REFERENCE MANUAL OF LPL

Tony Hürlimann

info@matmod.ch

November 30, 2022
Version 6.93

MatMod GmbH
CH-6300 Zug Switzerland
Email: info@matmod.ch
WWW: https://matmod.ch

Para mi amada mujer Lilí
y mis niñas adoradas Malika y Selma

Acknowledgements

I wish to thank the following institutions for their financial support and the use of their infrastructure. The first institution I would like to thank is the Swiss National Science Foundation who supported my research (under Project No. 1217-45922.95 and 12-55989.98). The second is the Department of Informatics at the University of Fribourg in Switzerland (DIUF) where I developed and implemented most of my ideas in a favorable and agreeable atmosphere of collaboration. The Department also let me use over many years all the necessary equipment to accomplish this work.

Availability of the LPL-Modelling-System

The mathematical modeling and all case studies in this book are based on the software LPL[1]. It is my own contribution to the field of computer-based mathematical modeling.

You do *not* need to install any software to run most of the examples in this manual – all you need is an electronic version of this manual, Internet connection, and an Internet browser (see later). However, a demo-version of LPL is available free of charge from the Department of Informatics at the University of Fribourg (DIUF), Switzerland. Currently, there is an implementation for Windows. A Linux console version is also available.

LPL – the software – together with documentation containing this Reference Manual, several papers and model examples can be downloaded at :

MatMod Homepage

[1] The secret meaning of **LPL** is *L'art Pour L'art* – because modeling should be considered as a science – but also as an art. Initially, it was an abbreviation of *Linear Programming Language*, since it was designed exclusively for mathematical Linear Programs. During the intervening years, LPL's capability in logical modeling has become so important that one could also call it *Logical Programming Language*. My intention, however, is to offer a tool for full model documentation too. Therefore, in the future it might also be called *Literate Programming Language* (see [1]).

Contents

Chapter 1

Introduction

"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**

This document is the Reference Manual for the mathematical modeling language LPL. It contains the complete syntax and semantic specification. A free version of LPL and various shorter papers as well as many modeling examples are available from the LPL-Site at: MatMod.

1.1. What is LPL?

LPL is a structured mathematical and logical modeling and programming language which allows one to build, maintain, modify and document large linear, non-linear, and other mathematical models.

Large optimization problems encoded as models in the LPL language are interpreted by the LPL executable. This Program **communicates with** various commercial and free **solver packages** to solve linear, mixed-integer and non-linear problems. Among the supported solvers are Gurobi, Cplex, Xpress, MOPS, GLPK, lp_solve, XA, Mosek, Conopt, Knitro and Loqo. In addition to those LPL comes with its own LP-solver that is suitable for small LP-problems as well as with a heuristic Tabu-search algorithm that allows to efficiently solve permutation problems such as many scheduling problems.

The LPL language grew **motivated by practical modeling** tasks and is as such perfectly suitable to express the problems encountered in practice in an concise and readable way. As a result, LPL is at its core **declarative**, similar to the mathematical models on paper. However, LPL also offers the benefits of an **algorithmic** programming language, based on standard control flow. This is often useful for pre- and post-processing data but also to loop through multiple 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 themselves be entire optimization models which can be processed individually or communicate their results. None the less, LPL is clearly focused on the model representation and algorithmic programming capabilities are limited. It is not a stack-based programming language. (As a result recursive function calls are not possible and all parameters are static.)

1.2. Key features

The following list gives a quick overview of the main features and highlights of LPL.

- A simple and concise syntax of mathematical and logical models with indexed expressions close the mathematical notation. By remaining close to conventional notation, the LPL language is very easy to learn and understand when having a background in modeling.

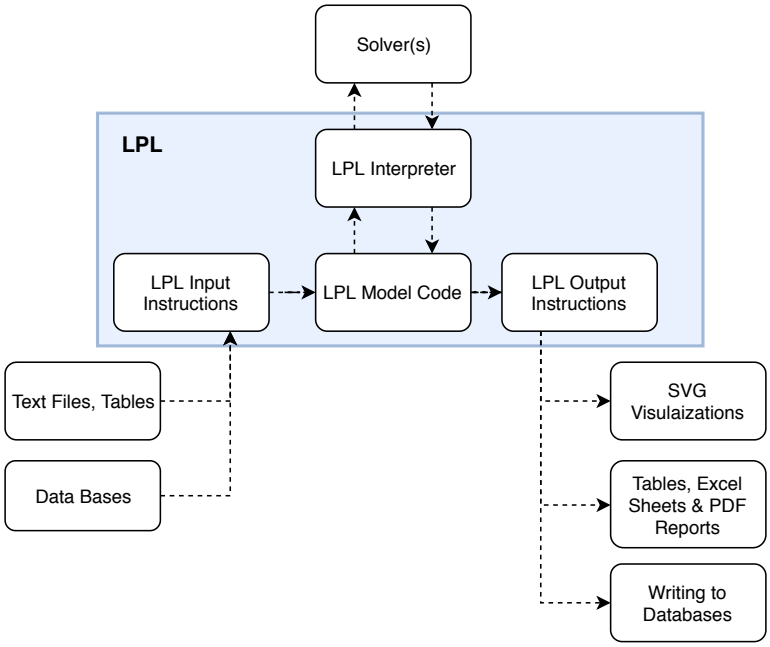


Figure 1.1: An overview how LPL is commonly employed.

- LPL has strong support for large multidimensional arrays. Those are by default sparse, which is an advantage in many practical problems, where we normally work with graphs and networks that are large and have a sparse adjacency matrix.
- LPL supports modular programming: The key unit called *model* is used to separate unrelated functionalities and increase readability and maintainability of more complex problems.
- Code is translated into an intermediate language that is then interpreted by the LPL executable which communicates with solvers. This intermediate language supports some aspects of reflection, that can be valuable for debugging and model exploration. For an overview of how LPL is commonly employed see figure 1.1.
- Fast production of MPSX standard files and other output files is a key feature of LPL. LPL also supports interaction most commonly used LP and MIP solvers (Gurobi, Cplex, Xpress, MOPS, GLPK, lp_solve, XA, Mosek, Conopt, Knitro and Loqo among others).
- LPL comes with a built-in Tabu-search heuristic which allows the efficient solution of permutation problems (a common problem type,

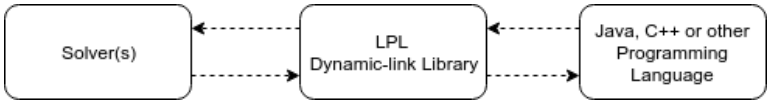


Figure 1.2: An overview how LPL integrates with other programming environments.

found in scheduling). A simple LP solver is also part of LPL. However, this solver is mostly suitable for small problems or an educational setting.

- LPL can easily read and write from and to a variety of sources. Text files, tables (such as the Excel format), can be read and written to. The software comes also with a Draw library with which solutions can be visualized and exported as SVG files.

- Connection to common databases is possible with the same IO-functions. Not only is it easy to read from databases, LPL can also create Databases in the third normal form (3NF).

- An innovative and high-level report generator is also part of LPL. The report generator that can read/write complete tables form and to databases and can generate reports as PDF files or other formats.

- LPL also comes as dynamic link libraries that can be integrated into other applications. Examples for Java, C++, Pascal and Visual Basic are fond in chapter 10. The diagram 1.2 shows how LPL is used in this scenario.

- LPL is split into a client and server, that can be deployed separately. This allows setting up a dedicated server for the solver, with which clients communicate over the web.

- A sophisticated model environment to browse and edit large models comes with LPL. Tools for debugging and viewing the model such as explicit equation listing are integrated.

- LPL code with comments can be used to directly generate a documentation file using the powerful LaTeX environment.

1.3. Overview of the Manual

Chapter 2 explains the use of the LPL executables.

Chapter 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.

Chapter 5: indices and index-lists, the most fundamental elements of LPL, are explained in detail.

Chapter 6: internal data and table formats of data in a model are explained.

Chapter 7 gives detailed information about variables and constraints.

Chapter 8: the Input/Report Generator of LPL, which contains the whole input/output handling, is described.

Chapter 9: other miscellaneous elements, such as compiler switches, are explained.

Chapter 10 explains the use of the dynamic link library, normal and Java-specific (*lpl.dll* and *lplj.dll*).

Appendix A: The entire language syntax description is collected.

1.4. What to do next ?

- First read the README.TXT file in the software distribution to get quick information about installing LPL.
- Read an introductory paper or a brief overview of the LPL modeling language. For example, the modeling3.pdf Paper or the lpl.pdf Paper.
- If you are a new LPL user, read the Tutor Paper tutor.pdf and run the tutor examples.
- Read through Chapter 3 and Chapter 4 to get an detailed view of LPL basic syntax. Here, one also can find the references of all functions.
- If you are not familiar with the very basics of mathematical indexed notation, then read the indexing.pdf Paper.

- If you need to build an application in, say, Python, Visual Basic, Java, or another language using the LPL capabilities and its powerful dynamic library, read Chapter 10.

Chapter 2

Install and Run LPL

LPL can be downloaded from the LPL-Site as an **Free Package**, which contains some limitations. All files at the site are stored as a compressed file *install-lpl.exe*. Go to the install guide at: install.pdf witch gives more detailed instructions on how to install and use LPL.

2.1. Functional Overview

LPL is a modeling language to code mathematical and other models. The LPL-compiler first parses the LPL code and (optionally) runs it. The source code is written in LPL-syntax and is stored in a text-file, called **LPL-file**. Depending on the compiler switches, it generates various output files. It can call a solver (another program that solves 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 function 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 file *lplmsg.txt* (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 ???-files (where ??? are 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 it 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. 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. Various versions of this file are possible depending on compiler switches.

- **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 readable file and cannot be displayed or printed directly.

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

- **IN1-file**: another human readable form of the LPO-file (similar to the INT-file).

- **SPS-file**: A snapshot of the complete data (inclusive the variables) stored in the LPL-kernel can be saved to a file. Later on this can be loaded with a single instruction (see read/write).
- **LPX-file**: a solution file of the model. This file is generated by the Internet Server and sent to the LPL-client.
- **TEX-file**: a LaTeX file for documenting the model (see 9.9).
- **ENG-file**: a separated doc file of the model documentation is generated.
- **LPY-file**: regenerates the model source code.
- **SYM-file**: produces a list of the internal symbol table.
- **BUG-file**: a file for debugging the model.
- **STO-file**: another debug file to inspect the model instance.
- **SQL-file**: an SQL script for creating a complete relational database out of a LPL syntax specification (see chapter 8).
- **SQ2-file**: an LPL source file containing the read/write statements needed to communicate with the database created by the SQL-script file.
- **IIS-file**: If the model is infeasible then this file will be generated automatically (if solver cplex, lindo, or gurobi are used).
- **Two LOG-files**: which document the compiling process (files: *lpllog.txt* and *lplStat.txt*).

These files are generated using different compiler switches (see 9.7). 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'. Several files (EQU, BUG, MPS, INT-file) are generated multiple times – after each optimization. These files are numbered.

2.2. Programs

LPL consists of five executables of the compiler: two console versions (lplc.exe, lpls.exe), a wysiwyg version (lplw.exe), and two dynamic link libraries (lpl.dll, lplj.dll). All executables and the libraries are based on the same code.

2.2.1 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 9.7). Note that *lplc.exe* can be interrupted gracefully anytime by pressing the ESCAPE key. The program is called as follows:

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

<modelfile> is the LPL-file. The LPL-file *must* have a filename extension *lpl*. CompilerSwitches can be empty. The APL parameter specifies the assigned parameter list (APL) (see 9.8) for a run.

Example call: runs *alloy.lpl* and generate a solution in file *alloy.nom* :

```
c:\lpl\>  lplc alloy
```

2.2.2 lpls.exe

The executable <it lpls.exe is another 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 in a special format, then it exits. The program is called as follows:

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

<modelfile> is the LPO-file. The LPO-file must have a file extension lpo. CompilerSwitches can be empty. But only a few options are interpreted. The parameter APL is not used.

This program is used as an LPL-server for the LPL Internet Service. The Service can be used automatically by any LPL client just by configuring the Internet-Solver. The process is as follows: A LPL client (user) somewhere in the Internet compiles the model with LPL and writes an LPO-file that is transferred to the LPL server through the Internet. The generated LPX-file at the LPL Server then is retransferred to the client. The executable *lpls.exe* is typically installed in a Tomcat environment, where it can be called as a process.

2.2.3 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 the other executables:

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

The User manual at user.pdf explains the modeling user interface in more details.

2.2.4 Libraries *lpl.dll* and *lplj.dll*

The library *lpl.dll* (and *lplj.dll* for Java) is a dynamic link library that integrates the complete LPL functionality and can be called and used from any other applications. This library and its use is explained in details in Chapter 10.

Chapter 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 following Unicode character have a special meaning in a LPL code, they are used to form "words" in the language:

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

Since all these characters are printable, an LPL code can be edited and manipulated by any text editor. Names (identifiers), reserved words, operators, and other elements (the "words" or "tokens" of the language) are formed using one or more of these 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 can be placed to separate the words. 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 sub-models or distribute it into several files helps also to make the model readable and maintainable.

3.2. LPL Tokens

An LPL model consists of different basic elements: the tokens (or "the words" of the language), as already mentioned. They are:

1. *Reserved Words*, words that have certain fixed meaning, they form an integral part of the language.
2. *Identifiers*, words that designate and name indices, elements of indices, parameters, variables, constraints, and other names specified by the user.
3. *Numbers*, *strings* and *dates*, words to define various data.
4. *Functions*, words that call a certain code to calculate a specific value or run a code.
5. *Operators*, words that are used to construct expressions.
6. Various other "words" such as comments.

All these elements are called "tokens" (words) and are the basic building block for the language. The different kinds of tokens are explained 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 and they have a fixed meaning. The reserved words are:

addconst	**alldiff**	**and**	**argmax**	**argmin**	**atleast**
atmost	**binary**	**break**	**constraint**	**count**	**date**
default	**do**	**else**	**end**	**exactly**	**exist**
expression	**external**	**for**	**forall**	**friend**	**frozen**
function	**if**	**in**	**integer**	**max**	**maximize**
min	**minimize**	**model**	**nand**	**nor**	**or**
parameter	**priority**	**prod**	**real**	**return**	**set**
solve	**string**	**subject_to**	**sum**	**then**	
variable	**while**	**within**	**xor**		

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. Hence, `AllProducts` and `ALLPRODUCTS` are two different identifiers. Examples are:

```
Var_1
TEXT
_None
ImportedProducts
This_is_A_long_Identifier
3xY            (not correct, starts with a digit)
Two words      (not correct, cannot contain spaces)
```

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

```
Submodel.x     (denotes x in model Submodel)
b.x.i          (denotes i in model x within model b)
Draw.Rect      (a function in the Draw library)
```

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)   (same as: 0.0036)
0.5
.5          (a number can begin with a dot)
5.          (a number can end with a dot)
```

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

3.6. Dates

Dates are values to define a time point. Internally they are stored and handled as numbers, 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          (beginning of year 2003)
@2002-02       (beginning of February 2002)
@2001-05-12    (beginning of 12th of May 2001)
@2003-10-07T12:01:01
    (7th of Oct 2003 at midday plus 1 min. and 1 sec)
```

For a small model example see learn07[1].

3.7. Strings

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

[1] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn07

```
'MyString'
'another string'
'45\'==? and character in string'
```

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

```
\a          bell
\b          backspace
\f          form feed
\n          new line
\r          carriage return
\t          tab
\\          \   (backslash char)
\'          '   (single quote)
\"          "   (double quote)
```

Example: a string with two tabulators and a linefeed can be written as follows:

```
'ABC\tDEF\tGHI\n'
```

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:

```
'this is the first line\n\
 leading spaces in this line are removed\n'
```

Note that filepath-names in Windows can 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 "...". A demo model example that manipulates strings can be opened at learn26[2].

3.8. Delimiters

A blank (the space), all character with Unicode number lower then 32 (such as new-line or tab), all characters with Unicode number greater than 127 are considered as token delimiters. They separate the "words" of the language. Several consecutive blanks or tabs are considered as a single delimiter.

[2]https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn26

3.9. Comments

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

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 */
   ```

   This comments can be nested. It 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
   parameter b;   // here is another short comment
   ```

   Note that the double dash -- can also be replaced by `//` .

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.

3.10. Elements

Elements of a set can be identifiers, strings or numbers. Examples:

```
Summer      (an name, same as 'Summer')
'4to5'      (a string)
623         (a number)
```

Note also that the element syntax is case-sensitive. Hence, `'June'` and `'JUNE'` are considered different elements.

3.11. Simple 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
`within`	membership position of an element (index) within a set
`( )`	expression nesting
`<< >>`	piecewise linear expression
`^ % && ||`	power, modulo, bitwise and, bitwise or
`/ *`	division, multiplication
`<index-op>`	all indexed operators (see below)
`- + &`	binary plus, minus, string concatenation
`= <> <= >= < >`	relational operators (see below)
`and nand`	binary *and*-operators
`or nor`	binary *or*-operators
`-> <- <-> xor ..`	implication, rev. impl., equivalence, excl. or, range
`:=`	assignment operator
`, |`	a comma or a bar, for an expression list

These operators form expressions such as (see learn01[3]:

```
3 * 5 + 34.9        a numerical expression
3 * (5 + 21)        another numerical expression
X and Y xor Z       a Boolean expression
S & T               a string concatenation
```

The parentheses `( )` can be used to change the operator's default precedence. The index operators `<index-op>` are explained in the next section.

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 within s`	return the position of element i within set s or 0 if i is not within s
`~ x`	return 1 if x=0, else return 0
`( x )`	return x , (nesting x)
`'x'`	return string x

[3] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn01

`<num>`	return the number `<num>`
`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 x bitwise and y
`x || y`	return x bitwise or y
`x / y`	return division (error if y=0)
`x * y`	return multiplication
`x - y`	return subtraction
`x + y`	return addition
`x & y`	return 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          result is: 18
3=2            result is 0 (=FALSE)
3<2-6          result is 1 (=TRUE)
1 or 6-6       result is 1 (=TRUE)
~(0 and 1+1)   result is 1 (=TRUE)
a:=3 , 4       result is 4 (side eff.: assigns 3 to a)
```

Note, there is no explicit *true* or *false* Boolean value. Like in the language **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 be

written just as *'a'*. Since the value of *'a'* can be a real, this rises a problem of precision and must be used with care. Note also that numerical and Boolean operators can be mixed in the same expression.

The **within**-operator tests, whether a particular element is within a specific set. The first argument must be a local index or a bounded index (see Chapter 5), the second argument must be an unbounded set name. See model learn03[4], learn05[5], or learn15[6].

The **relational** operators `= <> <= >= < >` compare numerical *or* alphanumerical values.

```
2 <= 3          result is 1
'abc' = 'bcd'   result is 0
```

Piecewise linear expression can be added within constraint expressions. The syntax is as follows:

```
... << x , {k}(a,b) >> ...
```

where `x` is a variable and the `{k}(a,b)` are the k piece-wise parts. An example is given in model Bill227[7].

The **assign**-operator `=:` need an identifier on the left-hand side and an expression on the right-hand side to assign a value to an user defined identifier.

3.12. Indexed Operators

Indexed operators are used together with sets. In Mathematics we use for example the expression:

$$\sum_{i=1}^{10} i$$

as a shortcut for

$$1+2+3+4+5+6+7+8+9+10$$

[4] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn03
[5] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn05
[6] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn15
[7] https://lpl.matmod.ch/lpl/Solver.jsp?name=/Bill227

The symbol $\sum$ is an indexed operator for the summation. There are other indexed operators, such as Π, for multiplication, max and min for the maximum or minimum, or even Boolean indexed operator, such as $\bigwedge$ for the indexed *and*, and others[8].

In LPL, The indexed operators are keywords. The following words are defined:

`sum{i} a[i]`	return the sum of `a` over index i ($\sum_i a_i$)
`count{i} a[i]`	return the count of `a` over index i ($\text{count}_i\, a_i$)
`prod{i} a[i]`	return the product of `a` over index i ($\Pi_i a_i$)
`min{i} a[i]`	return the smallest `a[i]` ($\min_i a_i$)
`max{i} a[i]`	return the largest `a[i]` ($\max_i a_i$)
`argmin{i} a[i]`	return the index of the smallest `a[i]` ($\text{argmin}_i\, a_i$)
`argmax{i} a[i]`	return the index of the largest `a[i]` ($\text{argmax}_i\, a_i$)
`forall{i} a[i]`	same as: `and{i} a[i]` ($\forall_i a_i$)
`exist{i} a[i]`	same as: `or{i} a[i]` ($\exists_i a_i$)
`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`

The operator `for` is the loop operator in LPL. A shortcut for the loop is the missing indexed operator: just write `'{i}A'` to run the expression `'A'` as many times as `'i'` has elements (cardinality of `'i'`). The indexed operators are always followed by an index-list (see Chapter 5) and then by an expression. Examples:

[8] If the reader is unfamiliar with these operations, there is a paper that explains the details of the notation and mechanism of these operators: See Indexing Paper [2]

`sum{i} 1`	return the cardinality of `i` (same as: `#i`)
`sum{i} a[i]`	return sum all a over `a[i]` (`a[1]+a[2]+...`)
`max{i} i^2`	return the largest square of `i`
`exist{i} (a[i]=1)`	return true if at least one `a[i]` is 1, else false
`and{i} a[i]`	return true if all `a[i]` are true

3.13. Functions

LPL defines a certain number of functions that can be includes into expressions. There are general standard function, as well as functions collected in specific libraries.

3.13.1 List all Functions

General Functions

Abs(x)	Return the positive value of x
Addm(s,e)	Add an element e to set s
Alldiff(c)	Define Alldiff constraint c
Arctan(x)	Return Arctan of x
Ceil(x)	Return the smallest integer greater than x
ClearData(c,...)	Clear the data tables c,...
Complements(s)	Constraint: complements
Contains(x,i)	Alldiff function only
Cos(x)	Return the Cosinus of x
DateToStr(d)	Return date d as string
El(r,s)	Return element position in r of name s
En(r,x)	Return element name in r at position x
Exp(x)	Return e^x
Find(x,i)	Alldiff function only
Floor(x)	Return the greatest integer smaller than x
Format(s,e,...)	Return a formatted string s, filled with e,...
Freeze(c,...)	Inactivate constraints or fix variables c,...
GetAttr([r,] a)	Return attribute a of ref entity r as string
GetName([r,] a)	Return the name of reference r as string
GetParam(a)	Return a global parameter value as double
GetParamS(a)	Return a global parameter as string

GetProblemType()	Return problem type
GetSolverStatus()	Return the solver status
GetValue([r,] a)	Return a numerical value of reference r
GetValueS([r,] a)	Return a string value of reference r
Index(x,i)	Alldiff function only
Log(x)	Return the e log of x
Max(x,...)	Return the largest in the list
Min(x,...)	Return the smallest in the list
NextFocus(a)	Sets the focus the the next entity of genus a
NextPosition[(i)]	Jumps to the next data entry
Now()	Return actual date and time
OSCall(s)	Call an OS program s
Ord(c)	Return Unicode code of a char c
Read(s,e,...,[E])	Read data from file
Rgb(r,g,b)	Returns a Rgb color code
Rnd(x,y)	Returns an uniform random number in range [x...y]
Rnde(m)	Returns an exponentially distributed random number
Rndn(m,s)	Returns an normally distributed random number
Round(x[,y])	Returns a rounded integer of x
SetFocus[(r)]	sets the focus to reference r
SetPaths(c)	Add a file paths to the directory list
SetRandomSeed(x)	Sets the random seed to x
SetServer(s)	Sets the Internet address of the LPL server
SetSolver(s[,t])	Sets the solver and its parameters
SetSolverList(s)	Sets a list of solvers to s
SetWorkingDir(s)	Sets the working directory to s
Sin(x)	Return the Sinus of x
Sl(a[,b][,c])	add a slack variable
Sort(a,b[,n])	Sort data
Sos1(c)	Defines SOS1 constraint c
Sos2(c)	Defines SOS2 constraint c
Split(s,c,t,...)	Split a string s into parts separated by a char c
Sqrt(x)	Return the square root of x
Strbin(s	Return a integer from a binary string
Strbin0(n)	Return a binary string from a integer
Strdate(s[,c,n])	Return and convert string s to a date
Strfloat(s[,c,n])	Return and convert string s to float
StringToSet(s,p)	Links a set s to a (string) parameter p
Strlen(s)	Return the length of a string s
Strpos(s1,s)	Return the position of a substring s1 within a string s
Strreplace(s,s1,s2)	Returns a string that replaces s1 by s2 in s

Strreverse(s	Returns the reverse string of s
Strsub(s,b,l)	Returns a substring of s of length l beginning at b
Trunc(x)	Return the truncated integer of x
Unfreeze(c,...)	Activate constraints or unfix variables c,...
Write(s,...)	Write data to file
Writep(r,...)	Write table data to file

Draw Library (see also HERE)

Draw.Animate([id,t,id1,]dur,beg,fr,to[,fr1,to1,fr1,to1])	Animate a Draw
Draw.Arc([id,t,]x,y,r,a1,a2[,c2,w,o2,a])	Draw an arc
Draw.Arrow([id,t,] x,y,x1,y1 [,z,c2,w,o2,a])	Draw an arrow
Draw.CArrow([id,t,] x,y,x1,y1,d [,z,c2,w,o2,a])	Draw a curved arrow
Draw.CLine([id,t,] x,y,x1,y1,d [,c2,w,o2,a])	Draw a curved line
Draw.Circle([id,t,] x,y,ra [,c1,c2,w,o1,o2])	Draw a circle
Draw.DefFill(id,c1,c2,w,o1,o2)	Define the fill properties
Draw.DefFilter(id,f[,in1,re,x_i**])**	Define a filter
Draw.DefFont([id,] t,h,c1,c2,fw,fs,fv)	Define font properties
Draw.DefGrad(id,sm,x,y,x1,y1,fy,o_i**,**c_i**})**	Define a gradient
Draw.DefLine(id,sd1,sd2,sd3,slc,slj,sljl)	Define line properties
Draw.DefTrans(id,t, [x1,y1,z1])	Define a transformation
Draw.Ellipse([id,t,] x,y,rx,ry [,c1,c2,w,o1,o2])	Draw an ellipse
Draw.Line([id,t,] x,y,x1,y1 [,c2,w,o2,a])	Draw a straight line
Draw.Path([id,]t[,x_i**,**y_i**])**	Draw a path
Draw.Picture(t [,x,y,w,h])	Load/draw a picture
Draw.Rect([id,t,] x,y,w,h[,c1,c2,w,o1,o2,rx,ry,rot])	Draw a rectangle
Draw.Save(t)	Save drawings to file s
Draw.Scale(zx,zy [,x,y,w,h])	Define drawing space
Draw.Text([id,] t,x,y [,h,c1,c2,w,o1,o2,a,r])	Draw a text
Draw.Triangle([id,t,] x,y,ra [,c1,c2,w,o1,o2,rot])	Draw a triangle
Draw.XY(x,y[,z,w][,t,s,c])	Draw a XY plot

Statistical Library

Stats.Binomial(n,k,p,cum)	Return Binomial distribution
Stats.NegBinomial(n,k,p,cum)	Return negative Binomial distribution
Stats.Poisson(k,mu,cum)	Return Poisson distribution
Stats.Logseries(k,p,cum)	Return Log Series distribution
Stats.Geometric(k,p,cum)	Return Geometric distribution
Stats.Hypergeometric(n,k,n1,k1,cum)	Return Hypergeometric distribution
Stats.Gumbeldis(x,m,b)	Return cumulative Gumbel distribution

Stats.Normdis(x,m,s)	Return Normal cumulative
Stats.Norminv(x,m,s)	Return inverse normal cumulative

Graph Library

Graph.Bfs(r,c,n,m)	Return bfs tree c, of a graph r
Graph.Components(r,c)	Return components c, of a graph r
Graph.Dfs(r,c,n,m)	Return dfs tree c, of a graph r
Graph.Mincut(r,c [,s,th])	Return minimal cut
Graph.Mstree(r,c)	Return minimal spanning tree
Graph.SPath(r,c,s[,t])	Return length of shortest paths
Graph.Topo(r,c)	Return topological Sort

Geometry Library

Geom.Inside(x,y,X,Y)	Return true if (x,y) is inside polygon (X,Y)
Geom.Intersect(X,Y,i,j,m,n)	Return true if two line segments intersect

Struct Library

Struct.AddMap(x,y)	Return void
Struct.DeQueue()	Return a Queue entry
Struct.EnQueue(x)	add a Queue entry
Struct.GetMap(x)	Return a Map entry
Struct.InMap(x)	Check if in Map
Struct.Map(n[,c])	Create a Map
Struct.Queue(n[,c])	Create a Queue

3.13.2 General Functions

This section explains all functions.

Abs(x)

Back to index

Return the positive (absolute) value of x : $|x|$.

x : is an expression returning a double number

Return Value : double

Example :

`a:=Abs(-5.5)` Result : $a = 5.5$
`a:=Abs(5.5)` Result : $a = 5.5$

Addm(s,e) Back to index

Add an element e to set s

s : is an identifier defining a set

e : is a string or a number

Return Value : None

Example :

`Addm(i,'ABC')` Result : set i is extended with element ABC

Note : In LPL all sets are ordered. That means that each element has a position within a set. The element e is added to the set in the last position. The added element is the last element. All tables indexed over the set s are internally reorganized. Hence, this operation can be very expensive. If the set s has been defined as a integer range (*set s:=1..10*, for example), then the set is extended with a additional integer independent what e is. For an example see learn01b[9].

Alldiff(c) Back to index

Defines Alldiff constraint

c : a list of integer variable identifiers

Return Value : None

Example :

[9] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn01b

`constraint A: Alldiff({i} x[i])`
Result : All variables `x[i]` must have different values from each other
`constraint B: Alldiff({x,y,z,v,w)`
Result : All 5 variables must have different values from each other

Note : The variables should be *integer* variables. For an example see the model sudokuInt[10].

Arctan(x) Back to index

Return Arctan of x

x : a double. Specifies the tangent value.

Return Value : double. It is the angle for the tangent in radiant unit.

Example :

`a:=Arctan(1.0)` Result : $a = \pi/4 \quad (= 45^o)$
`a:=Arctan(-1)` Result : $a = -\pi/4 \quad (= -45^o)$

Ceil(x) Back to index

Return the smallest integer greater than x

x : is an expression returning a double number.

Return Value : double.

Example :

`a:=Ceil(-5.6)` Result : $a = 5$
`a:=Ceil(5.1)` Result : $a = 6$

ClearData(c,...) Back to index

Clear the content of data tables c,...

[10] https://lpl.matmod.ch/lpl/Solver.jsp?name=/sudokuInt

c,... : a list of identifiers, defining parameters, sets, variables, or (sub)-model names.

Return Value : None

Example :

`ClearData(a,b,c)` Result : data tables a, b, c are cleared
`ClearData(model)` Result :
all data tables within the model *model* are cleared

Note : Tables that are defined with the attribute `frozen` are *not* cleared.

Complements(s) Back to index

Constraint: complements

s : a constraint expression

Return Value : None

Example :

`Complements(3*x + 4*y)` Result : —

Note : This function is not yet implemented.

Contains(x,i) Back to index

Return true if alldiff variable xk,i contains i in subset k

x : is an partitioned alldiff variable

i : is an item within the alldiff variable

Return Value : boolean.

Example :

`Contains(x[k],i)` Result : true if subset k contains i

Note : Given a partitioned permutation variable $x_{k,i}$, then the function returns true of permutation item i is in the subset $x[k]$. A model example is capacitatedArcRouting1[11].

Cos(x) Back to index

Return the Cosinus of x

x : is an expression returning a double number.

Return Value : double.

Example :

`a:=Cos(3.14159)` Result : $a = 1$

DateToStr(d) Back to index

Return date d as string

d : is an expression returning a double number.

Return Value : a string defining a date/time.

Example :

`a:=DateToStr(Now())` Result : a:='15.08.2013 12:49:04'
`a:=DateToStr(40000)` Result : a=?06.07.2009'

Note : In LPL dates are stored as doubles. the date zero (0) is defined as `30.12.1899`. Another way to translate a date into a string is using the function Format.

El(r,s) Back to index

Return element s position in set r

[11] https://lpl.matmod.ch/lpl/Solver.jsp?name=/capacitatedArcRouting1

r : a set identifier.

x : a string

Return Value : an integer.

Example :

`a:=El(i,'Cc')` Result : a:=3 ('Cc' is 3rd element in set i

Note : See model learn35[12].

En(r,x) Back to index

Return element name in set r at position x

r : a set identifier.

x : an integer

Return Value : a string.

Example :

`a:=En(i,3)` Result : a:='Third' (is 3rd element in set i

Note : See model learn35[13].

Exp(x) Back to index

Return e^x

x : is an expression returning a double number.

Return Value : double

Example :

`a:=Exp(1)` Result : $a = 2.7183$
`a:=Exp(-1)` Result : $a = 0.3679$

[12]https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn35
[13]https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn35

Find(x,i) Back to index

Return true if alldiff variable xk,i contains i in subset k

x : is an partitioned alldiff variable

i : is an item within the alldiff variable

Return Value : boolean.

Example :

`Contains(x[k],i)` Result : true if subset k contains i

Note : Given a partitioned permutation variable $x_{k,i}$, then the function returns the position of permutation item i is in the subset $x[k]$.

Floor(x) Back to index

Return the greatest integer smaller than x

x : is an expression returning a double number.

Return Value : double

Example :

`a:=Floor(-5.1)` Result : $a = -6$
`a:=Floor(5.9)` Result : $a = 5$

Note : See also the function .

Format(s,e,...) Back to index

Return a formatted string s, filled with parameter list e,... at the placeholders.

s : a string containing place holders.

e,... : a list of parameters resulting in numbers, strings, or dates.

Return Value : a string, formatted

Example :

