UNIT "["
                 Expr "]" | STRING Id [Comment] | TO ["+" | "-"] Expr | FROM Expr | Comment |
                 String | IF Expr | SUBJECT TO List | (PROB|PRIORITY) Expr | AssOp (Expr|Table) }
AssignStat =     QualId IList Attrs ";"
ForStat =        FOR IList DO StatSeq END
WhileStat =      WHILE Expr DO StatSeq END
IfStat =         IF Expr THEN StatSeq [ELSE StatSeq] END
Proc =           (EMPTY|FREEZE|UNFREEZE) List ";" | QualId ";" | LibraryCall ";"
List =           ["~"] QualId { "," ["~"] QualId }

Expr =           SimpleExpr {Relation SimpleExpr}
SimpleExpr =     [Indexing] Term {MulOp Term}
Term =           [ "+" | "-" | "#" | "~" | "$" ] Factor
Factor =         Number ["[" Expr "]"] | Id IN QualId | "(" Expr ")" | QualId ["[" Expr "]"] |
                 Funct "(" Expr ")" | String
Relation =       "," | ":=" | ".." | "->" | "<-" | "<->" | XOR | | OR | NOR | AND | NAND | "=" |
                 "<>" | "<" | "<=" | ">" | ">=" | "+" | "-" | "&" | "?<" | "?>"
MulOp =           "*" | "/" | "^" | "%"
Indexing =       [IndexOp] IList
IndexOp =        OR | XOR | NOR | AND | NAND | EXIST | FOR | FORALL | PROD | MAX | MIN | PMAX |
                 PMIN | ROW | COL | SUM | (ATLEAST|ATMOST|EXACTLY) "(" Int ")" | FOR
Funct =          NOW | ABS | BREAK | CEIL | COS | EXP | FLOOR | LOG | SIN | SQRT | TRUNC | RND |
                 RNDN | SORT | IF | ARCTAN | POSSTR | SUBSTR | WHILE

Table =          TableA | TableB | TableC
TableA =         "[" {Data} "]"
TableB =         "/" [MultiOpt] {SubTable} "/"
SubTable =       [lp TemplateOpt rp] [ColonOpt] ListOpt
MultiOpt =       "|" Id {Id} "|"
TemplateOpt =    EleOrStar { "," EleOrStar }
ColonOpt =       ":" ["(tr)"] {Element} ":"
ListOpt =        {{Element} {Data} [","]}
lp =             "(" | "["
rp =             ")" | "]"
Data =           Number | "." | String
Element =        char1 {char1} | String
EleOrStar =      element | "*"
TableC =         "/" Int ":" Int "/"

Comment =        """ {char} """
String =         "'" {char} "'"
QualId =         Id { '.' Id}
Id =             letter {letter | digit}
Number =         Int | Real | Date
Int =            digit {digit}
Real =           {digit} "." {digit} [ "E" ["+" | "-" ] Int ]
Date =           "@" {digit} [ "-" {digit} [ "-" {digit} [ "T" {digit} [ ":" {digit} [ ":"
                 {digit} ]]]]]
AssOp =          ":=" | ":" | "->" | "<-" | "<->"
digit =          "0" | ... | "9"
letter =         "A" | ... | "Z" | "a" | ... | "z" | "_"
char =           (any character)
char1 =          any chars except blanks or one of the 13 chars: ( ) * , / : ; [ | ] ' ”
```

REFERENCES

HÜRLIMANN T., Reference Manual for the LPL Modeling Language, file *manual.pdf* at the LPL-site (this document).

HÜRLIMANN T., User Manual for the LPL Modeling Language, file *user.pdf* at the LPL-site.

HÜRLIMANN T., LPL Modeling Tutorial, file *tutor.pdf* at the LPL-site.

HÜRLIMANN T., [2004a], LPL: A mathematical programming language, An Introduction, Working Paper, Departement of Informatics, June, Fribourg, file *intro.pdf* at the LPL-site.

HÜRLIMANN T., [2004b], LPL: Eine mathematische Modelliersprache, Eine Einführung, Working Paper, Departement of Informatics, June, Fribourg, file *intro-g.pdf* at the LPL-site (German version of [Hürlimann 2004a]).

HÜRLIMANN T., [1999], Mathematical Modeling and Optimization, An Essay for the Design of Coputer-Based Modeling Tools, Kluwer Academic Publ., (Applied Optimization 31).

LPL-Site: www.virtual-optima.com