```
a:=Format('a=%5.2f',23.36543)
```
Result : $a =$ `'a=23.37'`

```
a:=Format('Result is: %d',12^2)
```
Result : `'Result is: 144'`

Note : The function has the same parameters as the function Write with the exception of the (first) file parameter. And also as the Write it can be indexed. For examples see the model tutor08e[14] and learn26a[15].

Freeze(c,...) Back to index

Inactivate constraints or fix variables c,...

c,... : a list of identifiers, defining parameters, sets, variables, constraints, or model names.

c,... : c can also be an indexed expression. In this case. An example is given in model learn34[16]

Return Value : None

Example :

```
Freeze(a,b)
```
Result : sets the `frozen` attribute for a and b

```
Freeze(mymodel)
```
Result : sets the `frozen` attr. of the model `mymodel`

```
Freeze({i|i<=2} C[i])
```
Result : This will remove the first two constraints of the constraint list `C{i}`

Note : The function Unfreeze clears the *frozen* attribute. If the *frozen* attribute of a *variable* is set then the value of a variable is fixed and cannot be modified by a solver or the ClearData function. If it is set for a *constraint* then the constraint is not passed to a solver, the

[14] https://lpl.matmod.ch/lpl/Solver.jsp?name=/tutor08e
[15] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn26a
[16] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn34

constraint is "switched off". If it is set for a *model* then the *frozen* attribute of all constraints within the model are set, all constraints are "switched off". If it is set for a *set* or a *parameter* then a call to the function *ClearData* has no effect on them.

GetAttr([r,] a) Back to index

Return attribute a of ref entity r as string

r : an entity identifier (as string)(set, parameter, variable, constraint, model)

a : an integer in the range: 2 . . . 26

Return Value : a string

Example :

```
a:=GetAttr(X,3)
```
Result : return the genus of X

```
a:=GetAttr(b,15)
```
Result : return the comment of b

```
a:=GetAttr(12)
```
Result : return first alias of focused entity

Note : This function returns an attribute of an entity. If the entity parameter r is missing then the attribute of the focused entity is returned (see function SetFocus. The parameter a determines which attribute is to be returned. They are:

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 index attribute is returned (internal use) (do not use)
7	the expression attribute is returned
8	the condition attribute is returned
9	the range attribute is returned
10	the if/priority attribute is returned
11	the index attribute is returned without condition
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	(Sparse) cardinality is returned
18	(Full) cardinality is returned
19	the frozen attribute is returned
20	the dot-notated statement name is returned
21	the name attribute as dot-notation is returned
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 if-attribute as a postfix tree

Any other value of the parameter a returns an empty string. See the model learn22[17] for an example.

GetName([r,] a)

Back to index

Return the name of reference r as string

r : an entity identifier as string (set, parameter, variable, constraint, model)

a : an integer)in the range: $0 \ldots 6$), any other integer return the ID name only

Return Value : a string the name of the instantiated indexed object

Example :

```
a:=GetName(X,0)
```
Result : `X[i1,A]` (see model)

Note : This function returns the instantiated name of a indexed object as a string. If the entity parameter r is missing then the attribute of the focused entity is returned (see function SetFocus. Depending on the parameter a it returns different formats (see model learn23[18] and learn27[19] for an example) :

0	use comma separated element names
1	use comma separated string names

[17] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn22
[18] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn23
[19] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn27

2 use _ instead of commas (element names)
3 use _ instead of commas (string names)
4 use tabs instead of commas (element names)
5 use tabs instead of commas (string names)
6 like 0, but index name and '=' are added

GetParam(a) Back to index

Return a solver attribute parameter

a : an integer 1 . . . 16

Return Value : a double

Example :

```
a:=GetParam(3)
```
Result : the problem type

Note : This function returns some general model information.

1 LPL running time in millisecs
2 total length solver time after solution
3 problem type (same as *GetProblemType*)
4 solver status (same as *GetSolverStatus*)
5 last objective value
6 lower bound of a MIP solution, if not optimal
7 upper bound of a MIP solution, if not optimal
8 Memory allocation for data in LPL
9 Memory allocation for string data
10 Memory allocation for model constraints
11 Number of variables
12 Number of binary variables
13 Number of integer variables
14 Number of constraints
15 Number of variables + constraints
16 1=Source is encrypted, otherwise not

GetParamS(a) Back to index

Return a global parameter as string

a : an integer 0 . . . 15

Return Value : a string, the global parameter

Example :

```
a:=GetParamS(8)   Result : "6.64" (LPL Version)
a:=GetParamS(4)   Result : "OPTIMAL" (Solver status)
```

Note : This function returns various global information of the model and LPL depending on the parameter *a* (see model learn24[20] for an example) :

0	filename of the LPL model
1	same as 0 but with the folder path
2	compiler switches used
3	problem type name
4	solver status name
5	APL parameter
6	folder paths of LPL
7	working directory
8	Version of LPL
9	execution folder
10	last log message
11	(not documented)
12	LPL licence package and due date
13	Database output option
14	Get SolverList
15	(not documented)
16	(not documented)

Any other value of parameter *a* returns a empty string

GetProblemType()

Back to index

Return problem type

Return Value : an integer between 0 and 11

[20] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn24

Example :

`a:=GetProblemType()` Result : $a = 1$ (the model is an `LP`

Note : LPL automatically recognizes the *problem type* when running and before a solver is called. It recognizes the following types:

0	NONE	no variables or constraints are defined
1	LP	linear program
2	MIP	mixed integer (linear) problem
3	QP	quadratic problem (obj has $x'Qx$, Q semi-definite)
4	iQP	quadratic integer problem
5	QCP	quadratic constraint as $x'Qx$, (convex constraints)
6	iQCP	quadratic constraints with integer vars
7	SOC	(no longer used)
8	iSOC	(no longer used)
9	NQCP	quadratic constraints (nonconvex)
10	iNQCP	quadratic constraints with integer vars (nonconvex)
11	NLP	non-linear optimization problem
12	iNLP	non-linear integer opt. problem
13	PERM	permutation problem

The function returns the *number* of the problem type (first column). If one needs the *name* (second column), a call to the function *GetParamS(3)* has to be used (see GetParamS). Se also the function SetSolverList that takes a string parameter with 14 entries separated by commas. The entries define the solver that should solve that type of problem.

GetSolverStatus() Back to index

Return the solver status

Return Value : an integer between 0 and 7

Example :

`a:=GetSolverStatus()` Result : $a = 7$ if the model is optimal
`a:=GetSolverStatus()` Result : $a = 2$ if the model is infeasible

Note : The solver status is a number between 0 and 7 as follows :

0	NOT SOLVED	no solver was called (yet)
1	UNBOUNDED	the problem is unbounded
2	INFEASIBLE	the problem is infeasible
3	ABORTED	the solver was aborted
4	TROUBLES	the solver had troubles
5	HEURISTIC	the solution is not necessary optimal
6	NORMAL	the solver terminated normally
7	OPTIMAL	an optimal solution was found

The status *NORMAL* is basically for non-linear solver. The solver terminated normally without proving optimality.

GetValue([r,] a) Back to index

Return a numerical value of reference r (a variable or a constraint)

r : an entity identifier as string (variable, or constraint)

a : an integer in the range: $0 \ldots 8$

Return Value : double

Example :

`a:=GetValue(x,3)`	Result : is dual price of variable x
`a:=GetValue(R,4)`	Result : slack (rhs-lhs) of constraint R

Note : Various values connected to variables and constraints are returned, for example, dual values, reduced cost, lower/upper bounds etc., depending on the parameter a. If the entity parameter r is missing then the attribute of the focused entity is returned (see function SetFocus (See model learn20[21] and learn27[22] for an example). The parameter a can have the following values :

0	the value is returned
1	the lower bound is returned
2	the upper bound is returned
3	the dual value is returned
4	the lrhs (slack) is returned

[21] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn20
[22] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn27

5	the lhs is returned
6	the rhs is returned
7	the lower range is returned
8	the upper range is returned
9	the value of the (first) slack variable (see Sl() function)
10	the value of the (second) slack variable (see Sl() function)
11	(used internally: floor of a big-M expression)
12	(used internally: ceil of a big-M expression)

The parameter values a of 0 to 3 are for variables, the values from 3 to 8 are for constraint. The value 3 returns the dual values for variables and the reduced cost for constraints. An example is given in model learn20[23], see also learn39[24]

GetValueS([r,] a) Back to index

Return a string value of reference r

r : an entity identifier as string (string parameter)

a : an integer 0

Return Value : a string

Example :

`a:=GetValueS(s,0)` Result : the value string of entity s is returned

Note : Various string values connected to sets or parameters are returned, for example, the elements of a set or the values of string parameters, depending on the parameter a. If the entity parameter r is missing then the attribute of the focused entity is returned (see function SetFocus (See model learn20[25] and learn27[26] for an example). The parameter a can have the following values :

0	the (string) value is returned

[23] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn20
[24] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn39
[25] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn20
[26] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn27

The parameter values a is always 0 (this may be extended in the future). For a basic set it returns the corresponding element name, for a string parameter it returns the corresponding string in the data table.

Contains(x,i) Back to index

Return true if alldiff variable xk,i contains i in subset k

x : is an partitioned alldiff variable

i : is an item within the alldiff variable

Return Value : boolean.

Example :

`Index(x[k],i)` Result : return the subset k

Note : Given an ordered partitioned permutation variable $x_{k,i}$, then the function returns the position of item i in the subset $x[k]$. (see also The function .

Log(x) Back to index

Return the e log of x ($\ln x$)

x : a double expression, x must be positive.

Return Value : a double

Example :

`a:=Log(10)` Result : $a = 2.3026$

Max(x,...) Back to index

Return the maximum

x,... : a list of double expression.

Return Value : a double

Example :

`a:=Max(10,5,4,12,5)` Result : $a = 12$

Min(x,...) Back to index

Return the minimum

x,... : a list of double expression.

Return Value : a double

Example :

`a:=Min(10,5,4,12,5)` Result : $a = 4$

NextFocus(a) Back to index

Sets the focus the the next entity of genus a

a : an integer expression.

Return Value : an integer (0 or 1, true or false)

Example :

`NextFocus(3)` Result : jumps to the next variable entity

Note : The parameter a must be an integer defining an genus (see genus list genus). The function jumps within the model to the next entity of genus a. The function returns true if the jump succeeded (there was one entity) or false if the end of the model has been reached. For an example see model learn27[27].

[27] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn27

NextPosition[(i)] Back to index

Jumps to the next data entry of the focused entity

i : an integer expression (0 or 1) (if missing then i=1, 0 means full traversal, 1 means sparse traversal.

Return Value : a integer (0 or 1, true or false) (default is 1)

Example :

`a:=NextPosition()`	Result : jump to the next data entry of focus.
`a:=NextPosition`	Result : Same as *NextPosition*()
`a:=NextPosition(1)`	Result : Same as *NextPosition*() (sparse version).

Note : This function sets an internal pointer to the next data entry within a entity table. This data can now be retrieved with the function *GetValue*. For an example see model learn27[28].

Now() Back to index

Return actual date and time

Return Value : a double

Example :

`a:=Now()` Result : $a = 41502.70815$
This corresponds to the date: `16.08.2013 16:59:43`

Note : Use the function DateToStr to translate the double into a string.

OSCall(s) Back to index

Call an OS program s

s : a string

[28] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn27

Return Value : None

Example :

`OSCall('notepad.exe')` Result : the editor *notepad* is opened

Note : LPL calls a program and waits until it terminates.

Ord(c) Back to index

Return Unicode code of a char *c*

c : a string of length 1, a character.

Return Value : the Unicode code number, an integer

Example :

`a:=Ord('a')` Result : $a = 97$
`a:=Ord(' ')` Result : $a = 32$

Read(s,e,...[,E]) Back to index

Read data from file.

s : a string, the source from where to read an what part to read. This parameter is a concatenation of several substrings as follows (explained below):

```
s = [*] [prefix] [type] filename [, table] [, delim] [, ignored chars]
```

e,... : a comma-separated list of `<destination=source>` to be read.

E : an optional arbitrary expression which is executed each time the function *Read* has read a line (or a record).

Return Value : None

Examples :

`Read('text.txt',a=2)`
Result : read from file 'text.txt' the 2nd token on the first line into a

`Read{i}('text.txt',b=3)`
Result : read the third column from file into b

`Read{i}('text.txt',a,b,c)`
Result : read the first 3 columns from file into a, b, c

`Read{i}('-:txt:text.xls',a,b,c)`
Result : read the first 3 columns from file as a text file into a, b, c, if file exists

Note : The function *Read* is a powerful mean to read from text files, databases, Excel sheets and snapshots in various forms. Each *Read* instruction opens a file, reads data from it, and closes it automatically. There are no open or close file instruction. Here only the syntax is explained. To get a broader view of reading/writing see Chapter 8.

The first string parameter **s** is now explained. The first part (**[*] [prefix] [type]**) of the first parameter **s** is the **file type**. It specifies from what type of file to read (text files, databases, Excel sheets or snapshots). Normally, if the **type** is missing, the LPL parser derives the file type from the filename extension (default). If it is **.mdb** or **.db**, then the file type is a *database*, if the extension is **.xls** the file type is a *Excel spreadsheet* (only on Windows), if it is **.sps** the file type is *a snapshot*, in all other cases the file is considered as a *text file*. The **[prefix] [type]** – if used – overrules this default. The **[type]** can be one of the following:

type	= `'txt:'`	read from a text file
	= `'rdb:'`	read from a database using specific driver libraries
	= `'xls:'`	read from an Excel spreadsheet
	= `'sps:'`	read from a snapshot generated by LPL before
	= empty	the default is used (file name extension)

The **[prefix]** is empty or '-:'. If it is '-:' it means that if the file does not exist then nothing is read without giving an error, a warning is given in the log file.

The very first character can be a @ (an at character). If the star is present than LPL records from a a previous read of the same read instruction where the file pointer is, and can than continue to read from that location. This is only relevant for text files, see model tutor07d[29].

[29] https://lpl.matmod.ch/lpl/Solver.jsp?name=/tutor07d

If the first character (after a @) is a # (a hash character) then the read will read a block token by token into a single datacube, see model tutor07e[30].

The next part of parameter **s** is the **filename**. It is followed by a optional comma and a **table**. Depending on the file type the **table** has different syntax:

- For **text-files**, the **table** begins with the percent sign % followed by a positive integer, the **block number**, an optional semicolon with an integer **number of skipping lines** and an optional **block begin and block end delimiter**, an list of token delimiting chars, and a list of ignored chars. Example:

  ```
  'text.txt,%2;1:Table:End,\t\n ,]-'
  ```

  This example reads a file block from file 'text.txt' beginning at the second occurrence of 'Table' (beginning a line) and ending with 'End', 'Table' or and end of file. It skips the first line after 'Table'. Tabs, new line and spaces are token separators, and "]" and "-" are ignored chars (that is, they are translated into spaces). (see Chapter 8 for more detailed information). It is not necessary to repeat the **filename** for subsequent *Read* instructions. For an example see model tutor07[31].
- For **databases**, the **filename** consists of the *connection string* extended by the database file name and the **table** consists of a database table name or an SQL statement *SELECT query*. For an example see model tutor07b[32].
- For a **Excel spreadsheet**, the **table** consists of a *range* specification of the sheet. For an example see model tutor07a1[33].

The comma separated list of a <**destination**=**source**> consists of:

- a **destination**, that is, a model entity name that receives the data.
- an equal sign, suggesting that there is a transfer of data from the source to the destination.

[30] https://lpl.matmod.ch/lpl/Solver.jsp?name=/tutor07e
[31] https://lpl.matmod.ch/lpl/Solver.jsp?name=/tutor07
[32] https://lpl.matmod.ch/lpl/Solver.jsp?name=/tutor07b
[33] https://lpl.matmod.ch/lpl/Solver.jsp?name=/tutor07a1

- a **source**, an indication from where the data are read. Basically, the **source** are columns (or fields in databases). In text files, the columns are numbered from left to right beginning with 1. The source can be dropped together with the equal sign, if the data are read beginning with column 1. In databases the **source** must always be a field name.

In text files, the concept of a *column* depends on the *delimiters*. By default, *tabs*, *spaces* and the *comma* are considered as delimiters, whereat several consecutive spaces (not tabs) are considered as a single delimiter. Note, however, that a tab followed by a blank space is considered as *two* delimiters. (The model learn12[34] explains how to read relations (indexed sets) from a text file.)

Rgb(r,g,b)

Back to index

Returns a Rgb color code

r : integer in the range: 0..255, red component

g : integer in the range: 0..255, green component

b : integer in the range: 0..255, blue component

Return Value :

Example :

`a:=Rgb(255,0,0)`	Result : *a* defines the color red
`a:=Rgb(0,0,0)`	Result : *a* defines color black
`a:=Rgb(255,255,255)`	Result : *a* defines color white

Note : The function *Rgb* is used basically in the *Draw* library.

Rnd(x,y)

Back to index

Returns an uniform random number in range [x...y]

x : double, lower limit

[34] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn12

y : double, upper limit, (not included)

Return Value : a double

Example :

`a:=Rnd(5,10)` Result : a is a number between 5 and 10

Note : The function returns an uniformly distributed random number in the range $[l \ldots u[$. The upper limit is excluded from the range. The random seed can be set with the function SetRandomSeed.

Rnde(m) Back to index

Returns an exponentially distributed random number

m : double, the mean of the distribution

Return Value : a double

Example :

`a:=Rnde(5)` Result : a is a random number

Note : The function returns a exponentially distributed number with mean m. The random seed can be set with the function SetRandomSeed.

Rndn(m,s) Back to index

Returns an normally distributed random number

m : a double, the mean

s : a double, the standard deviation

Return Value : a double

Example :

`a:=Rndn(0,1)`
Result : a is normal random number with $\mu = 0$ and $\sigma = 1$
`a:=Rndn(-5.5)`
Result : $a = 5.5$

Note : The function returns a normally distributed number with mean m and standard deviation s. The random seed can be set with the function SetRandomSeed.

Round(x[,y])

Back to index

Returns a rounded integer of x

x : a double, the number to be rounded

y : an integer, the number of decimals (default=0)

Return Value : an integer

Example :

`a:=Round(23.4)`	Result : $a = 23$
`a:=Round(5.5)`	Result : $a = 6$
`a:=Round(5.5549,-2)`	Result : $a = 5.55$

Note : The function rounds a fractional number to the closest integer, 0.5 is rounded up to the next higher integer. If a second argument$y$ is used then it rounds to that many decimals. The argument y must be negative for a positive number of decimals; positive numbers of y mean to round to tenner, hundred, etc.

SetFocus[(r)]

Back to index

Sets the focus to entity reference r.

r : an entity name (or empty parameter list)

Return Value : None

Example :

`SetFocus(X)` Result: the focus is set to entity X
`SetFocus()` Result: the focus is set to the very beginning of the model
`SetFocus` Result: same as ?SetFocus()?

Note : *SetFocus* sets an internal pointer to an entity. Calling this function without parameter sets the focus to the very beginning of the LPL model – before the first `model` declaration. The focus is then used in the other functions: GetAttr, GetName, GetValue and GetValueS. The focus can also be changed by NextFocus.

SetPaths(c) Back to index

Add file path list to the directory list (see 9.3).

c : a string, the path list to be added

Return Value : None

Example :

```
SetPaths('c:/mypath;d:/myotherpath)
```
Result: the two paths are added to the directory list (see 9.3)

Note : The global directory list of LPL contains always two paths: the path to the model called, (the working directory and the path to the executable (the EXEDir), (typically *lplw.exe*. Using the function *SetPaths* the directory list can be extended, and all files are searched automatically in this list, with the exception of the license file *lpl.lic*, the message file *lplmsg.txt*, the *lpl.ini* file, and the *lpl.file.policy* which must be in the EXEDir (directory of the executable).

SetRandomSeed(x) Back to index

Sets the random seed to x

x : an integer, the random seed

Return Value : None

Example :

`SetRandomSeed(0)` Result : set the random seed to "random"
`SetRandomSeed(1)` Result : set the random seed to 1

Note : Setting a random seed means to control the generation of random numbers. With the same random seed, the same sequence of random number will be generated each time. The argument 0 is special. The random sequence is different each time. It is as if the random seed is chosen at random each time. Note that by default the random seed is set to 1.

SetServer(s) Back to index

Sets the Internet address of the LPL server

s : a string, an Internet server address

Return Value : None

Example :

```
SetServer('https://lpl.unifr.ch/lpl/');
SetSolver('l,,Internet');
```
Result : The LPL Internet server is set to this address

Note : Together with the function *SetSolver*, the model is automatically send to the LPL Internet server where it is solved and the result is then sent back to the LPL client. In this case, *SetSolver* must be called with the parameter `'l,,Internet'`. The Server chooses an appropriate solver that will solve the problem. The solution is sent back to the LPL client using the LPX-file.

SetSolver(s[,t]) Back to index

Sets the solver and its parameters

s : a string, the *solver interface parameters* (SIP) (see 9.2.1)

t : a string, the solver options

Return Value : None

Example :

```
SetSolver(gurobiLSol,'timelimit=100;VarBarnch=1')
```

Result : Sets the gurobi solver and passes a time limit of 100secs as well as the V

```
SetSolver('l,,Internet')
```

Result : Sets the Internet solver

Note : The parameter `gurobiLSol` is defined in the file *lplcfg.lpl*. The two parameters s and t are explained in detail in Chapter 9.

SetSolverList(s) Back to index

Sets a list of solvers to s

s : a string, a solver list

Return Value : None

Example :

```
(SetSolverList(',gurobi,gurobi,gurobi,gurobi,
  gurobi,gurobi,lindoLSol,lindoLSol,gurobi,gurobi,conopt,knitroLSol,tabuSol');
```

Result : Assign the appropriated solver to each problem type.

Note : There are 14 *problem types* that LPL can automatically detect (see GetProblemType). The string parameter s consists, therefore, of 14 comma separated solver defined in the file *lplcfg.lpl*. With example above, if LPL detects that the problem is of type *iNLP* then the solver *knitroLSol* will be called to solve it. If the problem type is *MIP* then the solver *gurobi* will be called –because it is the 4-th entry in the list (problem type for MIP is 4). Note that the list begins with a comma. That is, the first entry is empty (no solver solves a problem of type *NONE*). Note also that the id *gurobi* or *knitroLSol* must be defined as string parameters (normally in the file lplcfg.lpl).

SetWorkingDir(s) Back to index

Sets the working directory to s

s : a string, a directory path.

Return Value : None

Example :

```
SetWorkingDir('c:/lpl')
```
Result :set the model directory path to `'c:/lpl'`

Sin(x) Back to index

Return the Sinus of x

x : a double

Return Value : a double

Example :

```
a:=Sin(3.14159/2)
```
Result : $a = 1$ (Sinus of 90^o)

Sl(a[,b][,c]) Back to index

add a slack variable to a constraint.

a : an upper bound of the new variable (can be an expression)

b : the weight (penalty) in the objective function for that variable (can be an expression), default 1 if this argument is missing.

c : the role parameter(expression): if evaluates to non-zero then the slack variable is frozen to the the actual value, otherwise (if zero) it is unfrozen again.

Return Value : None

Example :

```
Sl(10,100)
```
Result: a variable with upper bound 10, penalty of 100

Note : The function *Sl* implements goal programming. It can be used in a constraint expression or in the objective function. Using in a constraint, it adds a slack variable to the constraint with the goal to "soften" the constraint. Several *Sl* functions can be added in a constraint expression. LPL generates a variable with the same dimension as the constraint and a name CONSTNAME_x. A term is then automatically added to the objective function that penalties the corresponding variable. A small model example is given in learn39[35]. The third parameter c is important in multiple optimizations. After a first optimization this parameter can be set to a non-zero value which means that the corresponding value of the slack is fixed (frozen). It is becoming a fixed variable, that is, the frozen attribute of the variable is set.

Note also that the compiler switch 'p' adds an *Sl* function to all constraints with the goal to "soften" *all* constraints. This can be used to find infeasibilities.

Another use of the function *Sl* is in the objective function: $Sl(C[,n])$, the first parameter must be a constraint name and the second (optional) is a integer. If the second is missing, it is zero. A positive integer n make reference to the n-th occurrence of an *Sl* in that constraint, zero means "all". So, the use in an objective function adds the weighted slack variables of that constraint. An small example is given in model goalpr2[36], another model is library[37].

Sort(a,b[,n])

Back to index

Sort data

a : an identifier (vector to be sorted)

b : an identifier (the resulting permutation vector)

n : integer 1: sort descending else ascending

Return Value : None

Example :

[35] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn39
[36] https://lpl.matmod.ch/lpl/Solver.jsp?name=/goalpr2
[37] https://lpl.matmod.ch/lpl/Solver.jsp?name=/library

`Sort(a,b)` Result: b is the resulting permutation

Note : See the model example learn06[38].

Sos1(c) Back to index

Defines SOS1 (special ordered set of type 1) constraint

c : a list of variable identifiers

Return Value : None

Example :

```
constraint S1: Sos1(x,y)
```
Result : Variable `x` and `y` build a SOS1

Note : At most one of the variables in the list is 1 – all others are zero.

Sos2(c) Back to index

Defines SOS2 (special ordered set of type 2) constraint

c : a list of variable identifiers

Return Value : None

Example :

```
constraint S2: Sos2({i}x)
```
Result : All variables `x{i}` build a SOS2

Note : At most two consecutive variables in the list are different from zero.

Split(s,c,t,...) Back to index

Splits a string s into parts separated by a char c

[38] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn06

s : a string or an expression returning a string, it is the string to split

c : the splitting character

t , ... : either a list of parameter identifiers or a parameter index over (see below)

Return Value : 1 if successfully executed else 0

Example :

```
string parameter s; a; b; c; d;   Split(s,',',a,b,c,d);
```
Result : Split s into a,b,c,and d.

```
string s{i}; a{i}; b{i}; c{i}; d{i};  {i} Split(s,',',a,b,c,d)
```
Result : Splits s for each i

Note : See the model example learn25[39].

Sqrt(x) Back to index

Return the square root of x

x : a double

Return Value : a double

Example :

```
a:=Sqrt(49)
```
Result : $a = 7$

```
a:=Sqrt(2)
```
Result : $a = 1.4142$

Strbin(s) Back to index

Return an integer from binary string

s : a string (containing 0 and 1)

Return Value : an integer

[39] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn25

Example :

```
a:=Strbin('1101')
```
Result : $a = 13$

Strbin0(n) Back to index

Return a binary string from n

n : an integer

Return Value : a string

Example :

```
a:=Strbin(13)
```
Result : $a =' 1101'$

Strdate(s[,c,n]) Back to index

Return a date of a string s

s : a string

c : an char

n : an integer

Return Value : an integer, the date.

Example :

```
a:=Strdate('2018-12-20')
```
Result : $a = 43454$

```
a:=Strfloat('1,3,5,8',',',4)
```
Result : $a = 8$

```
a:=Strfloat('1.1 3.4 56.1',' ',3)
```
Result : $a = 56.1$

Note : The function converts a string s to a date and returns an integer. Note that the string must be in the format yyyy-mm-dd where yyyy is the year, mm is the month and dd ist the day. The returned integer counts the days since 1899-12-31 which day 1. If the string s consists of several substrings separated by a character c then the function also can extract the n-th occurrence from the substring and converts this extracted substring into a date (see model learn26[40]).

[40] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn26

Strfloat(s[,c,n])

Back to index

Return a float of a string s

s : a string

c : an char

n : an integer

Return Value : an integer, the length of s.

Example :

`a:=Strfloat('23.25')`	Result : $a = 23.25$
`a:=Strfloat('1,3,5,8',',',4)`	Result : $a = 8$
`a:=Strfloat('1.1 3.4 56.1',' ',3)`	Result : $a = 56.1$

Note : The function converts a string to a float and returns the float. If the string s consists of several substrings separated by a character c then the function also can extract the n-th occurrence from the substring and converts this extracted substring into a float (see model learn26[41]).

StringToSet(s,p)

Back to index

Links a set s to a (string) parameter p

s : a set identifier

p : a (string) parameter identifier, (can also be an numerical parameter)

Return Value : None

Example :

`StringToSet(i,a)`	Result : Link `i` with `a`

[41] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn26

Note : The parameter a must by indexed over i. In output – especially in $Writep$ the string a_i replaces the set name i. For an example see the model learn23[42]. Note also the parameter a can also be a numerical parameter it will then be translated into a string and treated as a string in the output.

Strlen(s) Back to index

Return the length of a string s

s : a string

Return Value : an integer, the length of s.

Example :

```
a:=Strlen('abcd')
```
Result : $a = 4$

Note : See model learn26[43].

Strpos(s1,s) Back to index

Return the position of a substring $s1$ within a string s

s1 : a string, (substring)

s : a string

Return Value : an integer, the position of the first character of $s1$ in s if $s1$ is a substring of s, otherwise return zero.

Example :

```
a:=Strpos('bc','abcde')
```
Result : $a = 2$

Note : See model learn26[44].

[42] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn23
[43] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn26
[44] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn26

Strreplace(s,s1,s2) Back to index

Return the string that replaces the substring $s1$ by $s2$ in s

s : a string, (substring)

s1 : a string

s2 : a string

Return Value : a string, the string s but where substring $s1$ is replaced by substring $s2$.

Example :

`a:=Strreplace('abcdef','ab','cc')` Result : $a =' cccdef'$

Note : See model learn26[45].

Strreverse(s) Back to index

Return the reverse string of s

s : a string

Return Value : a string

Example :

`a:=Strbin('abcdef')` Result : $a =' fedcba'$

Strsub(s,b,l) Back to index

Returns a substring of s of length l beginning at b

s : a string

b : an integer

[45] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn26

l : an integer

Return Value : a string

Example :

`a:=Strsub('abcdefg',2,3)` Result : a ='bcd'

Note : See model learn26[46].

Trunc(x) Back to index

Return the truncated integer of x

x : a double

Return Value : an integer

Example :

`a:=Trunc(-5.7)` Result : $a = -5$
`a:=Trunc(5.8)` Result : $a = 5$

Note : The function cuts the fractional part of the number

Unfreeze(c,...) Back to index

Activate constraints or unfix variables c,...

c,... : a list of identifiers, defining parameters, sets, variables, or model names.

Return Value : None

Example :

`Unfreeze(a,b)`
Result : remove the `frozen` attribute for a and b
`Unfreeze(mymodel)`
Result : remove the `frozen` attribute of model *mymodel*

[46] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn26

Note : See also Freeze. *Unfreeze* function removes the `frozen` attribute of a list of entities, *Freeze* sets the attribute.

Write([s,]f,e,...) Back to index

Write data to file *s*

s : a string, the destination to what file to write. This parameter is a concatenation of several substrings as follows (explained below):

```
s = [prefix] [type] filename [, table] [,templ[,report[,band]]]
```

f : a string, the formatting string

e,... : a (possibly empty) list of <**destination**=**source**> to be written.

Return Value : None

Example :

```
Write('written text\n')
```
Result : Writes a line to the default write file

```
Write('text.txt','Hello %s!\n, 'World')
```
Result : write 'Hello world!' to file *'text.txt'*

Note : The function *Write* is a powerful mean to write to text files, databases, Excel sheets and snapshots in various forms and formats. It also is a mean to generate sophisticated and professional reports. Each *Write* instruction opens a file, writes data to it, and closes it automatically. There are no open or close file instructions in LPL. Here only the syntax is explained. To get a broader view of reading/writing see Chapter 7.

The string parameter **s** is now explained. The first part of the parameter of **s** is the **prefix** – specifying how to write to a file. It can be the following:

prefix =	`'&:'`	create a new database file (deleting an existing one)
=	`'*:'`	create a new table inside the database file
=	`'+:'`	add records to the table, append mode for text file
=	`'-:'`	remove all records before adding new records
=	`'1:'`	write only if file is empty (normally a header line to a text file)

One can also use the specification `'-:'` for text files, which means that the file, if it exists, is first deleted before the writing takes place and `'+:'` for text files means append mode. The two other specifications of prefix (`'&:'` and `'*:'`) have no meaning for text files.

The second part of the parameter **s** is the **type** – specifying to what type of file to write. Normally, the LPL parser derives the file type from the **filename** extension. If it is **.mdb** or a **.db** the file type is a *database*, if it is **.xls** the file type is an *Excel spreadsheet*, if it is **.sps** the file type is a *snapshot*, in all other cases the file is considered as a *text file*. The **type** – if used – overrules this default. The **type** can be one of the following (same as in the *Read* call):

type	= `'txt:'`	write to a text file
	= `'rdb:'`	write to a database file using specific drivers
	= `'xls:'`	write to an Excel spreadsheet
	= `'sps:'`	write to a snapshot, LPL's internal file format.
	<empty>	the default is used

The next part in parameter **s** is the **filename**, followed (for database writings) by a comma and a **table** specification.

- For **text-files**, there is no **table** indication. It is not necessary to repeat the **filename** for subsequent *Write* instructions. For an example, see model tutor08[47].
- For **databases**, the **filename** consists of the *connection string* extended by the database file name and the **table** consists of a database table name. For an example see model tutor08b[48].
- For a **Excel spreadsheet**, the **table** consists of a *range* specification of the sheet. For an example, see model tutor08a1[49].

Note that (for text-files) the parameter **s** can be missing. In this case, the data are appended to the file, defined in a previous *Write* instruction. If no previous filename was given or if the parameter is 'nom', the file is the NOM-file, that is, the file with the same name as the model extended with '.nom'. The NOM-file is deleted at the beginning of each new run.

The next optional part in parameter **s** is **templ**. This is only for reports and can only be used when writing to databases. It is the name of the FastReport template filename. The filename has automatically

[47] https://lpl.matmod.ch/lpl/Solver.jsp?name=/tutor08
[48] https://lpl.matmod.ch/lpl/Solver.jsp?name=/tutor08b
[49] https://lpl.matmod.ch/lpl/Solver.jsp?name=/tutor08a1

and extension of **.fr3**. If templ is a star (*), then the name is the same as **table** (the previous string).

The fifth optional part in parameter **s** is **report**. It is the report filename. On its syntax see 8.2.3.

The final part in parameter **s** is **band**. It is a character specifying the type of report band that must be generated in the report template. This parameter is almost never used therefore it is not documented.

The parameter **f** is only used for text files, and specifies the format of the data to be written. All the format specifiers are explained in Chapter 7. A model example is tutor08e[50].

The comma-separated list of a <**destination**=**source**> consists of:

- a **destination**, that is, the location (column or field name) where to write the data.
- an equal sign, suggesting that there is a transfer of data from the source to the destination.
- a **source**, an expression that is evaluated and the result is the object to be written. Basically the **destinations** are columns (or fields in databases). In text files, the columns are numbered from left to right beginning with 1. They can be dropped together with the equal sign. In databases the **destination** must be a field name.

Writep(r,...['sum'])

Back to index

Write table data to file

r,... : a list of identifiers, defining parameters, sets, variables, or constraint names.

Return Value : None

Example :

`Writep(a,b)`	Result : Write the table *a* and *b* to a file
`Writep(a,b,'sum')`	Result : Write the table *a* and *b* to file with vertical/horizontal

[50] https://lpl.matmod.ch/lpl/Solver.jsp?name=/tutor08e

Note : The file name must be specified by a *Write* function call. If not defined then the tables are written to the NOM-file. The last argument can be `'sum'`, then the tables are printed with all subtotals in each dimension.

3.13.3 Draw Library

This section explains the Draw library functions.

Draw.Animate([id,t,id1,] dur,beg,fr,to,fr1,to1,fr2,to2])

Index

Draw an arc

id : string: id to object to be animated

t : string, attribute

id1 : string: identifier of the animate object

dur : integer: duration in secs

beg : integer: begin of animation

fr : integer: from (0 is initial x-position of id) (x-direction)

to : integer: to (difference to end point)

fr1 : integer: from (y direction)

to1 : integer: to y-direction

fr2 : integer, for "rotate" cy from

to2 : integer: for "rotate" cy to

Return Value : None

Example :

```
Draw.Rect('obj','label',52,53,60,60)
Draw.Animate('obj','translate','anim-id',2,3,0,50,0,60);
Draw.Animate('obj','translate','anim-id',3,6,50,100,60,60);
```

Result : The object with id "obj" (which a rectangle together with an inline text 'label') moves from its origin location (that is from (52,53) 50 right and 60 down. This animation starts after 3secs and lasts 2secs. In a second animation the same object then moves horizontally by 50, and it starts after 6secs an lasts 3secs. see model learn50[51]. Note that the 'obj' is a combined object consisting of a rectangle and a label (a text object).

Note : The attribute t can be 'translate', 'scale', 'rotate', 'skewX', 'skewY'

Draw.Arc([id,t,]x,y,r,a1,a2[,c2,w,o2,a]) Index

Draw an arc

id : string

t : string, text to be placed

x : double, x-center

y : double, y-center

r : double, radius of the circle

a1 : double, angle where the arc begins (degree)

a2 : double, angle where the arc ends (degree)

c2 : integer, stroke color, default black (0)

w : integer, width of the arc line, default 1

o2 : double [0,1], stroke opacity, default 1

a : double, position of the text t default 0.5

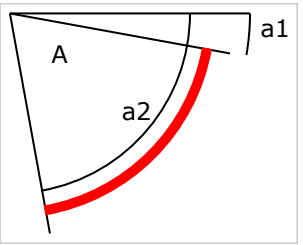
Return Value : None

[51] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn50

Example :

```
Draw.Arc('A',0,0,100,-10,-80,3,5)
```

Result : An arc (red) is drawn with circle center $(0, 0)$, radius 100, angle 10 to 80, color red (3), stroke width is 5.
Note that angle are measured in clockwise direction.



Draw.Arrow([id,t,]x,y,x1,y1[,z,c2,w,o2,a]) Index

Draw an arrow

id : string

t : string, text to be placed

x : double, starting x-position

y : double, starting y-position

x1 : double, ending x-position

y1 : double, ending y-position

z : double, position of arrowhead, default: 0

c2 : integer, stroke color, default black (0)

w : integer, width of the arc line, default 1

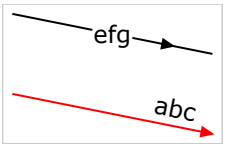
o2 : double [0..1], opacity, default 1

a : double, position of the text t, default: 0.5

Return Value : None

Example :

```
Draw.Arrow('efg',0, 0,100,20,20);
Draw.Arrow('abc',0,40,100,60, 0,3,1,-2,-.9);
```



Result : Two arcs is drawn. The first sets the text in the middle of the line ($a > 0$ and puts the arrowhead 20 pixels inside the line.
The second arc set the arrowhead at the end of the line $z = 0$, color is 3 (=red), arrow width is $w = 1$, opacity is default ($o2 = -2$, and text is off the line $a < 0$.

Note : The last parameter a can be positive: text is marked horizontally *on* the line. $a = 0.001$ means: close to the beginning (x, y), $a = .99$ means: close to the ending $(x1, y1)$. Same for negative values: if $a = 0.001$ then the text is placed close to (x, y). If $a < 0$ then the text is not place *on* the line, but the text follows the line from left to right. For a model example see xDrawLine[52] and xDrawAll[53].

Draw.CArrow([id,t,] x,y,x1,y1,d [,z,c2,w,o2,a]) Index

Draw a curved arrow

id : string

t : string, text to be placed

x : double, starting x-position

y : double, starting y-position

x1 : double, ending x-position

y1 : double, ending y-position

d : double, bend factor

z : double, position of arrowhead

c2 : integer, stroke color, default black (0)

w : integer, width of the arc line, default 1

[52] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawLine
[53] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawAll

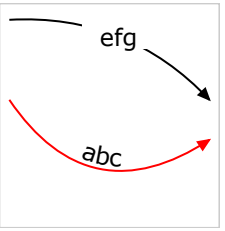
o2 : double [0,1], stroke opacity, default 1

a : double, position of the text t default 0.5

Return Value : None

Example :

```
Draw.CArrow('efg',0,0,100,40,25);
Draw.CArrow('abc',0,40,100,60,-50,0,3,1,-2,-.5)
```



Result : Two bent arcs are drawn. The bend factor d of the first is 25.
The bend factor of the second is -50.

Note : Parameter a, see Arrow. The bend factor d is the larger the more the line is curved. For a model example see xDrawLine[54] and xDrawAll[55].

Draw.CLine([id,t,] x,y,x1,y1,d [,c2,w,o2,a]) Index

Draw a curved line

id : string

t : string, text to be placed

x : double, starting x-position

y : double, starting y-position

x1 : double, ending x-position

y1 : double, ending y-position

d : double, bend factor

c2 : integer, stroke color, default black (0)

[54] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawLine
[55] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawAll

w : integer, width of the line, default 1

o2 : double [0..1], stroke opacity, default 1

a : double, position of the text t, default 0.5

Return Value : None

Example :

```
Draw.CLine(0,0,100,100,30)
```
Result : Draw a curved line

Note : The parameters are the same as in Arrow, with exception of the parameter z. For a model example see xDrawLine[56] and xDrawAll[57].

Draw.Circle([id,t,] x,y,ra [,c1,c2,w,o1,o2]) Index

Draw a circle

id : string

t : string, text to be placed

x : double, x-position of center

y : double, y-position of center

ra : double, radius

c1 : integer, fill color, default black (0)

c2 : integer, stroke color, default black (0)

w : integer, stroke width, default 1

o1 : double [0..1], fill opacity, default 1

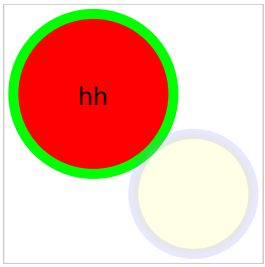
o2 : double [0..1], stroke opacity, default 1

Return Value : None

Example :

[56] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawLine
[57] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawAll

```
Draw.Circle('hh',50,50,40,3,4,5)
Draw.Circle(100,100,30,6,5,5,.1,.2)
```



Result : Two circles are drawn. The first contains a text inside, the fill color is 3 (red) the stroke color is 4 (green), the stroke width is 5.
For the second circle the fill color is 6 (yellow) with an opacity of 0.1, the stroke color is 5 (blue) with an opacity of 0.2.

Note : For a model example see xDrawCircle[58].

Draw.DefFill(id,c1,c2,w,o1,o2) Back to index

Define the fill properties

id : string

c1 : integer, fill color, default black (0)

c2 : integer, stroke color, default black (0)

w : integer, stroke width, default 1

o1 : double [0..1], fill opacity, default 1

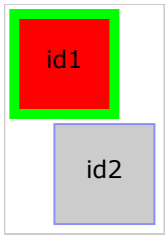
o2 : double [0..1], stroke opacity, default 1

Return Value : None

Example :

[58] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawCircle

```
Draw.DefFill('id1',3,4,5);
Draw.DefFill('id2',0,5,1,0.2,0.9);
Draw.Rect('#id1','id1', 0, 0,50,50);
Draw.Rect('#id2','id2',20,55,50,50);
```



Result : Two rectangles are drawn. The first with the fill properties defined in `id1`. They are: $c1 = 3$ (red) with opacity 1.0, $c2 = 4$ (green) with opacity 1.0 and $w = 5$.
The second is filled with fill properties defined in `id2`. They are: $c1 = 0$ (black= with opacity 0.1, $c2 = 5$ (blue) with opacity 0.9 and $w = 1$.

Note : The argument *id* is an arbitrary string identifier. In the drawing function *Rect* it must be headed with a `#` character. Then the properties are applied to the drawing object. For a model example see xDrawAll[59].

Draw.DefFilter(id,f[,in1,re,x_i]) Back to index

Define a filter

id : string, the id of this definition

fe : integer in the range: [1..24], filter type, see below

in1 :

res :

x_i : double, $i = 1 \ldots 6$, various parameters depending on filter

Return Value : None

Example :

```
Draw.DefFilter('f', 9,11,1, 4);
```
Result : define a GaussianBlur

Note : The parameter *id* is the filter *id*. Several filter-primitives with the same *id* form a single combined filter. The parameter `fe` is the filter type (a number for 1 to 24) of the filter-primitive. The parameter *re* is the resulting filter (an integer number from 0 to 9), that can be used

[59] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawAll

in subsequent primitives in the $in1$ parameter. The parameter $in1$ is an integer from 0 to 9 defining the in-filter of a previously defined primitive with the same re number; or it is a number from 10 to 15. The meaning of these number is defined as follows (see SVG Manual):

10	"SourceGraphic"
11	"SourceAlpha"
12	"BackgroundImage"
13	"BackgroundAlpha"
14	"FillPaint"
15	"StrokePaint"

All other parameters depend on the filter primitive. For a model example see xDrawFilters[60]. The different filter primitives are displayed in the following table.

	fe	name	in*	res
ok	1	feBlend	in	res
ck	2	feColorMatrix	in	res
ck	3	feComponentTransfer	in	res
ok	4	feComposite	in	res
no	5	feConvolveMatrix	in	res
ck	6	feDiffuseLighting	in	res
ck	7	feDisplacementMap	in	res
ck	8	feFlood	in	res
ok	9	feGaussianBlur	in	res
no	10	feImage	in	res
ok	11	feMerge		
ck	12	feMorphology	in	res
ok	13	feOffset	in	res
ok	14	feSpecularLighting	in	res
no	15	feTile	in	res
ck	16	feTurbulence	in	res
ck	17	feDistantLight	azimuth	elevation
ck	18	fePointLight	x	y
ck	19	feSpotLight	x	y
ck	20	feFuncR	type**	Values
ck	21	feFuncG	type**	Values
ck	22	feFuncB	type**	Values
ck	23	feFuncA	type**	Values
ck	24	feMergeNode		

[60] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawFilters

fe	x1	x2	x3	x4	x5	x6
1	in2	mode*				
2	type*	values=				
3						
4	in2	op*	k1	k2	k3	k4
5	order	kernelMatrix	divisor	bias	targetX	targetY
6	sS*	DiffuseConstant	kUL*			
7	in2	scale	xCS*	yCS*		
8	flood-color	flood-opacity				
9	stdDeviation					
10	eRR*	PAR*	xlink:href			
11						
12	op**	radius				
13	dx	dy				
14	surfaceScale	SC*	SE*	kUL*	lC*	
15						
16	baseFrequency	numOctaves	seed	sT*	typ***	
17						
18	z					
19	z	pAtX	pAtY	pAtZ	SE*	LCA*
20	slope	int.	amp.	exp.	offset	
21	slope	int.	amp.	exp.	offset	
22	slope	int.	amp.	exp.	offset	
23	slope	int.	amp.	exp.	offset	
24						

edgeMode*	1=duplicate, 2=wrap, 3=none
eRR* (externalResourcesRequired)	
in*	10=SourceGraphic, 11=SourceAlpha, 12=BackgroundImage, 13=BackgroundAlpha, 14=FillPaint, 15=StrokePaint
kUL* (kernelUnitLength)	
lC* (lightingColor)	
LCA* (LimitingConeAngle)	
mode*	1=normal, 2=multiply, 3=screen, 4=darken, 5=lighten
op* (operator)	1=over, 2=in, 3=out, 4=atop, 5=xor, 6=arithmetic
op** (operator)	1=erode, 2=dilate
PAR* (PreserveAspectRatio)	
preserveAlpha*	1=false, 2=true
SC* (SpecularConstant)	
SE* (SpecularExponent)	
sT* (stitchTiles)	1=stitch, 2=noStitch
type*	1=matrix, 2=saturate, 3=hueRotate, 4=luminanceToAlpha
type**	1=identity, 2=table, 3=discrete, 4=linear, 5=gamma
typ***	1=fractalNoise, 2=turbulence
xCS* (xChannelSelector)	1=R, 2=G, 3=B, 4=A
yCS* (yChannelSelector)	1=R, 2=G, 3=B, 4=A
feDiffuse- and feSpecularLighting use:	use: feDistantLight, fePointLight, feSpotLight
feComponentTransfer	uses: feFuncR, feFuncG, feFuncB, feFuncA
feMerge	uses: feMergeNode

For a deeper understanding on filters, see Filters in SVG.

Draw.DefFont([id,]t,h,c1,c2,fw,fs,fv) Back to index

Define font properties

id : string, the id of this definition

t : string, font name

h : double, font height

c1 : integer, fill color, default black (0)

c2 : integer, stroke color, default black (0)

fw : integer, font weight (1=normal, 2=bold

fs : integer, font style (1=normal, 2=italic

fv : integer, font variant (1=normal, 2=small-caps

Return Value : None

Example :

```
Draw.DefFont('Verdana',8)
```
Result : define *Verdana,8* as standard font

Note : If the parameter *id* is omitted or if it is `'text'` then all text output use this properties. The parameter *t* is the font name. There are 5 basic fonts that always work: *'serif'*, *'sans-serif'*, *'cursive'*, *'fantasy'*, and *'monospace'*. For a model example see xDrawText[61].

Draw.DefGrad(id,sm,x,y,x1,y1,fy,o_i,c_i}) Index

Define a gradient

id : string, the id of this definition

sm : integer, defines spreadMethod, (2=repeat, 3=reflect)

x : double, x-position of linear gradient starting, x-center for radial gradient (cx)

[61] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawText

y : double, y-position of linear gradient starting, y-center for radial gradient (cy)

x1 : double, x-position of linear gradient ending, radius ra for radial gradient

y1 : double, y-position of linear gradient ending, x-focus point in radial gradient (fx)

fy : double, not used in linear gradient, y-focus point in radial gradient (fy)

o_i : double in [0..1], $(i = 1 \ldots 5)$, max. 5 stop-offset values

c_i : double in [0..1], $(i = 1 \ldots 5)$, max. 5 stop-color values

Return Value : None

Example :

```
parameter white:=1; black:=0;
Draw.DefGrad('rGr',-2,-2,-2,-2,-2,-2, 0,white,1,black);
```

Result : define a radial gradient form white to black

Note : If the parameter `id` begins with a character `'r'`, then the definition is a radial gradient, else it is a linear gradient. Note that the parameters `x` to `fy` have different meanings depending on whether the definition is a linear or a radial gradient. For some model examples see xDrawGrad0[62], xDrawGrad1[63], and xDrawGrad2[64]

Draw.DefLine(id,sd1,sd2,sd3,slc,slj,sljl) Index

Define line (path) properties

id : string

sd1 : double, the length of the dash

sd2 : double, the length of the gap between the dashes (stroke-dasharray)

[62] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawGrad0
[63] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawGrad1
[64] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawGrad2

sd3 : double, the initial offset ((stroke-dashoffset) (sd1=-2 means that the attribute is not used)

slc : integer, is the stroke-linecap (-2=butt, 2=round, 3=square)

slj : is the stroke-linejoin (-2=miter, 2=bevel, 3=round)

sljl : is the stroke-miterlimit (default:4)

Return Value : None

Example :

```
Draw.DefLine('line.dash',8,3);
Draw.Line('#dash',690,460,590,265,2,3);
```

Result : Dashed line of length 8,3

Note : For a model example see xDrawAll[65]

Draw.DefTrans(id,t, [x1,y1,z1]) Back to index

Define a transformation

id : string

t : string, transformation type: `^'t'` (translate), `'r'` (rotate),`'s'` (scale),`'sX'` (skewX),`^'sY'` (skewY).

x1 : double, see below

y1 : double, see below

z1 : double, see below

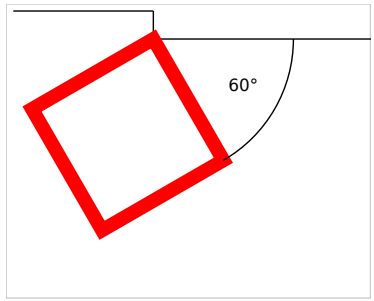
Return Value : None

Example :

[65] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawAll

```
Draw.DefTrans('tr','t',100,20);
Draw.DefTrans('tr','r',60);
Draw.Rect('#tr',0,0,100,100,1,3,10);
```

Result : An rectangle (red) is drawn with left/top $(0,0)$, and size $(100,100)$. It is then translate with $(100,20)$ and rotate 60 degree.



Note : The three parameters $x1$, $y1$, $z1$ are the corresponding numerical parameters for the transformation. For a model example see xDrawTrans1[66]. They are 7 possibilities:

translate(x1,y1)	moves the user coordinate system by (x1,y1)
scale(x1,y1)	moves the user coordinate system by (x1,y1)
scale(x1)	is the same as scale(x1,x1)
rotate(x1,y1,z1)	rotates the user coordinate system by angle x1 around center (y1,z1)
rotate(x1)	is the same as rotate(x1,0,0)
skewX(x1)	skews all x-coordinates by a specified angle x1
skewY(x1)	skews all y-coordinates by a specified angle x1

Draw.Ellipse([id,t,] x,y,rx,ry [,c1,c2,w,o1,o2]) Index

Draw an ellipse

id : string

t : string, text to be placed

x : double, x-center

y : double, y-center

rx : double, x-radius

ry : double, y-radius

[66] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawTrans1

c1 : integer, fill color, default black (0)

c2 : integer, stroke color, default black (0)

w : integer, stroke width, default 1

o1 : double [0..1], fill opacity, default 1

o2 : double [0..1], stroke opacity, default 1

Return Value : None

Example :

`Draw.Ellipse(0,0,20,30)` Result : Draw black ellipse at center $(0, 0)$

Note : For a model example see xDrawCircle[67]

Draw.Line([id,t,] x,y,x1,y1 [,c2,w,o2,a]) Index

Draw a straight line

id : string

t : string, text to be placed on or off the line.

x : double, starting x-position

y : double, starting y-position

x1 : double, ending x-position

y1 : double, ending y-position

c2 : integer, stroke color, default black (0)

w : integer, width of the line, default 1

o2 : double [0..1], stroke opacity, default 1

a : double, position of the text t, default 0.5

Return Value : None

Example :

[67] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawCircle

```
Draw.Line(0,0,100,100)
```
Result : Draw a line $(0,0)$ to $(100,100)$

```
Draw.Line('label',0,0,-100,-200)
```
Result : Draw a line $(0,0)$ to $(-100,-200)$ and label it.

Note : The parameters are the same as in Arrow, with exception of the parameter z. For a model example see xDrawLine[68] and xDrawAll[69], see also DrawClock[70] to show the parameter a.

Draw.Path([id,]t[,x_i,y_i]) Back to index

Draw a path

id : string

t : char, path type

x_i : double, $(i = 1 \ldots 5)$, x-coordinates

y_i : double, $(i = 1 \ldots 5)$, y-coordinates

Return Value : None

Example :

```
Draw.Path('M',10,10)
```
Result : Move the cursor to $(10,10)$

```
Draw.Path('V',20); Draw.Path('H',20);
```
Result : Draw a vertical then a horizontal line

Note : A single *Draw.Path* function call draws only a simple primitive path. However, consecutive calls to this function build a *single* unique path. Each path must begin with a moveto ('M') instruction, if not, LPL will automatically add the path instruction "M 0 0", that is, the path then begins at point (0,0). The path is ended by a call to a path type *'Z'* or *'Y'* or by a call to any other drawing function. The parameter t is a single character indicating the path type. Capital letters are for absolute, lower-case letters are for relative positions. They are defined in the next table as follows:

[68] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawLine
[69] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawAll
[70] https://lpl.matmod.ch/lpl/Solver.jsp?name=/DrawClock

'M','m'	moveto	"M 10 10", move to (10, 10) (without drawing).
		`Draw.Path('M',10,10)`
		this type can also draw a polyline of maximally 5 lines as follows:
		`Draw.Path('M',10,10, 20,30, -10,3, 45,34, 10,10)`

'L','l'	lineto	"L 20 20", lineto (20,20). `Draw.Path('L',20,20)`
'H','h'	hor. lineto	"M 10 10 H 20", moveto (10,10) then lineto (20,10). `Draw.Path('M',10,10); Draw.Path('H',20)`
'V','v'	ver. lineto	"M 10 10 V 20", moveto (10,10) then lineto (10,20). `Draw.Path('M',10,10); Draw.Path('V',20)`
'C','c'	cubic Bézier	"M 0 0 C 0 10 20 20 10 100", draws a curve from (0,0) to (10,100) using the two control points (0,10) and (20,20). `Draw.Path('M',0,0); Draw.Path('C',0,10,20,20,10,100)`
'S','s'	cubic Bézier	"S 15 15 50 50", draws a curve from the actual point to (50,50) using a second control point (15,15) (the first is calculated from a previous path). `Draw.Path('S',15,15,50,50)`
'Q','q'	quad. Bézier	"Q 10 10 50 50", draws a curve from the actual point to (50,50) using the single control point (10,10). `Draw.Path('Q',10,10,50,50)`
'T','t'	quad. Bézier	"T 100 100", draws a curve from the actual point to (100,100) using a calculated control point from a previous Path instruction. `Draw.Path('T',100,100)`
'A','a'	an elliptic arc	"A 5 6 45 0 0 100 100", draws an elliptic arc with radius (5,6) with rotation 45, using shorter path, counter-clockwise, with end point (100,100). `Draw.Path('A',5,6,45,0,0,100,100)`

'B'	MoveTo	(This definition exist only in LPL and is not part of the svg-specifications.) This entry is the same as an 'M' followed by several 'L'. It specifies a path of straight lines
		`{i}Draw.Path('B',x[i],x[i])`
		The previous instruction draws a path from (x_1, y_1) to (x_n, y_n) It is the same as:
		`Draw.Path('M',x[1],y[1]; {i|i>1}Draw.Path('L',x[i],x[i])`

'Y','y'	ends a path	(This definition exist only in LPL and is not part of the svg-specifications.) This entry can be armed with references to path attributes defined in DefFill, DefLine, etc. (the references can also be used in the 'Zz' section, in all other sections additional references have no effect).
		`Draw.Path('#fillid gradid','Y')`
		This type (as well as the 'Zz' type) can also contain the 5 additional attribute parameters as follows
		`Draw.Path('#fillid gradid','Y',c1,c2,w,o1,o2)`

'Z','z'	closepath	(no path parameters) (see also 'Yy' type). `Draw.Path('Z')`

There is a second variant of the Path function. The parameter t can consist of a string containing the whole path as specified in svg. In this case, x1 and y1 define the fill and stroke color, x2 is the stroke width, and y2, x3 are the fill- and stroke-opacity of the path. An example is:

```
Draw.Path('M50,90 C0,90 0,30 50,30 L150,30\

       C200,30 200,90 150,90 z',-1,5,10);
```

This function defines a closed path staring at (50,90), followed by two Bézier curves and a straight line in the middle. For a model example see xDrawPath[71], see also xDrawArcPath[72].

The path instruction can also be used to *define* only a path without drawing it. This is interesting for drawing text paths, for an example see xDrawTextPath[73].

Draw.Picture(t [,x,y,w,h]) Back to index

Load and draw a picture (bitmap)

t : file name of the picture

x : double, left position

y : double, top position

w : double, weight of the picture

h : double, height of the picture

Return Value : None

Example :

```
Draw.Picture('graphic.jpg')
```
Result : Loads the file at $(0, 0)$

Note : The picture is scaled if using the 4 positional parameters.

Draw.Rect([id,t,]x,y,w,h[,c1,c2,w,o1,o2,rx,ry,rot]) Index

Draw a rectangle

id : string

t : string, text to be placed

[71] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawPath
[72] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawArcPath
[73] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawTextPath

x : double, left position

y : double, top position

w : double, width of the rectangle

h : double, height of the rectangle

c1 : integer, fill color, default black (0)

c2 : integer, stroke color, default black (0)

w : integer, stroke width, default 1

o1 : double [0..1], fill opacity, default 1

o2 : double [0..1], stroke opacity, default 1

rx : double, x-rounding of the corners

ry : double, y-rounding of the corners

rot: double, rotation in degree around the center

Return Value : None

Example :

`Draw.Rect(0,0,50,40)` Result : Draw a black rect at $(0, 0)$

Note : For a model example see xDrawRect[74].

Draw.Save(t)

Back to index

Save the drawings to a file t

t : string, the file name, the extension is automatically `.svg`

Return Value : None

Example :

`Draw.Save('pict.svg')` Result : Creates a SVG-file `'pict.svg'`

[74] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawRect

Note : This function terminates a Draw function call list and writes the drawing instructions executed so far to a file – named *t*. The extension of the filename is automatically set to *svg*. If no *Draw.Save* instruction is used in the LPL code, then LPL automatically calls it at the end of a run to write a SVG-file with name of the model and extension *svg*.

Draw.Scale(zx,zy[,x,y,w,h])

Back to index

Define the drawing space, zoom

zx : double, zoom whole graphic on x direction

zy : double, zoom whole graphic on y direction

x : double, global minimal x-coordinate

y : double, global minimal y-coordinate

w : double, size of graphic in x direction

h : double, size of graphic in y direction

Return Value : None

Example :

```
Draw.Scale(2,1)
```
Result : Zoom graphic only in x direction by a factor 2

```
Draw.Scale(1,1,10,10,200,215)
```
Result : viewport of graphic is (0,0) to (210,225)

Note : This function – if called – must be called before any other drawing function and can be called only once per graphic. It can be called after a *Draw.Save*, however to initialize a new graphic. It can be used with two parameters (zx and zy). The graphic is zoomed by factor zx and zy (except the fonts). If zy is negative the graphic is placed upside-down (except all text). The size of the graphic is calculated automatically. If the user is not happy with the size, he can use the other four parameters to define the global viewport and size: $(x \cdot zx, y \cdot zy)$ is the left/top coordinate and $(w \cdot zx, h \cdot zy)$ is the size of the graphic.

Draw.Text([id,]t,x,y[,h,c1,c2,w,o1,o2,a,r]) Index

Draw a text

id : string

t : string, text to be placed

x : double, x-position

y : double, y-position

h : integer, font height

c1 : integer, fill color, default black (0)

c2 : integer, stroke color, default black (0)

w : integer, stoke width of the text, default 1

o1 : double [0..1], fill opacity, default 1

o2 : double [0..1], stroke opacity, default 1

a : double, align position of (x, y) (left [0-0.33], middle [0.34-0.66], right [0.67-1])

r : double, text rotation in degrees, default 0

Return Value : None

Example :

```
Draw.Text('Text',20,20)
```

Result :'Text' is placed at $(20, 20)$, left/bottom

Note : See the model example xDrawText[75].

Draw.Triangle([id,t,] x,y,ra [,c1,c2,w,o1,o2,rot]) Index

Draw a circle

[75]https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawText

id : string

t : string, text to be placed

x : double, x-position of center

y : double, y-position of center

ra : double, radius (semi-height)

c1 : integer, fill color, default black (0)

c2 : integer, stroke color, default black (0)

w : integer, stroke width, default 1

o1 : double [0..1], fill opacity, default 1

o2 : double [0..1], stroke opacity, default 1

rot: double, rotation around the center in degrees

Return Value : None

Example :

```
Draw.Triangle('hh',50,50,40,3,4,5)
Draw.Triangle(100,100,30,6,5,5,.1,.2,180)
```

Result : Two triangle are drawn, the second pointing downwards. The first contains a text inside, the fill color is 3 (red) the stroke color is 4 (green), the stroke width is 5.
For the second circle the fill color is 6 (yellow) with an opacity of 0.1, the stroke color is 5 (blue) with an opacity of 0.2.

Draw.XY(x,y[,z,w][,t,s,c]) Index

Draw a XY plot

x : a reference to a list of doubles

y : a reference to a list of doubles

z : a reference to a list of doubles (optional parameter)

w : a reference to a list of doubles (optional parameter)

t : 0 for dots, 1 for lines

s : size of the dots (of the line)

c : color (LPL color)

Return Value : None

Example :

```
Draw.XY(x,y)
Draw.XY(x,y,z)
```

Result : a xy-plot is generated with x in the x-axe and y (and z,w) in the y-axe.

Note: For a model example see xDrawXY1[76].

3.13.4 Statistical Library

This section explains the statistics library functions. The first 6 functions are discrete distribution functions. The last parameter *cum* may missing then it is zero. If *cum* is 0 the the point distribution $P(X = k)$ is calculated, if it is 1 the cumulative $P(X \leq k)$ is calculated.

Stats.Binomial(n,k,p,cum) Back to index

Return Binomial distribution

n : integer, the parameter n (number of draws)

k : integer, the number k (number of success)

p : double, the probability

cum : 0=point, 1=cumulative

Return Value : double, the Binomial distribution

Example :

[76] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawXY1

`a:=Stats.Binom(100,50,0.5)` Result : $a = 7.95\%$

Note : The function generates the value of the Binomial distribution $B(n, k, p)$. An model example is given in binomial[77].

Stats.NegBinomial(n,k,p,cum) Back to index

Return negative Binomial distribution

n : integer, the parameter n (number of draws)

k : integer, the number k (number of success)

p : double, the probability

cum : 0=point, 1=cumulative

Return Value : double, the negative Binomial distribution

Example :

`a:=Stats.NegBinomial(10,6,0.5)` Result : $a = 7.637\%$

Note : The function generates the value of the negative Binomial distribution $NB(n, k, p)$.

Stats.Poisson(k,mu,cum) Back to index

Return Poisson distribution

k : integer, the number k (number of success)

mu : double, the probability

cum : 0=point, 1=cumulative

Return Value : double, the Poisson distribution

Example :

[77] https://lpl.matmod.ch/lpl/Solver.jsp?name=/binomial

`a:=Stats.Poisson(0,1)` Result : $a = 37.8\%$

Note : The function generates the value of the Poisson distribution $P(k, mu)$. An model example is given in poisson[78].

Stats.Logseries(k,p,cum) Back to index

Return Logarithmic Series distribution

k : integer, the number k (number of success)

p : double, the probability

cum : 0=point, 1=cumulative

Return Value : double, the Log Series distribution

Example :

`a:=Stats.Logseries(1,0.8)` Result : $a = 50\%$

Note : The function generates the value of the Log Series distribution $L(k, p)$.

Stats.Geometric(k,p,cum) Back to index

Return Logarithmic Series distribution

k : integer, the number k (number of success)

p : double, the probability

cum : 0=point, 1=cumulative

Return Value : double, the Geometric distribution

Example :

`a:=Stats.Geometric(1,0.8)` Result : $a = 16\%$

[78] https://lpl.matmod.ch/lpl/Solver.jsp?name=/poisson

Note : The function generates the value of the Geometric distribution $G(k, p)$.

Stats.Hypergeometric(n,k,n1,k1,cum) Back to index

Return Hypergeometric distribution

n : population size

k : number of successes in population size

n1 : size of a samples

k1 : number of successes in the sample

cum : 0=point, 1=cumulative

Return Value : double, the Geometric distribution

Example :

```
a:=Stats.Hypergeometric(20,6,5,2,1)
```
Result : $a = 86.86\%$

Note : The function generates the value of the Hypergeometric distribution $H(n, k, n1, k1)$.

Stats.Gumbeldis(x,m,b) Back to index

Return Gumbel distribution

x : double, the input value

m : double, the parameter μ

b : double > 0, the parameter β

Return Value : double, the cumulative Gumbel distribution

Example :

```
a:=Stats.Gumbeldis()
```
Result : $a = 5.5$

```
a:=Stats.Gumbeldis(-5.5)
```
Result : $a = 5.5$

Note : The function generates the cumulative value up to x of the Gumbel distribution $F(x;\mu,\beta)$.

Stats.Normdis(x,m,s) Back to index

Return Normal cumulative

x : double, the input value

m : double, the mean

s : double, standard deviation

Return Value : double, the cumulative Normal distribution

Example :

```
a:=Stats.Normdis(0,0,2)
```
Result : $a = 0.5$

Note :

Stats.Norminv(x,m,s) Back to index

Return inverse normal cumulative

x : double, the input value

m : double, the mean

s : double, standard deviation

Return Value : double, the cumulative Normal inverse distribution

Example :

```
a:=Stats.Norminv(0.75,3,10);
```
Result : $a = 9.74$

Note :

3.13.5 Graph Library

This section explains the graph library functions.

Graph.Bfs(r,c,n,m) Back to index

Return bfs search t in p of a graph r

r : A 2-dimensional table of edges

p : A 1-dimensional parameter table of nodes

n : (integer) the starting node of the search tree

m : (integer) a random seed for randomizing neighborhood in the traversal

Return Value : none

Example :

```
set i,j "vertices";
set e{i,j} "edges";
parameter p{i} "the previous numbers";
a:=Graph.Bfs(e,p,0,2)
```

Result : p is built

Note : The first parameter r must be a 2-dimensional (sparse) matrix, reflecting the edges, p is a parameter vector that contain the number of the precedent vertex in the search tree. An model example is learn36[79] another is maze[80].

Graph.Components(r,c) Back to index

Return components c, of a graph r

r : A 2-dimensional table of edges

c : A 1-dimensional parameter table of components

[79] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn36
[80] https://lpl.matmod.ch/lpl/Solver.jsp?name=/maze

Return Value : integer, the number of components

Example :

```
set i,j "vertices";
set e{i,j} "edges";
parameter C{i} "the component numbers";
a:=Graph.Components(e,C)
```

Result : C is built

Note : The first parameter r must be a 2-dimensional (sparse) matrix, reflecting the edges, c then a parameter vector that contain a number of component membership. An model example is learn30[81].

Graph.Dfs(r,c,n,m) Back to index

Return a dfs search tree p of a graph r

r : A 2-dimensional table of edges

p : A 1-dimensional parameter table of nodes

n : (integer) the starting node of the search tree

m : (integer) a random seed for randomizing neighborhood in the traversal

Return Value : none

Example :

```
set i,j "vertices";
set e{i,j} "edges";
parameter p{i} "the previous numbers";
a:=Graph.Dfs(e,p,0,2)
```

Result : p is built

Note : The first parameter r must be a 2-dimensional (sparse) matrix, reflecting the edges, p is a parameter vector that contain the number of the precedent vertex in the search tree. An model example is maze[82].

[81] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn30
[82] https://lpl.matmod.ch/lpl/Solver.jsp?name=/maze

Graph.Mincut(r,c [,s,th])

Back to index

Return minimal cut

r : A 2-dimensional table of edge weights

c : A 2-dimensional table, the min-cut list

s : A 1-dimensional table of vertices, a partition

th : a threshold of min-cut

Return Value : the size of min-cut

Example :

```
set i,j "vertices";
parameter w{i,j} "edge weights";
parameter C{i,j} "the cut";
set S{i} "one vertices subset of the cut";
a:=Graph.Mincut(w,C,S)
```

Result : The Min-Cut of w

Note : The function *Mincut* returns three results: (1) the min-cut edge list in the table c, the subset of vertices that are on one side of the cut as S, and the resulting weight of the min-cut. The threshold can be set to any number th, if the algorithm has found a cut of at most th it stops. As an model example see learn31[83]. (LPL implements the Stoer-Wagner algorithm for finding the minimal cut of a undirected graph.)

Graph.Mstree(r,c)

Back to index

Return minimal spanning tree

r : A 2-dimensional table of edge weights

c : A 2-dimensional table of edges, a list of min tree

Return Value : The weight of the min-tree

Example :

[83] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn31

```
set i,j "vertices";
parameter w{i,j} "edge weights";
parameter T{i,j} "the min-tree";
a:=Graph.Mstree(w,T)
```

Result : The min-tree of w

Note : The function *Mstree* returns two results: (1) the min-tree edge list in the table c, and the resulting weight of the minimal spanning tree. For an model example see learn32[84].

Graph.SPath(r,c,s[,t]) Back to index

Return shortest path (tree)

r : A 2-dimensional table of edge weights

c : A 2-dimensional table of edges, a list of the path tree

s : an integer, the starting node

t : an integer, the destination node

Return Value : The shortest path length between s and t

Example :

```
set i,j "vertices";
parameter w{i,j} "edge weights";
parameter T{i,j} "the sp-tree/path";
a:=Graph.SPath(w,T,0,10)
```

Result : The shortest path length between node 0 and 10

```
a:=Graph.SPath(w,T,0)
```

quad Result : The shortest path tree starting at node 0.

Note : The function *SPath* returns two results: (1) the shortest path edge list in the table c, and (2) the resulting length. If the parameter t (the destination) is missing then the shortest path tree (starting from node s is returned. In this case the length is zero. If the graph contains a negative cycle then negative Infinity (-99999999) is returned. If no path exists between s and t then Infinity (99999999) is returned. For an model example see learn37[85].

[84] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn32
[85] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn37

Graph.Topo(r,c) Back to index

Return topological sort of a DAG

r : A 2-dimensional table of arcs

c : A 1-dimensional list of the sorting of nodes

Return Value : 0

Example :

```
set i,j "vertices";
parameter r{i,j} "edge list";
parameter c{i} "the sorted nodes";
a:=Graph.Topo(r,c)
```

Result : 0, result is in `c`

Note : The function *Topo* returns the topological sorting of the nodes of a DAG (Directed Acyclic Graph). Note that The function does not check whether `r` is a DAG. For an model example see learn52[86].

3.13.6 Geometry Library

This section explains the Geometry library functions.

Geom.Inside(x,y,X,Y) Back to index

Return 1 if point (x,y) is inside (simple) polygon (X,Y)

x : double, the x-coordinate

y : double, the y-coordinate

X : a set of x-coordinates

Y : a set of y-coordinates

Return Value : double, (1 or 0)

[86] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn52

Example :

```
set i:=[1..3]; parameter x{i}:=[0 2 0]; y{i}:=[0 1 2];
a:=Geom.Inside(1,1,{i}x,{i}y);                    Result : a = 1
a:=Geom.Inside(3,3,{i}x,{i}y);                    Result : a = 0
```

Note : The number of point in the third (X) and the fourth parameter (Y) must be the same. For a model example see xDrawInside.lpl[87].

Geom.Intersect(X,Y,i,j,m,n) Back to index

Return 1 if line segment (i,j) intersects with line segment (m,n)

X : name to a list of x-coordinats

Y : name to a list of x-coordinats

i : i-th entry in (X,Y)

j : j-th entry in (X,Y)

i : m-th entry in (X,Y)

j : n-th entry in (X,Y)

Return Value : double, (1 or 0)

Example :

```
set i:=[1..4]; parameter X{i}:=[0,1,0,1]; Y{i}:=[0,0,1,1];
a:=Geom.Intersect(X,Y,1,2,3,4);                   Result : a = 1
a:=Geom.Intersect(X,Y,1,3,2,4);                   Result : a = 0
```

Note : The points on a plane are given by two vectors X and Y. The i-th point then is (X[i],Y[i]). example see xDrawIntersect.lpl[88].

[87] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawInside.lpl
[88] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawIntersect.lpl

3.13.7 Struct Library

This section explains the Struct library functions.

Struct.AddMap(x[,y]) Back to index

Return 0

x : first entry

y : second entry (if missing it is 0)

Return Value : 0

Example :

```
Struct.AddMap(17);      Result : adds (17,0) to Map table
Struct.AddMap(17,18);   Result : adds (17,18) to Map table
```

Struct.DeQueue() Back to index

Return the (first) queue entry

Return Value : queue entry

Example :

```
Struct.DeQueue();   Result : adds (17,0) to Map table
```

Struct.EnQueue(x) Back to index

Return 0

x : Queue entry

Return Value : 0

Example :

```
Struct.EnQueue(23);   Result : adds 23 to Queue table
```

Struct.GetMap(x) Back to index

Return second entry

x : first entry

Return Value : second entry

Example :

```
Struct.GetMap(17);
```
Result : return second entry (0 if none)

Struct.InMap(x) Back to index

Return 1 or 0

x : first entry

Return Value : 1 or 0

Example :

```
Struct.InMap(23);
```
Result : 1 if 23 is in the Map else 0

Struct.Map(x[,c]) Back to index

Return 0

x : integer : size of the Map

c : char: 'i', 'l', 'd'

Return Value : 0

Example :

```
Struct.Map(10000,'d');
```
Result : A map of size 10000 of double entries

```
Struct.Map(1000);
```
Result : A map of size 1000 of integer (4 bytes) entries

Note : The content of a map can be integers (4 bytes) ('i'), Int64 ('l'), or double ('d'), depending on the second argument. If the second argument is missing or different from 'i', 'l', 'd' the entries are integers.

Struct.Queue(x[,c])

Back to index

Return 0

x : integer : maximal size of the Queue

c : char: 'i', 'l', 'd'

Return Value : 0

Example :

```
Struct.Queue(10000,'d');   Result : A Queue of size 10000 of double entries
Struct.Queue(1000);        Result : A Queue of size 1000 of integer (4 bytes) entrie
```

Note : The content of a queue can be integers (4 bytes) ('i'), Int64 ('l'), or double ('d'), depending on the second argument. If the second argument is missing or different from 'i', 'l', 'd' the entries are integers.

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
&&	bitwise and operator
||	bitwise or operator
%	modulo operator, read/write format specifier
^	power operator

`<`	less operator
`>`	greater operator
`=`	equal operator, match-operator in database communication
`( )`	to change operator precedence in expressions
`[ ]`	to bracket applied index-lists, lower/upper bound specification
`{ }`	to bracket an index-list
`'...'`	string literals
`"..."`	to enclose a comment attribute
`|`	data tables delimiter, and expression selector in index-lists
`.`	decimal point, or default value in data tables
`,`	to list items or expressions
`:`	*define* operator
`;`	terminal delimiter of a statement
`#`	cardinality operator
`$`	set transform operator
`~`	NOT operator
`@`	date/time data

Tokens consisting of one or more characters are:

`--`	begin a one-line comment
`//`	same as –
`/*`	begin a multiline (nested) comment
`*/`	end multiline comment
`<>`	not-equal operator
`<=`	less or equal than operator
`>=`	greater or equal than operator
`:=`	assign operator
`->`	logical implication
`<-`	reverse implication
`<->`	logical equivalence
`<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
`<Functions>`	to a function name

3.15. LPL 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 have some particularities because indexed operators are usually or part of typical programming languages:

- The indexed operators such as $\sum$ used in mathematics to sum terms over indices is replaced by reserved words, such as `sum` etc.
- Indices are not subscripted as in $a_{i,j}$. 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]`.
- In contrast to programming languages, tables such as `a{i,j}` are sparsely stored if necessary. Hence there access is not necessarily constant. But large potential tables can be stored.

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 for *true* and to 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 a default value. The same is true for variables. Indices used in an expression return the position of a specified element within the set (set are always considered as ordered in LPL). If indices are used in an expression, they must be bound (see Chapter 5) 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 singleton
`sum{i} a[i]`	is an expression, evaluate to a singe value
`a[i,j] + c[j]`	is an indexed expression
`a[i] + sum{i} c[i]`	is another indexed expression

Expressions are basically used in three 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
- in parameter of functions

Examples:

`parameter composed{i} := a[i] + sum{i} c[i]`	to assign data
`constraint r: x+y-23*z <= 78`	to define a constraint
`sum{i,j,k,m | a[i]>b[j] or c[k]<d[m] } ...`	in an index-list
`Abs(a-b*3)`	in parameters

The comma operator is a list operator the left term is evaluated then the right term is evaluated (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 an 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 code to calculate the greatest common divisor of the two numbers 994009 and 96709 (Euclid's algorithm):

```
parameter a:=994009; b:=96709; t;
Write('The gcd is: %d\n',
      while(b>0,(t:=a, a:=b, b:=t%b, a)));
```

Note that *while* is a loop operator *within an expression* (not to be confound with the while statement, see next example). The loop operator is followed by a two expressions within parentheses. The first expression is the loop break condition and the second is the return value. Since the second is itself a list of expression, they are executed until the break condition is false and the last (a) is returned. This feature allows the modeler to implement algorithms as expressions in LPL.

In LPL the syntax that is somewhat more familiar to programmer is:

```
parameter a:=994009; b:=96709; t;
while b>0 do t:=a; a:=b; b:=t%b; end
Write('The gcd is: %d\n',a);
```

Another operator that can be used within an expression is *if*. Example:

```
parameter a:=1; b:=2;
c:= 5 + if(b=1,10,b=2,a,23);
```

The if expression returns 10 if b is 1, it returns a if b=2, else it returns 23, (see model learn01a[89].

[89] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn01a

Chapter 4

The Structure of a LPL Model

In Chapter 2 all basic elements of a LPL model are explained. This Chapter explains how to put these elements together to form a complete model. The overall structure is:

```
model <ModelHeader>
  <statement list>
end
```

The model code begins with the reserved word *model* and ends with the reserved word *end*. Enclosed it contains a list of *statements*. The statements can be classified into 6 *declaration statements*, 7 *executable statements* and the *empty statement*. They can be in any order. They are:

set declaration	to declare sets
parameter declaration	to declare data
variable declaration	to declare variables
constraint declaration	to define constraint expressions
expression declaration	to define expressions
model declaration	to declare sub-models
solve statement	to solve a model
if statement	to branch a statement list
while statement	to loop statements
for statement	to loop statements

model-call statement	to call sub-models
expression statement	to execute an expression
assignment statement	to assign an expression

The syntax of all statements is similar and consists of a sequence of at most 13 attributes: They are:

type attribute	to define the type of a declaration
genus attribute	to identify the kind of statement
name attribute	to define a name for the declaration
index attribute	to specify vector and tables
range attribute	to define lower/upper bound range
subject_to attribute	to specify the constraints to solve
if attribute	to specify whether the statement is used or not
friend attribute	to access the scope of another model
priority attribute	to define hints/start value/priority of variables
quote attribute	used to mark a statement
comment attribute	to specify a qualified comment
default attribute	to define a default value
frozen attribute	to activate/fix an entity
expression attribute	to assign an expression

Not all statements contain all attributes. Each statement ends with a semicolon. An example for a variable declaration is the following:

```
integer variable x,X{i,j}
       "Quantity to transport on link (i,j)" [0..100];
```

The keyword *integer* is the type attribute, *variable* is the genus attribute, x, X is the name attribute (the variable has two names x and X), *"Quantity to transport"* is the comment attribute, and *[0..100]* is the range attribute.

First, the attributes are explained, then the different statements are listed.

4.1. The Statement Attributes

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

4.1.1 The Type Attribute

The type attribute only occurs in a declaration statement. It consists of one of the keyword as follows:

integer	to declare integer values
binary	to declare binary (0-1) values
date	to declare date/time values
string	to declare alpha-numerical values
alldiff	to declare integer values (all different) (only for variables)

The attribute defines the type of a declaration. If no type attribute is used, the value domain of a declaration is **double** (8-byte floating point). The type **integer** stands for integral numbers. If it is used for variables it declares (discrete) integer variables. The type **binary** declares the values to be zero or one. Used in variables it declares logical variables (0-1 variables). This type overrules the range attribute. The type **alldiff** is applicable for indexed variables. It says that all variables of this declaration must be integer and different from each other. LPL automatically generates the corresponding constraints. The type **date** is basically the same as double, however the input and output are dates. The type **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 dynamically allocated. Examples:

```
integer parameter a;     -- declare integer parameter a
binary variable b;       -- b is a 0-1 variable
alldiff variable x{i};   -- all variables are different from each other
string parameter c;      -- c is a string parameter
```

4.1.2 The Genus Attribute

The genus attribute is for identify the statement and consists of one of the following keywords:

set	for a set declaration
parameter	for a parameter declaration
expression	for an expression declaration
variable	for a variable declaration
constraint	for a constraint declaration
model	for a (sub)-model declaration
maximize	for a solve statement
minimize	for a solve statement

solve	for a solve statement
if	for a if statement
while	for a loop statement
for	for a loop statement

The model-call and the expression statement do not contain a genus attribute. Examples:

```
set i;                    -- declares a set named i
parameter a := 2;         -- declares a parameter a
expression E := a+2;      -- declares an expression
variable x; y;            -- declares variable x and y
constraint c : x+y>=2;    -- defines a constraint named c
maximize obj: x+y;        -- solve statement obj
minimize this: x-y;       -- solve statement this
solve;                    -- solve statement
model submodel ... end    -- sub-model declaration
if a>12 then ... end      -- if statement
while a<>b do ... end     -- while loop statement
for{i} do ... end         -- for loop statement
submodel;                 -- a model call statement
```

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 variable y). The model declaration is special in the sense that it can contain itself a complete model See below). Models can be recursively declared within other models.

4.1.3 The Name Attribute

This attribute consists of a user-defined identifier. Every declaration as well as a minimize/maximize solve statement has a unique user-defined name. It is used for reference in expressions and other parts of the code.

```
minimize this: x-y;  -- name is: 'this'
variable x,X;        -- two names 'x' and 'X'
set I;               -- name of set is 'I'
```

The name attribute can consists of at most three identifiers separated by commas. Each declaration has a unique identifier (the name attribute). However, sometimes it is useful to have more than one name for the same entity. These two additional identifiers are called aliases. Normally the aliases are shortcuts which can be used in complex expressions later on in the model. It is useful especially for sets and index names. Example:

```
set i,j;  -- use two names,
variable X{i,j}; -- declare a matrix of variables
constraint C{i}: sum{j} X[i,j]
```

Because set names can also be used as index names in LPL – if the context allows this – it is very convenient to define several names for the same set. This avoids to introduce local index names in expressions when i and j have to be referenced separately as in the example above. There is no need to use more than one identifier for a set. The code fragment above could also be written in LPL as follows used the in keyword (which is closer to the mathematical notation):

```
set I;  -- use one name only
variable X{i in I,j in I};    // i and j are locally defined indexes
constraint C{i in I}: sum{j in I} X[i,j]
```

I personally prefer the first formulation as it is somewhat shorter.

4.1.4 The Index Attribute

The index 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 matrix x of variables
```

These previous five attributes – if used – must be in this order. The next 8 attributes can be in any order within a statement.

4.1.5 The Range Attribute

The range attribute consists of an expression (containing exactly one ..) 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 between 3 and 5
variable x  [3-2..10+12]; --lower/upper bound is 1..22
```

The bounds are defined only for numerical entities otherwise they are ignored. Variables of linear and qp models without a range indication have range $[0 \ldots \infty]$. Parameters and variables of non-linear models have the default range $[-\infty \ldots \infty]$. Binary variable automatically have range $[0 \ldots 1]$.

The range attribute can also be used to define *semi-continuous*, *semi-integer*, and even *multiple-choice* variables. A semi-continuous(integer) variable is defined as an (real/integer) variable that can be zero or between a (positive) lower and upper integer bound. For model examples see semi-1[1], semi-2[2], and mchoice-3[3].

4.1.6 The subject_to Attribute

This attribute begins with the keyword **subject_to** followed by an expression. The expression must be a 'subjectList', that is, a list of constraint- or model-names. It is using only in a solve statement. Each name can be preceded by a negation operator ( ). Example:

```
minimize obj: x+y subject_to (modA, ~modB, const1, ~const2);
```

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

Alternatively, the expression can also begin with an **if** keyword. It then has the syntax (where `BoolExpr1` and `BoolExpr2` are two arbitrary Boolean expressions and `subjectList1` and `subjectList2` are two lists as described before):

```
subject_to  if( BoolExpr1 , (subjectList1) , BoolExpr2 , (subjectList2) , .... )
```

The expression is evaluated as follows: if `BoolExpr1` is true then `subjectList1` is taken, else if `BoolExpr2` is true then `subjectList2` is taken, else no constraints are taken. A model example can be found in learn40[4].

[1] https://lpl.matmod.ch/lpl/Solver.jsp?name=/semi-1
[2] https://lpl.matmod.ch/lpl/Solver.jsp?name=/semi-2
[3] https://lpl.matmod.ch/lpl/Solver.jsp?name=/mchoice-3
[4] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn40

4.1.7 The If Attribute

This attribute begins with the keyword *if* followed by an expression. It is applicable for the constraint declaration and a solve statement. Applied to a constraint, it means that the constraint has no effect if the condition is false – it is like the *frozen* attribute. 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.

4.1.8 The friend Attribute

This attribute is only valid in the model header. It begins with a keyword *friend* followed by a model name. It means that all identifiers in the friend model are also valid in the actual model without the qualified (dot) notation. For example:

```
model output friend data;
```

This means that a identifier `id` within the model `data` can also be used in the output model without dots: just use `id` instead of `data.id`. See model learn30[5].

4.1.9 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 (This is basically for the cplex solver).

For the Gurobi solver, this attribute can be used to pass a (partial) solution to the solver (warm start) or to pass hint values for variables. If the expression is -1 then the value of the variable is passed as a warm start, if the expression is positive then the value of the variable is passed as a hint and the expression value is passed as a hint priority (see Gurobi documentation for hints). If the expression is zero then nothing is passed. If the expression is less than -1 then its positive value is passed as a branching priority attribute. Note that for all these options to be effective the solver option MIPSTART=1 (see solver options) must be added.

[5] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn30

4.1.10 The Quote Attribute

This attribute consists of a single quoted string. It is used in the objective function for defining various attributes (see Chapter 7) and in some other case to mark the statement (see f.e. 7.3).

4.1.11 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, it is especially useful for automatic documentation the model. Example:

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

4.1.12 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;            -- default value of a is 1
parameter b;                      -- default value of b is 0
string parameter s;               -- default value is '' (empty string)
string parameter t default 'abc'; -- default value is 'abc'
```

4.1.13 The frozen Attribute

This attribute consists of the keyword *frozen*. Applied to set and parameter declarations make them immune against *ClearData()* function or snapshot read instructions, that is they are not modifiable. Example:

```
model m;
  parameter a := 4 frozen;  -- keep that value
  parameter b := 5;         -- volatile
  ClearData(m);  -- delete all data within m (except a)
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 – the constraint is “switched off”.

4.1.14 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;     -- declare and assign a
expression 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). One can use the two operators interchangeable. Good practice is to use the colon only with constraint and expression statements.

4.2. The Statements

The 15 different statements are now explained.

4.2.1 Set Declaration

The set statement declares or defines an index set or a relation (see compound set) used in the model. An index set is also called a set. 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 one or several index declarations or definitions. (The semantic of sets is explained in Chapter 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
```

Note that it is not necessary to repeat the keyword *set* after each declaration.

4.2.2 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 multidimensional tables (vectors, matrices or n-dimensional arrays) may be defined. The dimensions are determined by the sets, 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 one or several parameter declarations or definitions. The semantic of data (tables) is explained in Chapter 6. Example:

```
parameter
  a;
      -- declares a single parameter a (no data)
  b := -4.678;
      -- defines a single parameter and assigns a value
  c{i};
    -- declares a vector c where i is an set
    -- 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 i times j)
  e{i} := b*c[i];
    -- define e from an expression
```

i and j are set 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 Chapter 5. The data table formats are explained in Chapter 6.

4.2.3 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 Chapter 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
```

4.2.4 Constraint Declaration

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 Chapter 7. Example:

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

4.2.5 Expression Declaration

This statement defines arbitrary expressions. These expressions are only evaluated when used, as shows the next example:

```
expression a := b;
parameter b := 2;
Write(a);   -- write the value 2
b:=3;
Write(a);   -- writes the value 3
```

Expression names can also be used within constraints. They will be expanded when send to the solver automatically.

```
variable x; y; z;
expression XY: x+y;
constraint A: XZ + 2*z <= 5;
```

Internally, an **expression** is translated to a **frozen constraint**.

4.2.6 Model Declaration

A LPL model can be a hierarchical structure, since sub-models can be declared within models. Sub-models are like 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 sub-models. They are even visible outside model m and can be accessed by the dot notation in an expression such as:

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

The parameters a and bb on line 4 are defined within 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). The parameter a (line 2) could nevertheless 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.

Models can have formal parameters like a function in other languages – note that the keyword @function@ is equivalent to @model@ in LPL. Parameters are declared like declarations. All parameters are passed by reference, except singleton @parameter@ declarations they are passed by value. See the tutor examples learn41[6] and learn42[7].

[6] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn41
[7] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn42

4.2.7 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 *solve*. 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 Chapter 7. Example:

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

4.2.8 if Statement

This statement 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 keyword *end*. 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. The semicolon before an *else* be be dropped.

4.2.9 while Statement

The while statement implements loops. It begins with the keyword *while* followed by a (Boolean) expression. This is followed by a keyword *do* 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.

4.2.10 for Statement

The for statement 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.

4.2.11 Model-call Statement

A (parameterless) model-call statement consists of a single model name that was declared before or after the call. Note that recursive model calls are not allowed in LPL. Example:

```
MyModel;
```

If a submodel is declared with formal parameters then a model call must also contain these parameters. The type and the genus must match. Only singleton @parameter@ parameters are passed by value, all others are passed by reference. Note that in the calling list, singletons can be arbitrary expressions. See the tutor examples learn41[8] and learn42[9].

4.2.12 Expression Statement

Any expression can be itself used as a statement. Of course, a expression such as $1 + 3$ is a correct statement in LPL, but it has no effect. Only expressions that produce a side-effect are useful statements. Examples are function calls such as the following:

```
Draw.Ellipse(10,10,3,4);  -- draws an ellipse
a:=b+c;                   -- assigns a value to a
Read('file.txt',a);       -- reads a from a file
```

[8] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn41
[9] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn42

4.2.13 Assignment Statement

An assignment statement assigns values to an entity. An identifier is followed by an assignment operator `:=` and by a arbitrary expression. Note that an assignment statement is basically an expression statement. Internally they are transformed into a pure expression statement. An example is:

```
b{i} := a[i];  -- copy a into b
```

Strictly speaking, the previous statement is not a correct expression, but LPL makes some compromise to readability for assignments. The correct expression would be:

```
{i} (b[i] := a[i]);
```

4.2.14 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*, as seen above. So, we may write

`set i; j;` instead of `set i; set j;`

An empty statement ends the declaration list. Normally, a empty statement is not needed. Only if a call to a model or an assignment follows, it is needed. Example:

```
set i;
    j;;
MyModel;
```

The set declaration list *must* be ended with an empty statement (that is, with an additional semicolon) after the declaration of `j`, because the identifier `MyModel` would be a set and not a call to a submodel `MyModel`.

4.3. How is a Model processed?

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

1. Parse the code completely,
2. Eventually, transform the model completely
3. 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.

Parsing the code means to create an internal structure of the model. The LPL parser is a three-pass parser: on the first pass all entities are collected, on the second pass all index-set identifier are attached to the indexed entities, and on the third pass the model and all expressions are considered. The three pass parser guarantees that the order of declarations in the code is not important.

Transforming a model means to modify the internal structure of the model (and all submodels). The procedure is explain in details in the paper [3]. Basically, logical constraints are transformed into linear inequalities. This transformation is not done, if (1) the model is a permutation model or (2) if the compiler switch is '0' (zero).

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 statement. Models can be declared after their use as can be any identifier as mentioned above since the order of declaration is not important.

If the top model contains two submodels with the name "data" and /or "output" then these models are executed without calling them explicitly. If called explicitly, they behave like any other model call (without an extra implicit call). If not called explicitly, then "data" is called before the first executable statement and the "output" model is called at the very end of a run. One can redirect the implicit calls to other submodels with the APL parameter @IN and @OUT. In this way, the data can be separated completely from the model structure and even be in another file.

Chapter 5

Index-Sets

This chapter explains how sets and indices are declared and used in LPL. They are the most important construct in the LPL language[1].

5.1. Introduction

A *set* is a finite collection of different *elements*. In mathematical modeling, sets are used to define multidimensional objects like vectors or matrices. In mathematical notation, a matrix is written as follows:

$$x_{i,j} \quad \text{with } \forall\, i \in I, j \in J$$

We say x is a two-dimensional matrix spanned over the Cartesian product of *sets* – or *index-sets* – I and J with $I = \{1, \ldots, m\}$ and $J = \{1, \ldots, n\}$ with $m, n > 0$. The name i and j are called *indexes*.

The matrix $x_{i,j}$ consists of $m \times n$ single entries, which – explicitly written in a matrix form – is as follows

$$\begin{pmatrix} x_{1,1} & x_{1,2} & \ldots & x_{1,n} \\ x_{2,1} & x_{2,2} & \ldots & x_{2,n} \\ \ldots & \ldots & \ddots & \ldots \\ x_{m,1} & x_{m,2} & \ldots & x_{m,n} \end{pmatrix}$$

[1] For a more formal definition and exposition on indexed notation in mathematical modeling see the indexing paper [2]

Sets and indexes are used in the mathematical modeling to build expressions using index-operators over vectors and matrices. An expression, for example, that sums all items $x_{i,j}$ in the matrix can be written as:

$$\sum_{i \in I} \sum_{j \in J} x_{i,j}$$

In LPL, sets are defined and declared in a **set statement** as follows:

```
parameter m; n;   --value to be defined
set I := 1..m;
set J := 1..n;
```

On the base of these declarations, one can define matrices in LPL as:

```
parameter x{i in I, j in J};
```

The summation expression (see above) can be written as:

```
sum{i in I} sum{j in J} x[i,j]    --or shorter
sum{i in I, j in J} x[i,j]
```

In contrast to mathematical notation, LPL can also use the same identifier as *index name* and as *set name* if the context allow it. The previous example could also be written as follows:

```
parameter m; n;   --value to be defined
set i := 1..m;
set j := 1..n;
parameter x{i,j};
sum{i,j} x[i,j]
```

This is much shorter and more readable and does not give cause for confusion: the identifiers `i` and `j` are used as set names and index names in different well defined contexts.

Sets can also be declared by enumerate their elements explicitly. An example is the following set of seasons:

```
set i := [ spring summer autumn winter ];
```

The number of elements that a set contains is called its *cardinality*. The cardinality of a set `i` can be returned by the expression `#i` in LPL.

5.2. Elements and Position

The elements of a set can be numbers, identifiers or strings. Element names are always considered as strings even if they are written as numbers. Hence the two following sets containing 4 elements are exactly the same in LPL:

```
set i := [ 1930 an_element 2 '19:00' ];
set i := [ '1930' 'an_element' '2' '19:00' ];
```

The elements can be separated by optional comma. The order of the elements are important in LPL. Each element has a *position* within the set beginning with 1. The last element has a position that is equal to the cardinality of the set. In the previous example the element `'1930'` has position 1 and the element `'2'` has position 3.

A consecutive list of integers can be used as elements, but they should always begin with 1. The three following set declarations are basically the same:

```
set i := [ 1 2 3 4 5 ];
set i := [ 1..5 ];
set i := 1..5;
```

Note, however, that the last of the three statement is an expression and 5 can be an expression, whereas in the second form this is a fixed set already known after parsing.

In an expression, an element can be referenced by the element name or by the position. For example let the two declarations be as follows:

```
set i := [ 1930 an_element 2 '19:00' ];
parameter a{i};
```

Now the first entry of `{a{i}` can be referenced in two ways:

```
a['1930']   -- referenced by element name
a[1]        -- referenced by position
```

Note that `a[1930]` is not the same. It is correct from the syntactical point of view, and would reference the 1930-th entry of `a{i}` – that does not exist. LPL would not complain and just returns the default value of `a{i}` (which is zero in this case).

5.3. Relations (Compound Sets)

Sets can also be indexed. In this case, they declare or define *tuple-lists*, *compound sets*, or *relations*. Example:

```
set location,i := [ NY BO LA ];
set links{i,i} := [ NY BO , NY LA , LA NY ];
```

the set `location` is a simple set containing 3 elements. The set `links` is a compound set also containing 3 elements – *tuples* of length two, and models the list of connections between the locations, which is a subset of the Cartesian product on `location`×`location`). Relations (compound sets) are a powerful mean to define multidimensional sparse tables.

5.4. Index-Lists

Sets and indices are used in LPL to define multidimensional objects such as parameters, variables, constraints, and compound sets – as seen. In mathematical notation, subscripts are used to define such objects as in $x_{i,j}$. x is said to be a two-dimensional matrix since it has exactly *two* subscripts. The number of subscripts following an object is called the *dimension* (or the *arity*) of that object. In LPL, the subscripts are written within curly braces, for example: `x{i,j}`. The `{i,j}`-part is called *index-list*. Index-lists are used to define parameters, variables, constraints, and relations. They are also be used after index-operators. 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
```

5.5. Index-Lists with Conditions

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

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

This index-list is the same as the following mathematical expression:

$$\sum_{i\in I, j\in J, k\in K | i=k \wedge a_i \neq 12} \cdots$$

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) if $a_{i,j} \neq 0$. 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 are defined as following:

$$x_t = y_t + \sum_{k<t} a_k z_k \quad \text{with } k, t \in \{1, \ldots, 5\}$$

They can be formulated in LPL as (note the use of two set names:

```
set t,k := [1..5];
constraint R{t}: x[t] = y[t]+sum{k|k<t} a[k]*z[k];
```

Also relation-names can be used in index-lists. This is shown in the next example. Suppose, there is a network with three nodes (cities NY, BO, and LA), linked by three routes (edges) (NY,BO), (NY,LA), and ((LA,NY). In general, we have a graph $G = (I, E)$ with a nodes set I and a n edges set E. Let take a trivial example with 3 nodes and 3 edges: the declarations in LPL are as follows:

```
set i,j := [ NY BO LA ];
set e{i,j} := [ (NY,BO) , (NY,LA) , (LA,NY) ];
parameter a{i,j} := ...(some data)  ;
variable x{e};
parameter a{i,j};
```

Note that the variable `x` is indexed over `e` which is a subset of the Cartesian product `{i,j}`. LPL remembers this, so the following three expressions are basically the same for LPL:

```
sum{i,j} x[i,j]
sum{i,j|e[i,j]} x[i,j]
sum{e}  x[e] ...
```

The first expression "runs through" the whole Cartesian product, generates the sum and excludes all `x[i,j]` that are not in part of `e`. So, for example, `x['BO','LA']` is not defined by `e` therefore it will be removed automatically. The second expression is an explicit way to sum over the `e`. The third is the most straight forward way. It is also the most efficient way. Whereas the first two formulations have running complexity of $\Omega(|i| \cdot |j|)$ – that is the full Cartesian product, the third formulation has complexity $\Omega(|e|)$. (For small graph, it is not big deal, and one can use either formulation!)

One can also mix sparse and not sparse tables as in the following expression – using the declaration above:

```
sum{e[i,j]} c[i,j]*x[e]
sum{e[i,j]} c[i,j]*x[i,j]  -- the same
sum{i,j}  x[e] ...
```

Note that `c` was declared over the full Cartesian product `{i,j}` while `x` was only over `e`. Both expressions are correct in LPL without sacrificing efficiency. In mathematical notation, one would write (where $E \subseteq I \times I$ and $i, j \in I$, I is the set of vertices and E is the set of edges in a graph):

$$\sum_{(i,j) \in E_{i,j}} c_{i,j} \cdot x_{i,j}$$

A more complicated situation is given when the sparsity is distributed over several (active) index-lists. The simplest such case is the flow constraint:

$$\sum_{j:(i,j) \in e} x_{i,j} = \sum_{j:(j,i) \in e} x_{j,i} \text{ forall } i \in I$$

In LPL, this can be written in two ways, the first been a non-sparse way and the second being sparse:

```
constraint A{i}: sum{j|e[i,j]} x[i,j] = sum{j|e[j,i]} x[j,i];
constraint A{i}: sum{j in e[i,j]} x[i,j] = sum{j in e[j,i]} x[j,i];
```

The sparse way *must* be used in the big model short-swiss-5[2], for example.

It is also possible to use a range syntax as an index. For example $2 \ldots s_i$ below:

[2] https://lpl.matmod.ch/lpl/Solver.jsp?name=/short-swiss-5

```
set i:=1..7;
parameter s{i}:=i;
parameter r{i,j in 2..s[i]} := 10*i+j;
```

5.6. Passive Index-Lists and Binding

The construct `{i,j}` is called *index-list*. We call it also ,index-list the emphasize that its role is "active" – it guides the "running through" all elements, its indexes are like loop variables that are assigned sequentially to an element within the set. In expression:

$$\sum_{i \in I} a_i$$

the name i beneath the $\sum$ runs through all elements in I. It plays the "active" role of receiving an element after the other. The i in the expression a_i plays a "passive" role of being assigned the same element as its active counterpart. We call i in a_i *passive index-list*. LPL marks a clear syntactical difference between active and passive index-lists. The expression above must be formulated in LPL as follows:

```
sum{i} a[i]
```

While active index-lists contain curly braces, passive index-list always must be surrounded by brackets. *Every* index in a passive index-list must have an sibling index (its counterpart) with the same name in a previously defined active index-list. This mechanism is called *binding*. Each index in a passive index-list must be bound uniquely to an index in an active index-list. (Note that an index in the active index-list does not necessarily need an sibling in a passive index-list.) The LPL parser tries hard to bind every index. If it cannot it generates an error "no binding possible". Here some examples:

```
parameter a{i,j} := b[i,j] + sum{i} c[i,j];
parameter x{i} := b[i,i];
parameter y{i,i} := b[i];  -- ambiguous
parameter z{i} := a[k];    -- error
```

The first expression is perfectly correct. LPL binds the `i` in `b[i,j]` to the `i` in `a{i,j}`. However, the `i` in `c[i,j]` is bound to closer `sum{i}`. The second expression is correct if we want to copy the diagonal entries of `b` to the vector `x`. The third expression does not generate an error, but it is not clear to which `i` the index will be bound. The last expression is a clear error, because `k` cannot be bound.

Note that the passive index-list can be an arbitrary expression surrounded by brackets. Entries of a table like `a{i,j}` are stored in lexicographical ordering, with increasing positions of elements from right to left within the index-list. That is, we have the ordering from 1 to mn – if the cardinalities of i and j are m and m:

```
1:    a[1,1], 2: a[1,2], ...,  n: a[1,n],
n+1:  a[2,1]  n+: ...        2*n: a[2,n],
2n+1: a[3,1]      ....
(m-1)*n: a[n,1]   ...        m*n: a[m,n]
```

In LPL, we might alternatively write the passive index-list in two ways:

```
a[i,j]           --or
a[(#i-1)*i+j]
```

The first notation is a convenient way to access the entry (i, j). What LPL really does is calculating $(m-1)i+j$ to access this entry. Note that this allows also to place any expression such as:

```
a[i+1,j-1]           --which is the same as
a[#i*i+j-1]
```

The notation is very common the access the "next" entry of `i`, etc. This is useful in ordered sets such as time period and others. In this way we can conveniently access the next or previous time period.

Another application is copying while shifting a table:

```
integer parameter n;
parameter a{i} := b[i%#i+1]
parameter a{i} := b[(i-1+n)%#i+1]
```

Note that `%` and `#` are the modulo and the cardinality operators. The first assignment is a copying `b` into `a` while shifting down by one position. The second expression is copying while shifting down n positions (shifting up if n is negative.

Chapter 6

Data in the Model

All numerical values, strings or dates used in an LPL model are called data. Data in LPL are normally read from external devices, such as text-files or databases. However, they can also be stored inside models in tables or generated by expressions. While sets are defined in the set declaration, numerical and alpha-numerical data are declared in the **parameter declaration**. Example:

```
parameter a := 10;                --a singleton
parameter b{i,j,k} := Rnd(5,10); -- a table
```

The first declaration assigns the value 10 to the identifier `a`. The second declaration assigns a random number to every triple (i, j, k) of `b`.

Data can also be alpha-numerical. Strings are declared within the parameter statement by adding the reserved word `string` as a type attribute. Example:

```
string parameter
  c := 'Hello';
  d := c & ' world';
  text :=  'A whole phrase might be \
            assigned to a string variable\n';
  e{i} := 'X_' & i;
```

The first declaration defined such a string, the second is a concatenation, the third is a longer string broken into two lines containing a linefeed character. The last is an array of strings.

Data can also be added directly to a LPL model in two special table formats: the *format A* and the *format B*. The first is for full table data, the second is more convenient for sparse tables. Example:

```
parameter a{i,j} := <..table data..>;
```

6.1. Format A

In the *format A*, the table data are included in brackets after the assignment operator. A example is:

```
set i := [1..6];
parameter a{i} := [10 20 30  . 7 9];
```

Note that the set `i` has 6 elements. The parameter vector `a{i}`, therefore contains also 6 entries. They are written in lexicographical ordering. *All* data elements must be written. Optional commas can be used to separate the single data. A data can also be replace by a dot (.) which is replaced by the *default value*. If no default value is given it is zero/empty string. In the example above, this means that `a[4]:=0` and `a[1]:=10`, for example.

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

```
string parameter s{i,j} :=
      ['one' 'two' '3'  'four' '5' 'six'];
```

Note that the format A always defines full tables. All entries must be written sequentially one after the other.

The format A is also used for defining lists of elements of sets or tuples sets.

```
set i:=[A B C];    // a set of two elements
    j:=[1 2 3 4];  // a set of four elements
    IJ{i,j} := [ (A,1) (B,2), (B,3), (B,4), (C,4)];
```

The last is a compound (tuple-list set (see below). And it can also be written as:

```
      IJ{i,j} := [ (A,1) (B,*) 2 3 4, (C,4) ];
```

Stars are used like place-holders, see below in format B section *template option*.

This format A can also be used to assign sparse numeric tables as in:

```
set i:=[A B C];    // a set of two elements
    j:=[1 2 3 4];  // a set of four elements
parameter a{i,j} := [ (A,1)=1 (B,2)=2, (B,3)=4, (B,4)=7, (C,4)=-55];
```

A somewhat shorter way is to use format B to assign the same sparse table (see below):

```
set i:=[A B C];    // a set of two elements
    j:=[1 2 3 4];  // a set of four elements
parameter a{i,j} := /A 1 1, B 2 2, B 3 4, B 4 7, C 4 -55/;
```

6.2. Format B

The table *format B* is a powerful format specification to define **sparse** multi-dimensional 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, and (4) the template-option. The table specification begins with a slash (`/`) and ends with a slash (`/`).

6.2.1 The list-option

Using the list-option, the entries in the tables are list in any order. An entry consists of the tuples list of the set elements followed by a value. Example:

```
set k  := [A B C D E] ;
parameter v{k} := /A 1, C 3, E 5/;
```

The set `k` contains 5 elements. The sparse table `v{k}` contains three entries: We have `v[1]:=1` (`v['A']:=1`), `v[3]:=3`, and `v[5]:=5`. Note that `v[2]` and `v[4]` are not defined. (An entry that is not defined can be accessed by an expression – it returns the default value.) Note that the sparse entries consists of the set element name followed by the value. An optional comma may separate two entries.

If the table is two-dimensional, then the two set elements are written followed by a value. The order in the entries is not important. Example:

```
set i  := [A B C] ;
set j  := [X Y Z] ;
parameter w{i,j} := /A X 1, C Y 3, A Z 5/;
```

If the table is a relation (tuple-lists), then the value is missing, only the tuples are written and one must use *format A*. Example:

```
set i  := [A B C];
set j  := [X Y Z];
set t{i,j} := [A X, C Y, A Z];
set s{i} := [B C];
```

Note that the last declaration `s{i}` defines just a subset of `i`.

The declaration of `t{i,j}` can also written by enclosing the tuples within parentheses as in (comma are optional):

```
set t{i,j} := [(A X), (C Y), (A Z)];
```

We summarize: Sparse tables can be declared just by listing the element tuples in any order. If the table is numerical or alpha-numerical, 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 sub-tables separately. Suppose, the following sparse 14×14 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 relatively dense sub-tables 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 blocks begin with a colon (`:`) followed by a list of elements of the last set (`cols`) in any order. A second colon terminates the set element 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 block 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 block 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. Consider the following table `e{i,j,k}` which was declared as:

```
parameter e{i,j,k}:= /i1 j1 k1 1, i2 j1 k 2, i3 j2 k3 3, i3 j2 k4 4/;
```

Using the colon-option, the table can be declared as:

```
set e{i,j,k} := / :  k1  k2  k3  k4 :
               i1 j1  1   .   .   .
               i2 j1  .   2   .   .
               i3 j1  .   .   3   .
               i3 j2  .   .   .   4  /;
```

Another way to represent the relation is:

```
parameter e{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 multidimensional table into non-sparse blocks of the same dimension as the original table. Sometimes it is useful to partition the table into blocks of lower dimension. To do this, the modeler can use the *template-option*. Consider again our 14×14 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 a list of elements or stars separated by commas, and terminates with a right bracket. The number of elements or stars in the template must correspond to the number of indices used in the index-list `{...}`. 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 14×14 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 sub-table) 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. Let a four-dimensional $2 \times 3 \times 4 \times 5$ sparse table `d{i,j,k,m}`be as follows:

```
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 /;
```

Using the template-option, it can be written as: 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  /;
```

Or as:

```
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, multidimensional table down into blocks (sub-tables) of different dimensions. And the syntax is straightforward and simple. See a model example at multmip1[1] and the data in file multmip1.txt[2]

- If the sparse table contains just some unrelated tuples, the list-option is an appropriate mean.

[1] https://lpl.matmod.ch/lpl/Solver.jsp?name=/multmip1
[2] https://matmod.ch/lpl/doc1/multmip1.txt

- If the table can be broken down into homogeneous, two-dimensional blocks, the colon-option is your choice.
- If the table can be broken down into blocks of lower dimensional tables, the template-option is the right choice.
- If the blocks 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.

Chapter 7

Variables and Constraints

This Chapter explains the use of *variables* and *constraints* in a model.

7.1. Variables

All unknowns used in an 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**. They can also be assigned exactly like parameters. The difference is that their values are under the control of a solver, that is, if a solver is called they their values change. The declaration of variable is as follows:

```
variable
  x;          -- a single variable declaration named 'x'
  Name;       -- another single variable 'Name'
  y{i,j};     -- an indexed variable 'y' (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. Lower and upper bounds on variables directly produce a bound constraint, and an integer attribute produces a mixed integer model. Note that the default lower bound of variables for linear and

(convex) quadratic models is zero, and for non-linear model it is minus infinity. The default upper bound is always plus infinity. Variables and parameters are almost the same: they have both numerical values, which can be used within other expression. Example:

```
integer variable
  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, 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`, it gets a start value of 3 that will be modified by the solver, eventually. In the definition of `z`, a constraint `z=x+y` is generated. However, LPL substitutes the variable `z` by `x+y`.

A special kind of integer variables are the `alldiff` variable. They define a permutation. The permutation can be partitioned or repeated depending on the Quote attribute (full fetched model examples can be found in [4]):

```
alldiff x{i};                    //a simple permutation
alldiff x{i} 'subset';           // a subset of the permutation
alldiff x{k,i};                  // a repeated permutation
alldiff x{k,i} 'partition';      // a partitioned permutation
alldiff x{k,i} 'set partition'; // an unordered partitioned permutation
alldiff x{k,i} 'cover';          // an covered partitioned permutation
```

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 t :  x-y >= 0;
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 in the first expression `x-y`, then it is interpreted as being greater or equal to zero. Hence, the first two constraint are identical:

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;
```

This will be translated by LPL into:

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

A constraint can be made inactive by adding the keyword `frozen` as an attribute. An inactive constraint does not produce any output to the LPO-file and to the solver, only the active once does. An inactive constraint can be made active by an `Unfreeze` function call. It can be inactivated again with a `Freeze` function call. Example:

```
constraint
  r1 : x+y+z <= a;
  r2 : x-y-z > b;
  r3 frozen: 2*x-y-1 < d;  --inactive constraint
  ....
Unfreeze(r3);          -- reactivates the constraint r3
```

A constraint expression can contain also the comma operator. This allows the run immediately before the constraint is generated the run some code for data manipulation. An example is given in model learn48[1].

[1] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn48

7.3. The Objective Function

The objective function is defined in a solve statement:

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

The solve statement within model X generates a model instance – that is, it collects all constraints within X – calls a solver and reads the solution back into LPL in the meantime the execution of the model run is stopped.

By default, all constraints (whether defined before or after the solve statement) within the model, where the solve statement occurs, are sent to the solver. Note that no constraints in other models even within submodels are sent to the solver, even if the submodel would have be called. There is another mechanism to collect constraints: the `subject_to` attribute. It can be used together with the solve statement and lists all constraints that must be sent to the solver. Example:

```
maximize profit : x + y + z  subject_to mod1, mod2, mod3.const1, ~mod2.const2;
```

This solve statement collects all constraints within model `mod1` and model `mod1` together with the constraint `mod3.const1`, but excludes the single constraint `mod2.const2`.

The solve 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` function. 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. The different problem types are listed in the function GetProblemType.

LPL checks whether all constraints are linear. If they are, it returns `LP`, or `MIP`. If the objective function is quadratic and all constraints are linear,

LPL returns QP, with integer variables it returns iQP. If some constraints have quadratic expressions, then it returns QCP, and with integer variables it returns iQCP, if the constraints are non-convex then the type is NQCP or iNQCP. In all other cases it returns NLP or iNLP with integer variables. The type PERM is a special problem type (see Chapter 9.2.3) and can be "solved" using the integrated TABU-heuristic solver. PERM problems are models in which: (1) Only one variable is declared as: alldiff, (2) Only an optimization function without any other constraints is declared. The problem is to find a permutation which optimizes the objective. Many problems in scheduling can be formulated as permutation problems. LPL supports this model class with its integrated heuristic TABU solver. More complicated permutation problems cab be solved with LocalSolver.

A quote attribute 'relax' can be used in a objective function:

```
minimize obj 'relax': ...
```

This means that all integer and binary variables are transformed to real variables. A IP model becomes a LP model and the LP relaxation is solved.

A quote attribute of 'nosolve' means that the model is not solved.

If the quote attribute is '∼cbcSol' then this means that the model cannot be solved by the free CBC solver (this is basically for serial model testing for the case books).

7.4. Logical Constraints

Logical constraints are supported by LPL. That is, all logical operators are allowed in the constraints. These constraints are translated by default into a MIP (mixed integer) formulation. Suppose one wants to impose the following constraint to an otherwise mathematical model, where x is a variable of type double:

```
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 linear formulation of a model constraint, an LP/MIP-solver would not be able to solve the model. Therefore, LPL translates it automatically 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, whatever d is, x can only be in the two intervals $[0 \ldots 10]$ or $[20 \ldots 100]$. (Note that the upper bound (100) on x is important and necessary).

In the above example, LPL introduces automatically a binary variable. The user also can explicitly introduce "indicator" (or binary) variables as follows (note the colon is the same as an assignment :=) :

```
binary variable d: x>0;
```

This declaration introduces a binary variable and the constraint $d \rightarrow x > 0$ ("If d is true then the quantity x must be strictly positive").

Chapter 8

Input and Report Generator

Two powerful functions (**Read** and **Write**) give the user a tool to read data from and write data to (1) plain text files, (2) databases, (3) Excel spreadsheets, and (4) snapshots. The **Write** function also allows one to generate complex reports using the FastReport library. Together, they cover the Input and Output functionality in LPL.

Both functions have similar structure, and reading/writing to different kind of files also have unified syntax. Reading/Writing in LPL is somewhat different from other languages. Each reading/writing call opens and closes the file each time, there is no open or close function. However, the file name of the previous reading and writing instruction is retained, so they must not be repeated in each Read/Write call.

First, the semantics of the *Read* function are explored.

8.1. The Read Function

Each Read call is based on the same concept: from text files it reads a text line[1], from databases it reads a record, and from Excel spreadsheet it reads a consecutive number of horizontal cells. Hence, reading is based on *row-wise reading*. The second concept is *table based reading*: using index-sets, several

[1] A *text line* is a sequence of characters ending with a new-line character

lines, records, or spreadsheet rows can be read in a single call. In text files, this concept is called a *table* (such a part of the text file), in database it is a *table*, and in spreadsheet this concept is called *range*.

8.1.1 Reading from Text-files

The syntax of a text read call is as follows:

```
Read('prefix filename , table , delimiter , ignore',
                          list-of-<destination=source>);
```

The first parameter is a string, consisting of at most 4 parts: (1) The first is **prefix filename**, it has already been explained here. (2) The second part is **table** and consists of a optional *block number*, an optional *number of skipping lines* and an optional *block begin and block end delimiters*. The block number begins with a `%` followed by an integer, the number of skipping lines begins with a `;` followed by a integer, and the block delimiters begins with a `:` followed by the block begin delimiter and an optional block end delimiter separated by a `:` . An example of the **table** parameter is:

```
%2;5:Table:Endtable   -- an example
    '%2'        (block number is 2)
    ';5'        (number of skipping lines is 5)
    ':Table'    (block begin delimiter , reading begins on next line)
    ':Endtable' (block end delimiter , line to end reading)
```

A *block* is a part of a text file that begins with the *block delimiter* at the beginning of a line and ends with the same block begin delimiter (or the block end delimiter) at the beginning of a line or end with the end of file. If the file does not contain any block delimiter then the whole file is considered as a single (first) block.

The *block number* tells which block within the file has to be read. A block number of `%3` means to jump to the 3-rd block, that is, the file part that begins with the 3-rd occurrence of the block delimiter within the file and ends with the 4-th occurrence of the block delimiter or the end of file. The *number of skipping lines* instructs to skip lines within the block at the beginning of the block. A *number of skipping lines* of `';4'` means to skip 4 lines within the block and to begin reading at the 5-th line within the block. The number of skipping lines can be -1 (but not less) meaning that the line with the block begin delimiter is part of the next reading instruction (see model tutor07e[2]).

[2]https://lpl.matmod.ch/lpl/Solver.jsp?name=/tutor07e

The filename and the block delimiter string are remembered from previous read calls and do not need to be repeated over and over again. A missing *number of skipping lines* or a missing *block number* is considered as 0.

Now, an *Read* function-call opens a file, reads a text line from a text file, and closes the file. Text lines beginning with `/*`, `--`, `//`, or empty lines[3] are skipped automatically – we call them *unreadable lines* (all others are *readable lines*. Hence, unreadable lines *cannot* be read by LPL's *Read* function.

After the line was read it is manipulated depending on the third parameter **delimiter**. The delimiter is a string containing all characters that are considered as token-separators. The line is "tokenized", that is it is broken into parts – called *tokens* delimited by one of the *delimiter* characters. The tokens are numbered beginning with one from left to right – the *token number*. By default, the three characters *comma*, *tab*, and *space* are considered as token-separators. Note that several consecutive spaces (Unicode 32) are always considered as a single character. The fourth parameter is **ignore**: it is a list of single characters that are ignored by the reader – they are translated into spaces.

The next parameters in the *Read* function have the syntax: **destination** = **source**.

The part `destination` is an LPL identifier (typically a parameter, a set) defining the receiver of the data. This is followed by an option equal sign and a integer – the *token number* (the `source`), indicating the token to be assigned to the `destination`. If the destination is a numerical then the token is automatically translated into a numerical type. If this is not possible, zero is assigned (no error occurs). Some Examples:

```
set i; parameter c{i};
parameter b;
string parameter a;
Read('file.txt' , a , b);
Read(';1' , a=2 , b=4);
Read('file.txt,%3;2:Table,\n', a);
Read('text.txt,%1;1:Table,\t ', a=2 , b=7);
Read('file.txt,,\t', {i} i);
```

In the example, a set `i`, a numerical parameter `c`, a numerical parameter `b` and a string parameter `a` are declared.

[3] a line that contains only white space characters such as spaces, tabs, and others, is considered as an *empty line* by LPL

- The first *Read* opens the file with the name `'file.txt'`, reads the first readable line, tokenizes it into the tokens delimited by a comma/tab/space char, assigns the first token to `a` and the second token to `b` (while type-casting the second to a number).

- The second *Read* opens the file with the name `'file.txt'` (remembered from the previous *Read*), reads the second readable line – skipping one readable line (`;1`), tokenizes it into the tokens delimited by a comma/tab/space char, assigns the second token to `a` and the fourth token to `b` (while type-casting the second to a number).

- The third *Read* opens the file with the name `'file.txt'`, jumps to the 3-rd block, reads the third readable line – skipping two readable lines (`;2`), tokenizes it into the tokens delimited by an newline char (that is, the whole line is takes as is), assigns the first (and unique) token to `a`.

- The fourth *Read* opens the file with the name `'text.txt'`, jumps to the 1-st block, reads the second readable line – skipping one readable line (`;1`), tokenizes it into the tokens delimited by an tab char, assigns the second token to `a` and the 7-st token to `b`.

- The last *Read* opens the file with the name `'file.txt'`, reads the first readable line, tokenizes it into the tokens delimited by an tab char, runs through the tokens and assigns them to the elements of set `i`. Note that we use `{i}` to force the repeated assignment on the read line.

Note that the reader is quite error tolerant. If, for example, the corresponding line does not exist, because it reached already the end of the file, then nothing is assigned; if a token cannot be converted into a number, zero is assigned; If the corresponding token does not exist, nothing is assigned.

A *Read* call can be indexed, meaning that all readable lines within a block will be read sequentially. In the following *Read* call, the second block lines are read, where the first token is assigned to `i` and the third to the parameter `a` (for a complete example see the model tutor07[4]:

```
set i;
parameter a{i};
Read{i}('file.txt,%2' , i, a=3);
```

[4] https://lpl.matmod.ch/lpl/Solver.jsp?name=/tutor07

8.1.2 Reading from Snapshots

A snapshot is LPL internal file format that can be generate by a snapshot *Write* (see below). The whole internal data structure is stored in a file – a "snapshot" of LPL's internal data store is taken at the moment of writing – and can be re-stored using a snapshot reading. A snapshot reading/writing only has one parameter: the snapshot file name. LPL automatically does a snapshot read with a file extension of `.sps` or a *prefix* of `sps:`.

```
Read('file.sps');
Read('sps:file');
Read('+:file.sps'); --read cumulatively
```

Snapshots can be read cumulatively, that is, several snapshots can be read and the store is extended by each snapshot read. All tables are augmented by an additional index-set `_SNAP_`. For a complete model example see learn33[5].

8.1.3 Reading from Databases

A *Read* call reads a record from a database table. The syntax is as follows:

```
Read('prefix filename , table', list-of-<destination=source>);
```

The *prefix* has already been explained here. The `filename` consists of a *database connection string* and a database file name. It is supposed that the corresponding database driver has been installed. For example for accessing a *MS Access* file, it is necessary that Microsoft Jet Engine is installed (this is automatically the case under Windows). If one needs to access a MySQL table then the corresponding ODBC-driver must be installed.

The part `table` contains a database table name or a SQL SELECT statement.

The next parameters in the *Read* function have the syntax:

```
destination = source.
```

The part `destination` is an LPL identifier (typically a parameter, a set) defining the receiver of the data. This is followed by an equal sign and a quoted field name within the database table (the `source`). The fieldname can also be replaced by a integer defining the field number. If the destination is a numerical then the token is automatically translated into a numerical type. If this is not possible, zero is assigned (no error occurs). Some Examples:

[5] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn33

```
set i; parameter b{i};
string parameter DB:=
   'Provider=Microsoft.Jet.OLEDB.4.0;Data Source=tutor.mdb';
Read{i}(DB&',iTable', i='ID', b='Price');
```

The examples declares a set `i` and a parameter `b{i}`. Next follows a string parameter `DB` defining the connection string for the MS Jet Engine and the database file. The *Read* call says to read from the database table `iTable` the two fields with name `ID` and `Price`. The two data are assigned to the set `i` and the parameter `b`. Note that *all* records within the table are read sequentially – since this is an indexed *Read*. For a complete model example see tutor07b[6] and tutor07a[7] (reading from Excel in a database "style").

8.1.4 Reading from Excel

A *Read* call reads a record from a Excel Sheet. The syntax is as follows:

```
Read('prefix filename , table' , <destination>);
```

The *prefix* has already been explained here. The `filename` consists of a Excel filename (with extension `.xls`).

The part `table` contains an Excel "range", such as `[Sheet1$C3:N12]`, denoting the cells from `C3` (top/left) and `N12` (bottom/right).

The next parameters in the *Read* function have the syntax: `destination`. The part `destination` is an LPL identifier (typically a parameter, a set) defining the receiver of the data. Some Examples:

```
set i;
Read('tutor.xls,[Sheet1$C3:L3]', {i} i)
```

The examples declares a set `i`. The *Read* call says to read from the range `[Sheet1$C3:L3]` sequentially the cell content is assigned to the set elements. For a complete model example see tutor07a[8] and tutor07a1[9][10]

[6] https://lpl.matmod.ch/lpl/Solver.jsp?name=/tutor07b
[7] https://lpl.matmod.ch/lpl/Solver.jsp?name=/tutor07a
[8] https://lpl.matmod.ch/lpl/Solver.jsp?name=/tutor07a
[9] https://lpl.matmod.ch/lpl/Solver.jsp?name=/tutor07a1
[10] LPL uses the 32bit jet engine of Microsoft to work with Excel. Since Microsoft announced that no 64bit version of the OLE Jet Engine will be provided, these models only work with LPL 32bit version.

8.2. The Write Function

The function *Write* is a powerful mean to write to text files, databases, Excel sheets and snapshots in various forms. It also is a mean to generate sophisticated and professional reports. Each *Write* instruction opens a file, writes data to it, and closes it automatically. There are no open or close file instructions.

8.2.1 Writing to Text-files

The syntax of a text write call is as follows:

```
Write('prefix filename', 'formatting', list-of-<sources>);
```

The first parameter is a string, consisting of the *prefix* and the *filename* have already been explained here. The first parameter can be missing. In this case the data are written to a default file, called the NOM-file, or to the file specified in the previous *Write* call. Since the second parameter is also a string, LPL will recognize that the first parameter is missing by investigating it: If it contains a newline character, or the char `%` or the *Write* has only one parameter then the first prefix-filename parameter is missing.

The second parameter `formatting` is also a string and this string is written to the text file. If it contains *place-holders* they are replaced by the `list-of-<sources>`. There are many different place-holder each beginning with a character `%`. The `list-of-<sources>` are expressions that are evaluated and the result is placed in a place-holder within string `formatting`. Examples

```
set i:=1..10; parameter c{i}:=i^3;
string parameter a := 'Hello world';
parameter b := 1234;
Write('A small text\n');
Write('The message is: %s\n' ,a)
Write('file.txt','The result is: %d\n', b);
Write{i}('i = %2s , i^3 = %4d\n', i , c);
```

In the example, a string parameter `a` and a numerical parameter `b` are declared and defined.

- The first *Write* opens the default file NOM-file, jumps to the end of the file and appends the string `'A small text'` and adds a newline char (`\n`), it then closes the file again.

- The second *Write* also opens the default file NOM-file, jumps to the end of the file and appends the string `'The message is: Hello World'` and adds a newline char (`\n`), it then closes the file again.

- The third *Write* opens the file with the name `'file.txt'`, jumps to the end of the file and adds the string The result is: 1234' and a newline char and closes the file.

- The fourth *Write* opens the default file NOM-file, jumps to the end of the file and appends 10 lines as follows:

```
i =  1 , i^3 =    1
i =  2 , i^3 =    8
i =  3 , i^3 =   27
i =  4 , i^3 =   64
i =  5 , i^3 =  125
i =  6 , i^3 =  216
i =  7 , i^3 =  343
i =  8 , i^3 =  512
i =  9 , i^3 =  729
i = 10 , i^3 = 1000
```

  it then closes the file again.

The `formatting` string can contain place-holder having the syntax as follows:

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

A format specifier (for numerical data) begins with a `%` char. It follows in this order:

`0`	an optional zero to fill leading chars with zero (otherwise they are spaces)
`-`	an optional left justification indicator
`width`	an optional width specifier (an integer)
`.prec`	an optional precision specifier (`prec` is an integer)
`type`	the conversion type character (`type` is one or two characters)

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.

`D` same as d, but zero values are not printed.

`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` Format for the value of Boolean: "T" or "F" (not in Delphi).

`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.

`F` same as f, but zero value are not printed

`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.

For a model example see tutor08e[11].

[11] https://lpl.matmod.ch/lpl/Solver.jsp?name=/tutor08e

8.2.2 Writing a Snapshot

A snapshot file (see snapshot reading) can be written using the instruction *Write* as follows:

```
Write('file.sps');
Write('sps:file.abc');
```

The unique parameter is the file name. Note that the file name must have extension `sps` or the prefix must contain `sps:`. For a model example see learn33[12].

8.2.3 Writing to Databases

The syntax of a database write call is as follows:

```
Write('prefix filename' , list-of-<destination=source>);
```

The first parameter is a string, consisting of the *prefix* and the *filename* have already been explained here. Note that the filename must contain a *database connection string*.

The other parameters `list-of-<destination=source>` contains a quoted field-name as a *destination* and an arbitrary expression as *source*. The source is evaluated and assigned to the field within the database. Examples:

```
string parameter
  DB:='Provider=Microsoft.Jet.OLEDB.4.0;Data Source=tutor08out.mdb';
Write(DB&',iTable', 'Robots'='Robot1', 'Price'=234567);
Write('-:'&DB&',iTable', 'Robots'='Robot1', 'Price'=234567);
Write('+:'&DB&',iTable', 'Robots'='Robot1', 'Price'=234567);
```

The parameter `DB` defines the database connection string and the database file name.

- The first *Write* opens the database `tutor08out.mdb` tries to write data into the table `iTable`. If the database or the table does not exist, the run is stopped with an appropriate error message. Otherwise, LPL looks for the primary key field `Robots`, jumps to the corresponding record and fills the data (here field `Price`). The database is finally closed.

[12] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn33

- The second *Write* opens the database `tutor08out.mdb` clears all records of the table `iTable` before writing a new record. The database then is closed.
- The third *Write* opens the database `tutor08out.mdb` tries to write data into the table `iTable`. A new record is added to the table. The database then is closed.

Creating a Database

Using the *Write*, one also can create a database and its tables on the fly. There is only a different prefix in the file name. Example:

```
string parameter
  DB:='Provider=Microsoft.Jet.OLEDB.4.0;Data Source=tutor08out.mdb';
Write{i}('&:'&DB&',iTable', 'Robots'=i, 'Price'=Price,'Ordered'= Ordered);
Write{i}('*:'&DB&',rTable', 'Robots'=i, 'Quan'=Robots);
```

The first *Write* has a prefix of `&:`, which means to create a new MS Access database `DB`. Note that an existing database file will be deleted. At the same time a new table named `iTable` is added to the database with the fields `Robots`, `Price`, and `Ordered`. The type of the fields correspond to the type of the LPL specification: sets and string parameter generate string types in the database. There are some convention about the field names:

- If the field name begins with a capital 'I' then this field is considered as a primary key field, and in addition to the field a primary key will be generated.
- If the field name begins with a character '_' (underscore) then the field is an indexed field. A index will be generated too.

For a complete model example see tutor08b[13].

Creating Reports

LPL includes the functionality of a professional report generator, called **FastReport**. Creating a report in LPL is very similar to create a database. There is only a small difference in the first string parameter: The filename must be concatenated with a comma and a second file name which is the template file name for a report. Example:

[13] https://lpl.matmod.ch/lpl/Solver.jsp?name=/tutor08b

```
string parameter
  DB:='Provider=Microsoft.Jet.OLEDB.4.0;Data Source=tutor08out.mdb';
Write('&:'&DB&',tbl,tmpl' , 'a'='A Report', 'b'='Title');
Write('&:'&DB&',tabl,*' , 'a'='A Report', 'b'='Title');
Write('&:'&DB&',tble,*,*.pdf' , 'a'='A Report', 'b'='Title');
```

In the first *Write* a new database `tutor08out.mdb` is created with the table `tbl` and the fields `a` and `b` – as explained in the previous section. In addition, the first parameter contains the part `',tmpl'`. With this additional comma separated part, LPL will generated a new report template file with the name `tmpl.fr3`. This is a FastReport generator compliant template file. If this file already exists then it will *not* be created once again. The name can also be replaced by the character star (`*`), meaning that the template file name is the same as the table name with extension `fr3`. The template name of the second *Write*, therefore is `tbl.fr3`.

Note that LPL works hard to generate a first useful template using the fields and the fieldnames if the database table to place the data on the template page. Nevertheless, the user can open the template file using the template designer of FastReport and modify every single item within the page. LPL modeling system (lplw.exe) contains a menu item to access directly to the designer (menu *View / Open Report Designer*).

An additional part separated again by a comma can be added to the first parameter. In the third *Write* in the previous example it is `*.pdf`. This is the report file name. If this part is used then the so far generated template pages are collected and a complete report file is written. In the example the file `tble.pdf` – a PDF file – is generated automatically. Depending on the file name extension various file type can be generated. If the extension are as follows:

`view`	a preview is generated
`pdf`	a PDF file is generated
`rtf`	a RTF file is generated
`html`	a HTML file is generated
`txt`	a simple text file is generated
`jpg`	the pages are composes as JPEG files

A complete model example is given in tutor08c[14].

[14] https://lpl.matmod.ch/lpl/Solver.jsp?name=/tutor08c

Chapter 9

Miscellaneous

This chapter contains miscellaneous topics.

9.1. File Inclusion

Files can be included into the source file using the directive `/*$`. There are two directives.

```
/*$I 'filename' */
/*$E 'filename:int:delimiter' */
```

A compiler directives can be placed anywhere within the model where a normal comment is legal. The `filename` is a string denoting a file. The filename can be parameterized with a `@` character. The character `@` is replaced by a *string parameter* that was marked with the `'incl'` quote attribute. Of course, the parameter must be assigned at parse-time otherwise it is empty.

9.1.1 File Include /*$I*/

The directive `/*$I` 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 source may be split in different LPL files. For an example

see the model tutor06[1]. Nested include files are also possible up to a level of 5. The filename must be within single quotes.

The include file name can contain a wildchar * (star), which will be replaced by a real filename if just extists. If the file does not exist then a warning ist generated, but not error occurs.

9.1.2 File Part Include /*$E*/

The directive `/*$E` redirects the LPL scanner to read from another file part at this point and – at the end of that file part – reading continues from the calling file. In contrast to `/*$I`, this directive reads only a part of a file between two delimiter in a similar way then does the text reading instruction. Example:

```
/*$E 'MA.pla:8:Table' */
```

This instruction opens the file `MA.pla` and reads from the 8-th occurrence of the delimiter "Table". From there on, it copies the part to a new file `M08.pla-8` (filename composed from filename and the int separated by a dash), a file created on-the-fly. The scanner then opens this file `M08.pla-8` and continues scanning the code from within this file.

9.2. The Solver Interface

LPL can communicate with various solvers – commercial and free once. LPL's solver interface is guided by three functions:

An LPL model can also contain function calls. They define various directives to control behavior of the LPL compiler, solver or other part of LPL. They are as follows:

`SetServer(URL)`	Sets the URL of the Internet server of LPL.
`SetSolver(SIP[,SO])`	Sets the Solver Interface Parameters (SIP), and sets the Solver Options (SO)
`SetSolverList(Sl)`	Sets the Solver List (Sl), see SetSolverList

All function arguments are strings.

[1] https://lpl.matmod.ch/lpl/Solver.jsp?name=/tutor06

9.2.1 The Solver Interface Parameters (SIP)

LPL has a flexible and transparent solver interface. The communication between LPL and a solver is basically determinate by 22 solver interface parameters (SIP) placed in the single string and separated by commas within the string. For later reference, we call these parameters SIP1, SIP2, ..., and SIP22).

For several solvers the SIP is already predefined in the file `lplcfg.lpl` as string parameters; such as:

```
string parameter gurobiLSol :=
    ',,lib:gurobi95,gurobi.prm,,,,,,,,,,,,,LP;MIP;QP;iQP;QCP;iQCP;NQCP;iNQCP';
string parameter InterSol := 'l,,Internet';
string parameter mySolver :='l,,Internet7,https;//lpl.unifr.ch');
```

In the first declaration (`gurobiLSol` – the interface to the gurobi95 library), SIP1 and SIP2 are empty (the SIP begins with two commas, and the third (SIP3) is `lib:gurobi95`. The second declaration `InterSol` is the interface to the default LPL Internet solver. That is, the problem is sent to a LPL Server (defined in the function SetServer), and the server chooses the appropriate solver if it exists. The last definition (`mySolver`), sends the model to the server defined as "https;//lpl.unifr.ch" and chooses the solver number 7 (which is "mipcl", see file lplcfg.lpl at the last lines) if it is installed.

Having predefined several solvers in such a way (preferably i the lplcfg.lpl file), the call to the function `SetSolver` (within the model) becomes:

```
SetSolver(gurobiLSol);  -- sets the gurobi as solver
SetSolver(InterSol);    -- sets the Internet solver
SetSolver(mySolver);    -- sets another Internet solver
```

It is also possible to predefine the solvers for all problem types (see GetProblemType) using the function SetSolverList.

The 22 Solver Interface Parameters (SIP1 to SIP22) are defined as follows:

1. **SIP1**: Indicates what to do before the solver is called. It is:

`(empty)`	nothing to do (the default)
`0`	(zero) no model transformation necessary
`l`	the LPO-file is generated before solving
`m`	the MPS-file is generated before solving
`s`	the LSP-file is generated for the LocalSolver before solving
`e?`	the EQU-file is generated before solving – for `e?` see compiler switch `e?`
`@*`	(* is a string) the string * is interpreted as a program name to be executed as a child process (while LPL – the parent – waits)

Example: If the SIP1 consists of the string 'lm', then LPL generates the LPO-file and the MPS-file just before it calls the solver.

2. **SIP2**: Indicates what to do after the solver has terminated. It is:

`(empty)`	nothing is done (the default)
`s`	the SOL-file and the DUA-file are read after solving
`l`	the LPX-file is read after solving (from Internet solver)
`o`	the LSX-file (output from LocalSolver) is read after solving
`e?`	the EQU-file is written after solving – for `e?` see compiler switch `e?`
`@*`	(`*` is a string) the string * is interpreted as a program name to be executed as a child process (while LPL – the parent – waits)

Example: If the SIP2 consists of the string 's', then LPL reads the SOL-file just after the solver terminates.

3. **SIP3**: The program or the library that is called as solver. Note that strings beginning with 'lib:' will call a dynamic link library. First, this substring is removed and the string is concatenated with '.dll' or '.so' depending on which OS the solver is called (Windows or Linux). Note also that '*' is a wild-char. For example: if the string is: 'lib:gurobi90', then under Windows the library 'gurobi90.dll' is searched for, under Linux the library 'libgurobi90.so' will be searched for. The implemented solver are as follows :

`lpl-lp`	LPL's own LP solver is called (the default)
`perm`	LPL's internal permutation solver is called
`lib:gurobi*`	the dynamic link library of gurobi is called
`lib:cplex65`	The dynamic link library of cplex65 is called
`lib:cplex*`	The dynamic link library of cplex is called
`lib:mops*`	The dynamic link library of MOPS solver is called
`lib:lindo*`	The dynamic link library of Lindo's solvers is called
`lib:EMO_Mipkit*`	The dynamic link library of the MIPKIT solver is called
`lib:mosek*`	The dynamic link library of MOSEK solver is called
`lib:xprs*`	The dynamic link library of Xpress solver is called
`lib:xa*`	The dynamic link library of XA solver is called
`lib:loqo*`	The dynamic link library of the loqo is called
`lib:conopt*`	The dynamic link library of the conopt is called
`lib:knitro*`	The dynamic link library of the Knitro is called
`lib:cfsqp*`	The dynamic link library of cfsqp solver is called
`lib:OptQuest`	The dynamic link library of OptQuest is called
`Internet*`	The model is sent to an Internet server ('*' is a digit, see above)
`<empty>`	nothing is called (same as `nosolver`)
`nosolver`	nothing is called (no solver is called)
`<otherwise>`	the program specified by the parameter is called

4. **SIP4**: The name of the Solver Options Mapping File (SOMF) (explained below), if SIP3 is `'Internet'` then this is an optional server address (see above).

5. **SIP5**: The name of the Solver Parameter File (SPF) (explained below),

6. **SIP6**: The default Solver Options (SO). The different options are separated by a semicolon. Each solver option must be defined as a name value pair as follows:

   ```
   Name=Value
   ```

   `Name` is a (case-sensitive) string defining the option, for example in cplex the option to define the time limit for the solver is `TILIM`. The `Value` is a value (either an integer, a double or a string) setting that option to this value. For example to set the solver time limit in cplex to 60secs the string is:

   ```
   TIMLIM=60
   ```

   `Name` is defined either in the SOMF file (see below) or they are passed directly to the solver.

7. **SIP7**: The string replacing `%3` within the SIP6, if the model has to be maximized, if the string in SIP7 is 'max2min' then the MPS-file modified to a minimizing problem, this for solvers that do not have a parameter for minimizing/maximizing the objective (this is particularly the case for mps_mipcl, scip, andgurobi_cl),

8. **SIP8**: The string replacing `%3` within the SIP6, if the model has to be minimized,

9. **SIP9**: The filename from which LPL has to read the solution (SOL-file),

10. **SIP10**: An integer indicating on which physical position on a line in the SOL-file the first character of the variable name is found. If this parameter is missing or zero then the SOL-file has the format "X>nr> <space> >value>" beginning with the second line (for gurobi and mps_mipcl) when solution is passed by files. (In Gurobi normally the passing is in memory anyway).

11. **SIP11**: An integer indicating on which physical position on a line in the SOL-file the first digit of the value is found,

12. **SIP12**: An integer indicating the length of the numerical value of the variable in the solution file,

13. **SIP13**: The filename from which LPL has to read the dual values (DUA-file),

14. **SIP14**: An integer indicating on which physical position on a line in the DUA-file the first digit of the dual value is found,

15. **SIP15**: A substring searched in the SOL-file to indicate that the model is optimal,

16. **SIP16**: A substring searched in the SOL-file to indicate that the model is infeasible,

17. **SIP17**: A substring searched in the SOL-file to indicate that the model is unbounded,

18. **SIP18**: A substring containing the string names of problem types (see GetProblemType), separated by semicolons, that this solver can solve. An empty substring means that the solver has no restriction.

19. **SIP19**: The string replacing `%4` within the SIP6, if the model is a MIP (actually used only in Xpress).

20. **SIP20**: 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. **SIP21**: A solver dependent parameter: (time interval in milli-secs between two calls of the solver callback) (only for library solvers). (default 3000ms).

22. **SIP22**: A solver dependent parameter: (time interval in milli-secs between two model call, called from the solver callback if an (integer) solution is found) (gurobi only). The LPL modeler defines a submodel with an arbitrary name X. If this model is defined it will be called from within the solver callback at this interval of time – provided that a solver option 'CALLBACK=X' has been added to the solver options list. If X is not a submodel name within the LPL code or if X is an empty string (either CALLBACK=" or CALLBACK=) then no model is called (no error occurs). This option is interesting for adding lazy constraints or when the solution process takes a long time as in many MIP models, but we want to get an intermediary result (using Writes). Of course, the submodel is only called after the first (integer) solution has been found by the solver. (default 0ms). The model example learn10[2] and also model tsp-6[3] – using lazy constraints in Gurobi show how to use this feature.

23. **SIP23**: A solver dependent parameter: Number of times LPL tries to ask the solver for a free licence. LPL checks the solver licence. If SIP23 is larger then 0 and if a free solver licence is not found, LPL rests for a certain time specified in SIP24 and tries again to obtain a licence. This is repeated as many times as is specified in SIP23. (default is 0).

24. **SIP24**: A solver dependent parameter: Time between two requests in millisecs to ask the solver for a free licence. (default is 0ms)

All SIP parameters (except parameter 8 and 9) may contain also the following strings:

[2]https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn10
[3]https://lpl.matmod.ch/lpl/Solver.jsp?name=/tsp-6

`%1`	will be substituted by the model file name
`%2`	will be substituted by the objective function name
`%3`	will be substituted by SIP7 or SIP8, depending whether the model is to be maximized or minimized
`%4`	will be substituted by SIP19, if the problem is a MIP-problem.
`%e`	will be substituted by the directory of the executable
`%%`	will be substituted by the Solver Options string (SO) defined as second parameter in the function `SetSolver()`.

Any other characters or strings are taken literally. (Note that a backslash initialize a non-printable character). The parameters SIP9 to SIP17 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 are 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).

9.2.2 The Solver Options (SO)

Solver Options are solver specific parameters that are defined within LPL and communicated to the solver. The default solver options are defined in the solver interface parameter SIP6, these options are always pass to the solver. Additional solver options can also be defined as second argument in the `SetSolver` call. These options override the options defined in SIP6 eventually.

All solver options for a particular solver are specified in the Solver Option Mapping File (SOMF). An example is the file `cplex.prm` used for the Cplex's dynamic link library. As an example: the file contains the following line (defining the global time limit):

```
TILIM 1039 d Global time limit
```

The solver option instruction to limit the time to 60secs would be `TILIM=60`. LPL now makes a lookup in the SOMF file to translate this instruction into a library call of cplex – in this case `CPXsetdblparam(env,1039,60)` , see cplex manual for details.

Each line in the SOMF file is separated into four parts from left to right: (1) the name as used in LPL's SIP6 (option solver string) (`TILIM`), (2) the substitute (here: `1039`), (3) an attribute (here: `d`) (this is used in the LPL-CPLEX interface to call the right routine: possible values are `i` (for integer values), `d` for double, and `s` for string values) (4) the rest of the line, which

is a comment. The four parts must be separated by at least one blank. The first part is not case-sensitive.

The SOMF file for the Gurobi solver is *gurobi.prm*. For Gurobi all parameters (see Gurobi documentation) are defined in this file. For example, to set the time limit to 100secs, the MIPFocus to 1, and the Seed to 2, one would add the following statement to the LPL code (`gurobiLSol` is defined in the file *lplcfg.lpl*) :

```
SetSolver(gurobiLSol,'TimeLimit=100;MIPFocus=1;Seed=2');
```

Four additional solver options are specific to the LPL-Gurobi interface:

1. Callback within solver: Adding 'CALLBACK=X' means that an LPL (sub)model called **X** will be run from within the solver's callback at repeated time interval in millisecs (specified by parameter SIP22).
2. Start MIP solution: Adding 'MIPSTART=1' means to pass a starting solution to the solver. That is, all variable values are passed to the solver as a starting value (note that this value is the default value if not explicitly assigned). Depending on the Priority Attribute (see 4.1.8), this can also be used to pass only a partial solution or to pass hint values (see Gurobi documentation for hints).
3. Start Compute Server: Adding 'SERVER=<Servers>' means to call a Gurobi Compute Server where <Servers> is a comma separated list of Servers URL addresses in the Internet.
4. Default log output: Adding 'CBMESSAGE' means that the callback log output of gurobi is the same as with `gurobi_cl`.

9.2.3 Communication between LPL and Solvers

When LPL runs a *solve statement*, the following process is executed:

1. All constraints are generated (a model instance is created) and the problem type 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 (or the default values).

5. The solver options SIP6 are processed, that is, the parameters that are passed to the solver or written to a file that the solver then calls.
   (a) all substrings `%0` to `%3` within SIP6 are substituted as explained above
   (b) SIP6 is merged with the solver option string define as the second parameter (SOS) of the function SetSolver(). If the SIP6 contains the substring `%%`, it is substituted by SOS, otherwise the SOS is appended to the SIP6.
   (c) SIP4 (if not empty) is interpreted as a filename (the solver options mapping file (SOMF)) and all solver options in the SIP6 are substituted by a corresponding entry in the SOMF file,
   (d) SIP5 (if not empty) is interpreted as a filename (solver parameter file (SPF)) and the modified SIP6 string is written to this file.
6. The solver is called as specified by the SIP3 (LPL is waiting)
7. The SIP2 is processed after the solver returns.

Example 1: Suppose a model contains the following function call:

```
SetSolver(cplex,'timelimit 60');
```

The solver `cplex` must be defined before (typically in the file `lplcfg.lpl`) as following:

```
string parameter cplex := ',,lib:cplex125,cplex.prm';
```

Note that all SIP parameters 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 (in Windows) the DLL library of `cplex125.dll` as solver in step 6 above (in Linux `libcplex125.so`). SIP4 is needed to substitute the solver options (see above the SOMF) to work with the library. Note also that the function `SetSolver()` can be called with two parameters, where the second parameter is a string containing the solver options.

Example 2: Suppose a model contains the following statement:

quad `SetSolver(mopsSol, 'XMXLPT=1;XMXMIN=1');`

The solver `MOPS` (www.mops.fu-berlin.de) must have been defined before (typically in the file `lplcfg.lpl`) as following:

```
string parameter mopsSol := 'm,s,mops.exe,,XMOPS.PRO,\
     XFNMPS=\'%1.mps\';XMINMX=%3;XFNLPS=\'%1.sol\' ;\
     XFNIPS=\'%1.sol\' ;XOUTLV=3,\'max\',\'min\'\
     ,%1.sol,12,25,15,%1.sol,89,solu,infeas,unbound,LS:iLS:LP:MIP';
```

SIP1 is `'m'`, 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 following string:

```
XFNMPS='xyz.mps';XMINMX=max;XFNLPS='xyz.sol';     XFNIPS='xyz.sol';XOUTLV=3;
```

Next it is merged with the SOS string':

```
XMXLPT=1;XMXMIN=1
```

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 `'s'`. Therefore, the file specified in SIP10 is read as a SOL-file and the file specified in SIP13 is read as a DUA-file.

9.2.4 LPL's LP Solver

LPL comes with an internal LP solver. This solver is for small simple LP models. It cannot solve integer problems and should not be used for large problems. The solver is called as follows

```
SetSolver(lpl-lp);
```

It does not allow to pass any parameters.

9.2.5 The Heuristic Solver

LPL comes also with an integrated heuristic solver for certain problems, called permutation problems (PERM). A permutation problem is defined as following: Let π be the set of all permutations of a vector of the n numbers in the range $[1 \ldots n]$. The objective is to find a permutation $\pi_i \in \pi$ over all $n!$ permutations which minimizes a certain function f:

$$\min_{i \in \pi} f(\pi_i)$$

This problem has many applications. Four solvers are directly integrated: (1) a heuristic based on a simple tabu search method, (2) a heuristic based on local search; (3) a random search solver; and (4) a full enumeration solver. The purpose of this solver was to show the feasibility to integrate an arbitrary solver into the LPL system. The solver is not particularly powerful. The tabu search solver is useful in finding good solution to many small problems. The local search solver looks in the neighborhood repeatedly until it finds a local minimum and then stops. 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. The solvers are invoked by one of the following lines:

```
SetSolver(tabuSol);
SetSolver(randSol;
SetSolver(locaSol);
SetSolver(enumSol);
```

These solvers 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 solvers. 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 semicolons are passed to the heuristic solver. The nine parameters are as follows:

- 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(n2) 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 when an improved solution has been found (0=to be mute, 1=yes write it).

9.3. Directories and File Paths in LPL

The paths to all directories used in LPL are collected in a *directory list*. All files are searched in this directory list. The paths in the directory list are separated by a semicolon (Windows) or a colon (Linux). The directory list consists of the following paths (in this order):

1. The working directory or the current directory (the main model).
2. The directory of the executable (the lpl executable), called EXEDir.
3. The APL path-list defined as @PATH=
4. The *LPLPATH* directory list (system variable).
5. The user-defined directory list.
6. Recursively in all subdirectories of the working directory.

The directory list is displayed in the <LPL : Options> window of *lplw.exe* or at the beginning of a compilation when the compiler switch `'ww'` is used, see 9.7.

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 function

```
SetWorkingDir('<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* function specified by their own path).

The *directory of the executable* (EXEDir) is the directory where the executable (*lplw.exe* or *lplc.exe* or the application that calls the *lpl.dll*) is located. If LPL is called as as library, then the executable that calls the library defines the directory of the executable.

The *APL-path* is the path-list defined by the APL parameter @PATH (see below APL)

The *lplpath* directories is set by the environment variable named: *'LPLPATH'*.

The user-defined directories are specified by the function `SetPaths`. Example;

```
SetPaths('c:/lpl;c:/lpl/models;c:/solver/cplex');
```

If a file to be read is not found in any path before then LPL looks for the file in all subdirectories of the working directory recursively.

9.4. The File `lpl.ini`

In the file `lpl.ini` various initialization parameters can be specified. This file – if present in the EXEDir – is read at the very beginning even before console parameters are assigned. Four parameters can be assigned in this file, each defined on a separate line in the format `parametername=value`:

```
maxa=integer     //preset memory allocation for numerical data
maxt=integer     //preset memory allocation for alphanumerical data
maxa=integer     //preset memory allocation for constraints
copt=string      //preset compiler options
```

Example:

```
copt=wwvv
maxa=2000000
```

Note that, there is no need for memory allocation in LPL. LPL reallocate memory on the fly. However, for very large models it might be advantageous to do so.

9.5. The File `lplcfg.lpl`

If the file `lplcfg.lpl` exists in the LPL directory list, then this file is compiled before the model file. It is treated as an include-file after the first 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.

9.6. The File `lpl.file.policy`

If this file is present in the EXEDir directory then one can restrict reading and writing files from all directories except the working dir, its subdirectories, the EXEDir, and the once that are explicitly mentioned in its list. This is important basically in a Server. Within the file each allowed directory must be written on a separate line.

9.7. 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`. The second parameter – the *compiler switches* – is an optional string to instruct LPL how to compile, and the third (APL) is explained in the next section).

The second parameter (*compiler switches*) can be empty or can contain any characters. The following characters, however, modify the default behavior of the compiler:

0	No model transformation is done (for LocalSolver and others).
a	The NOM-files are never overwritten, but they are appended.
b	Enforce index binding
B	Check active and passive IndexLists (are they equal?)
c	The solver is not called.
d	Debugging information is written to the BUG-file.
e	Generates the EQU-file (element names separated by comma)
e0	Generates the EQU-file (element names separated by comma)
e1	Generates the EQU-file (string names separated by comma)
e2	Generates the EQU-file (element names separated by _)
e3	Generates the EQU-file (string names separated by _)
e4	Generates the EQU-file (element names separated by tabs)
e5	Generates the EQU-file (string names separated by tabs)
e6	Generates the EQU-file (element names with set name separated by comma)
e7	Generates the EQU-file (numbered var and const names)
e8	Generates the EQU-file (numbered var and const names - Latex)
ee?	Generates the EQU-file (sparse: only variables with value<>0, ? is empty or 0 to 7 as before)
E?	Generates the LIS-file (solution listing file, ? like for 'e')
f	Generates the ENG-file (generate the doc as an external file)
F	Generates the LPY-file (regenerate the source LPL code)
h	No write statement is executed
i	Generates the INT-file
k	set elements added only once (undocumented)
l	Generates the LPO-file
L	Generates the IN1-file (same as LPO-file, but readable chars)
m	Generates the MPS-file. Note that also a MPS file from the solver is generated (file name is $$$$-solver.mps)
o	Generates the SYM-file
O	Generates the STO-file
p	Generates slack variables (using *Sl* function) for all constraints, Adds a new minimizing function, minimize all slacks.
q	Generates a SQL-script and a (LPL) model file.
r	Optimize index lists only with 'opt' nomen attribute
R	Optimize all index lists
s	Only parse the model, but do not run it
ss	Only parse, strips off /*...*/ comments, stores a new file
sss	Only parse, strips off all comments, stores a new file
ssS	Only parse, strips off all comments, stores a new file, encrypt
t	Generates the TEX-file (a partial LATEX-file, as \section
t1	Generates the TEX-file (a partial LATEX-file, as \subsection
t2	Generates the TEX-file (a complete LATEX-file of the \MOproblem documentation part only)
t3	Generates the TEX-file (a complete LATEX-file)
u	The constraints are not generated and a solver is not called
v	Write output to files lplStat.txt and lplStat1.txt.
vv	Write output also to file lpllog.txt.
w	Generates more output during compilation.
ww	Generates even more output during compilation.
ww1	Additional output about the memory usage and more, in addition write debug files bugPalist.txt and bugequ.txt
W	Output generated by the Internet LPLserver
x	The configuration file lplcfg.lpl is ignored while parsing.
y*	IIS-set is generated when model is infeasible. '*' is empty or a number from 0 to 6 and outputs the name in the same way as the switches 'e' above. 'y6' also adds the comment of the constraint.
z	Compile all LPL models in a given directory (only for lplc.exe)
-1 to -8	Level of translation of logical constraints into 0-1 constraints (undocumented).
<else>	any other character does not modify the default behavior.

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 compiler switch p is interesting when a model is infeasible. Running it with this switch will reveal the infeasibilities eventually.

9.8. The Assigned Parameter List (APL)

The Assigned Parameter List (APL) is a list of parameters separated by a '#' character. Each parameter must have the format :

```
ID=value
```

where ID can be any non-indexed parameter defined within a LPL model or it can be one of the following strings: @IN, @INF, @OUT, @OUTF, @RAN, @DOC, @PATH, @ID, @IP, @IP1, @SOL. The second part value must by a value assigned to the corresponding parameters. An example of a APL is (RAN gets the value 2, PATH gets the value c:\modeling, and the model parameter seed gets the value 3) :

```
@RAN=2#@PATH='c:\modeling'#seed=3
```

The APL can be used as a third parameter when calling **lplw**, bf lplc, etc. In this case, the '#' character can also be replaced by a space. When using the lpl library, the APL must be assigned using the function **LPLsetParamS(5,APL)**.

- 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 the following command:

  ```
  lplc xx.lpl - aa=10
  ```

  then the parameter within the model gets the value 10 after a parse of the model xx.lpl. Note that if the parameter is assigned within the

model already, then it *must be assigned as a table, like (since a table is assigned during a parsing)*

```
parameter aa:= [4];   -- with value assigned
```

otherwise the APL parameter will be overwritten. Several parameters can be assigned in this way.

- *If* `ID` *is* **@IN** *then* `value` *must be a model name identifier. In this case, a model call is automatically added before the first executable statement, otherwise if the model with name* **data** *exists then a model call to this model is added (see model learn53[4]).*

- *If* `ID` *is* **@OUT** *then* `value` *must be a model name identifier. In this case, a model call is automatically added as a last statement in the main model, otherwise if the model with name* **output** *exists then a model call to this model is added.*

- *If* `ID` *is* **@INF** *then* `value` *must be a filename. The content of the file must be in LPL syntax and the main model name in that file must be* **data**. *In this case, The (data) model is included into the main model. (see model wl2[5]).*

- *If* `ID` *is* **@OUTF** *then* `value` *must be a filename. The content of the file must be in LPL syntax and the main model name in that file must be* **output**. *In this case, The (output) model is included into the main model. (see model wl2[6]).*

- *If* `ID` *is* **@RAN** *(RandomSeed) sets the random seed of an LPL run. For example the call to:*

```
lplc xx.lpl - @RAN=23
```

- *If* `ID` *is* **@DOC** *then* `value` *must be a filename. The file must contain the documentation part of the model. 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. If the filename contains a '*' then it will be replaced by the model filename.*

- *If* `ID` *is* **@PATH** *then* `value` *must be a string that denotes a path-list that is added to the global LPL directory list.*

[4] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn53
[5] https://lpl.matmod.ch/lpl/Solver.jsp?name=/wl2
[6] https://lpl.matmod.ch/lpl/Solver.jsp?name=/wl2

- *If* `ID` *is* **@ID** *then* `value` *is the clientID of the LPL executable.*
- *If* `ID` *is* **@IP** *then* `value` *is the remote host IP of the LPL executable.*
- *If* `ID` *is* **@IP1** *then* `value` *is the remote host IP of the LPL executable.*
- *If* `ID` *is* **@SOL** *then* `value` *is an integer specifying a predefined solver. They must be predefined in the lplcfg.lpl (can change!):*

  1 is 'gurobiLSol', 2 is 'cplexLSol', 3 is 'xpressLSol', 4 is 'cbcSol', 5 is 'glpkSol', 6 is 'scipSol', 7 is 'mipclSol', 8 is 'knitroLSol', 9 is 'conopt'. Note that this is basically interesting in a batch job when testing several solvers. Normally, the solver is chosen on the basis of a `SetSolverList()` *instruction in the lplcfg.lpl file.*

9.9. Model Documentation

The compiler switches `t0-t4` will generate a documentation file in LaTeX (called TEX-file): `t0-t1` and `t4` generate a partial LaTeX file that can be included into another LaTeX document using TeX's `\include{}` command. The option `t4` generates a subsubsection for the model. The option `t0` generates a subsection for the model, and the option `t1` generates a section for the model. The options `t2-t3` generates a standalone LaTeX file that can be translated into a PDF or HTML document using the MikTeX software (PDF can be translated into HTML using the free pdf2htmlEX software). The model documentation in the LPL source code is part of the model source code. It consists of comment attributes and documentation comments:

- *Comment attributes* are strings within quotes (`"..."`) and have already been explained in chapter 4.
- *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. 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 LaTeX typesetting system. The documentation comment in LPL may also contain one the following `@`-specifications (note the char `@` can be escaped by `\@`):

1. Text within two `@`, as in `@Here is text@`, for example : The part "Here is text" will be typeset as verbatim LaTeX code.
2. `@.` (`@` and a dot , beginning on a new line): This translates into an item of an enumerated list. The list ends with an empty line.
3. `@!` (`@` and a exclamation mark) adds the formal model in the listings package mode.
4. `@?` (`@` and a question mark) adds code in a listings package mode.
5. `@@` (two `@`'s beginning on a new line): Includes the whole formal model in the listings package. If `@@` is followed by the word or the first letter of "main", "data", or "output" then only the corresponding submodel is included.
6. `@+modelname` (`@` and a plus followed by a modelname, beginning on a new line): adds the model with the name `modelname` in mixed Math-Mode and text mode. This is very new and powerful for larger models.
7. `@$` (`@` and a $): adds the whole model in Math-Mode.
8. `@%` : adds the whole model in Math-Mode (a different format).
9. `@|` : adds LaTeXcode in Julia programming language.

All `@` specifications could also be realized by using the appropriate LaTeX commands.

The translation tool from LPL to a LaTeX code also uses a list of predefined TeX or LaTeX definitions, which are defined in the file called `lpl.tex`.

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 (Example see learn08[7].
3. Generate the TEX-file using lpl's command switch `t0-t4`, example:

   ```
   lplc mymodel t3
   ```

4. Translate the resulting TEX-file using `pdflatex` in the Miktex distribution (or use `modeldoc.bat` batch job in the LPL distribution).
5. The result is a PDF-file, that is called mymodel.pdf in our case

The batch job `modeldoc.bat` batch job can automatically be called from the `lplw.exe` program (menu Tools/Create ModelDoc (PDF). Verify all paths given in this batch job before running.

The whole documentation can also written to or read from a separate file. The file can be read automatically after a parse by the `@DOC` APL-parameter, as explained. (The compiler switch `f` writes the file.) The structure of the documentation file is as follows:

- 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:

  ```
  ##a Comment to this parameter
  ```

- 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.
- 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.

Each comment in the file can contain multiple line the comment ends when an `##` , an `#&` , and `#!` or an eof occurs. The last newline characters are removed from the comment content.

[7] https://lpl.matmod.ch/lpl/Solver.jsp?name=/learn08

9.10. 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 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 password. Each file in a model project can be encrypted separately and with a different password and can only be decrypted with the same password again. There is also a procedure in the lpl.dll that does the job.

9.11. Draw Library

The *Draw* library allows one to draw using LPL functions. The result is stored in a SVG-file Each function is explained here. Here, the context of the library is clarified.

Any of the functions initializes a new drawing space. The size, the viewport and scaling of the drawing space can be controlled by the function `Scale`. If the function is not used, the size of the drawing space is automatically derived. If it is used it *must* be the first function call of the library. Furthermore, it can be called only once (additional calls have no effect). (Well, it can be used again after the graphic has been written to a file using *Draw.Save*). The function *Scale* has two forms:

- Scale(zx,zy): the size of the graphic is automatically derived, with the difference that all coordinates are scaled by the factor (zx,zy) – in the x-axis and y-axis directions. zx can only be positive, while zy can also be negative which turns the whole graphic upside-down. If zx and zy are smaller than 1, then the graphic is reduced otherwise it is expanded (except the font-sizes, they are not scaled).
- Scale(zx,zy,x,y,w,h): The two parameters zx and zy have the same meaning as in the first call, the additional four parameters x, y, w, and h define the whole size and the viewport of the graphic. The size of the graphic is (w,h), and (x,y) are the top/left coordinate of the graphic. Note that these four parameters are also multiplied by zx and zy.

A call to *Draw.Save* terminates a graphic instruction list and writes the

drawing instructions so far to a file. The extension of the filename is automatically set to *svg*. If no Save instruction is used in the LPL code, then LPL automatically calls it at the end of a run to write a svg-file with name of the model. After the call to *Draw.Save* a fresh draw instruction list can begin to generate another graphic.

Most functions have mandatory and optional parameters specified in the parentheses [...]. Numerical parameters with value -2 are interpreted as "not-used", that is, a missing optional parameter value is the same as -2. All functions fall into two groups: *definition-functions* and *drawing-functions*. The function-name of a definition-function begins with *'Def'*. Definitions-functions define various drawing attributes, such as the font size, opacity, a color gradient, or a transformation type, a gradient or a filter. These attributes can then be referenced by the drawing-functions – using their *id* name. For example, one can define a font 'Verdana,sans-serif' with a font size of 50px as follows:

```
Draw.DefFont('myid','Verdana,sans-serif',50);
```

In the function that draws a text, this definition can be referenced by the name `myid` as follows (the starting # is mandatory for a referenced name):

```
Draw.Text('\#myid','My text to be written',50,50);
```

As another example, the following rectangle will be rotated by 30 degrees:

```
Draw.DefTrans('ro', 'rotate',30);  //defines a rotation of $30^{\circ}$
Draw.Rect('\#ro',100,100);
```

One can also cumulate the references in a drawing function. For example, if a text with the previous font `myid` must be drawn rotated as defined in `ro` definition then it is possible to use *both* names as follows (the two names `myid` and `ro` must be separated by a blank character):

```
Draw.Text('\#myid ro', 'My text to be written',50,50);
```

The six functions are definition-functions (see Drawing Functions):

`Draw.DefFill`	defines fill attributes
`Draw.DefFilter`	defines filter attributes
`Draw.DefFont`	defines font attributes
`Draw.DefGrad`	defines gradient attributes
`Draw.DefLine`	defines line attributes
`Draw.DefTrans`	defines a transformation

All the other functions in the *Draw* library are *drawing functions* and they are placed in the svg-file in the order of their appearances in the LPL code.

In many functions the two parameters *c1* and *c2* are used. They are color numbers: *c1* is the fill color ($c1 = -1$ means transparent), *c2* is the stroke color. They can also be assigned by the function *Rgb(r,g,b)*, which generates a color in the Rgb (red,green,blue) color space (with $255 \geq r, g, b \geq 0$), *Rgb(0,0,0)* means *black* and *Rgb(255,255,255)* means *white* (saturating all three colors), and *Rgb(255,0,0)* means *red* (red only is saturated). Another way to assign the color in LPL is by a positive number or an expression that result in a positive number. (The only negative number that can be assigned to a color is -1 which means transparent, as already mentioned.) The (positive) color numbers are assigned as follows:

Special colors (see below)	0 to 31
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 in LPL for a list of saturated colors. Model examples are xDrawColors[8].

As already mentioned, LPL generates a svg-file. This file can be translated to a raster format using various open-source programs, such as *Inkscape*. In the distribution of LPL, the user finds two batch script (*svg2pdf.bat* and *svg2png.bat*) that generates a pdf-file and a png-file. To run these scripts *Inkscape* must be installed.

9.12. The Tool compareEQU.exe

The program compares two EQU-files generated by LPL. It is easy to call: just add the two file names as parameters in the console. The program checks whether the two files are equal, if not, it generates two additional files with the same names and extension '.diff' that contains the differences. The program recognizes if the constraints are in a different permutation. It even recognizes the different permutations within the index-lists (of constraints and variables), if the EQU-files have been generated with the compiler switch 'e6'. The EQZU-files can be very large and the program is very efficient. Call

[8] https://lpl.matmod.ch/lpl/Solver.jsp?name=/xDrawColors

```
console> compareEQU file1.equ file2.equ
```

9.13. Undocumented Features

Undocumented features are extension of LPL on an experimental basis. That is, they can be removed or modified at any time.

1. The `$` operator is 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.

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. In an IndexList `{i,j}` one can also use an indexed *parameter* name. If the parameter is sparse then the index is running through the sparsity of the parameter.

Chapter 10

Runtime Library of LPL

The LPL package comes with a 32-bit DLL and 64-bit (Dynamic Link Library) – called *lpl.dll* (*lplj.dll* for Java). These DLLs allows the user to integrate the *complete* LPL functionality into an application written in another language, for example C++, Java, etc. 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 and to run and compile models.

10.1. Exported Functions

The dynamic link library of LPL exports the following functions (note that the parameter convention is **stdcall**) :

```
procedure LPLsetLicense(s:pChar);
procedure LPLsetCallbacks(c:TProc; w:TProcC);
procedure LPLinitParam();
procedure LPLinit(Fn:pChar; maxt,maxa,maxr:integer);
function  LPLcompile():integer;
function  LPLcompileWithCallbacks(c:TProc; w:TProcC):integer;
procedure LPLfree;
function  LPLwhere:integer;
procedure LPLgetErrMessage(n:integer; var msg:pChar);
function  LPLgetError:integer;
procedure LPLsetError(n:integer);
function  LPLgetParam(attr:integer):double;
procedure LPLgetParamS(which:integer; var sP:pChar);
```

```
procedure LPLsetParamS(which:integer; sP:pChar);
function  LPLsolve():integer;
procedure LPLsaveSnapshot(sP1:pChar);
procedure LPLloadSnapshot(sP1:pChar; add:integer);
procedure LPLgenACCESSdb;

procedure LPLsetFocus(name:pChar);
function  LPLgetGenus:integer;
procedure LPLgetAttr(attr:integer; var sP: pChar);
function  LPLnextFocus(genus:integer):integer;
function  LPLnextPosition(i:integer):integer;
procedure LPLgetName(opt:integer; var sP:pChar);
function  LPLgetValue(attr:integer):double;
procedure LPLgetValueS(attr:integer; var sP:pChar);

procedure LPLPivotSetParam(s:pChar);
function  LPLPivotGetParam(which:integer):integer;
procedure LPLPivotInit(var w,h:integer);
function  LPLPivotX:integer;
function  LPLPivotY:integer;
function  LPLPivotData:pChar;
procedure LPLsetSelect(n,what:integer);
procedure LPLEncryptFile(FileName,Key:pChar; encr:integer);
```

The following types are defined as:

`integer`	4 bytes
`double`	8 bytes float point
`pChar`	pointer to a null-terminated string
`TProc`	pointer to a parameterless procedure (4 bytes)
`TProcW`	pointer to `procedure(s:pChar); stdcall;` (4 bytes)
`char`	2 bytes char (Unicode)
`var pChar`	means also a null terminated string, the caller must reserve enough space to receive the string.

Certain procedures must be called in a specific order. Call the `LPLsetLicense` function first to set the license and the `LPLsetCallbacks` to specify the callback functions. Then `LPLinit` must be called to allocate the memory and to initialize all variables. Next one may call `LPLcompile` or `LPLsolve`. After these calls all model and solution information can be queried used various functions. 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 `LPLgetErrorMessage` 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 procedures (p.e. `LPLgetParamS`) return 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 must allocate space before the call. The procedure then fills the allocated memory with a zero-terminated string. It is in the client's responsibility to allocate enough space. The procedures are now explained in more details.

LPLsetLicense allows one to include a license key to the library. The parameter s is the license key (the string that is in the `lpl.lic` file). Alternatively, the license file `lpl.lic` must be present and found by the application.

LPLsetCallbacks allows the user to 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 his own procedure here to allow him 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 empty, `w` or `ww`. All messages are handled by the callback procedure `w:TprocC` (second argument of LPLsetCallbacks). The user can write his own write procedure and redirect these messages. Example: Suppose the user writes the two procedures and uses them as follows:

```
procedure MyWriteToLog(s:pChar); stdcall;
begin writeln(s); end;
procedure MyCallback; stcall; begin end; // do nothing

  ...
   LPLsetCallbacks(MyCallback,MyWriteToLog);
  ...
```

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`) which uses this same mechanism. 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 – that is the filename where the LPL-model is stored. It will overwrite a previously assigned modelname. An 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. Before calling, one can call LPLsetParamS(2,s) to set the compiler switches specified in the parameter s. If LPLcompile succeeds, its return a value of 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 LPLgetErrorMessage. This number reflects an error message in the text file `lplmsg.txt`. The message file must be present in the EXEDir 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	An error state (after an error occurred)
0	At startup and before 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 a 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 LPLsetError is useful in the callBack function (parameter c in the procedure LPLsetCallbacks). A typical callBack function for LPL is (it is used in the program `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...';
    10..19: Form.Label2.Caption:='Multi-Snapshot Analysis';
  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.

While compiling/running the model (using LPLcompile) this function 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.

LPLgetParam (corresponds to the functions (GetParam) returns a global data from LPL as double. The function returns the following value for the parameter attr :

1	elapsed time of a model run in msecs
2	elapsed solution time in msecs
3	problem type [0..11]
4	solver status is returned [0..7]
5	optimal value
6	best lower bound of a MIP solution
7	best upper bound of a MIP solution
8	memory allocation for numerical data (maxa)
9	memory allocation for alpha-numerical data (maxt)
10	memory allocation for alpha-numerical data (maxr)
11	number of variables
12	number of binary variables
13	number of integer variables
14	number of constraints
15	number of variables + constraints (11+14)

LPLgetParamS returns the same information as the function GetParamS(which) different information pieces from the LPL kernel depending on the first parameter which. The returned information is stored in the second parameter sP.

LPLsetParamS writes information in the same way as LPLgetParamS 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
5	the assigned parameter list (see : APL)
6	the File search path is extended by sP
7	the working directory
13	parameter guiding the generation of a Database (0:all entities, 1:data (parameters and sets only), 2,3: variable only.)
otherwise	do nothing

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.

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 extension sps.

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 following procedures work with a **focus**, that is, an internal pointer to a given entity of the model. This pointer can be set by the procedure LPLsetFocus. If not set, the focus points to the beginning of a model. We can set the focus to the beginning calling *LPLsetFocus(");* – that is calling LPLsetFocus with an empty string parameter. Note that the function works in the same way as LPL's function SetFocus.

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.

The argument *name* can be in dot-notation. Hence, if a name (say 'abc') is defined in a submodel (say 'mySub'), then the function must be called as follows: *LPLsetFocus('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.

LPLnextFocus jumps to the next entity of the corresponding genus. The function returns 1 as long as the next entity can be focused otherwise it returns 0. The parameter *genus* has the following meaning.

-1	any (all) genus
0	set declaration (base set)
1	set declaration (compound, indexed)
2	parameter declaration (numerical parameter only)
3	real variable declaration
4	binary variable declaration
5	integer variable declaration
6	constraint declaration
7	maximize, minimize statement
8	model declaration
10	if-(else) statement
11	while statement
12	for statement
13	Expression statement
14	model call
15	Assignment statement
16	else (part of if)
17	a variable declaration (real, binary or integer)
18	parameter declaration (string parameter only)
19	slack variables (generated by Sl())
21	end (while,if,for)
22	end (model)
23	else (others)

As an example let us run through the whole model an collect all set names. The (pseudo)code would be as following:

```
Declare : ThisSetName AS String
LPLsetFocus('')                //sets the focus to the beginning
while LPLnextFocus(0)=1 do     // jump to the next set entity and sets the focus
   ThisSetName = LPLgetAttr(4) //return the set's name
Endwhile
```

LPLgetAttr corresponds to the function GetAttr, see GetAttr(r,attr). It 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 was defined at GetAttr(r,attr).

LPLgetGenus returns the genus of the focused entity. The genus return

value is given in the following list:

-1	any (all) genus
0	set declaration (base set)
1	set declaration (compound, indexed)
2	parameter declaration (numerical parameter only)
3	real variable declaration
4	binary variable declaration
5	integer variable declaration
6	constraint declaration
7	maximize, minimize statement
8	model declaration
10	if-(else) statement
11	while statement
12	for statement
13	Expression statement
14	model call
15	Assignment statement
16	else (part of if)
17	a variable declaration (real, binary or integer)
18	parameter declaration (string parameter only)
21	end (while,if,for)
22	end (model)
23	else (others)

LPLnextPosition returns 1 or 0 from the focused entity, that is, whether the next position within the datacube exists or not. (see example below). The parameter i can be 0 or 1, 0 means sparce output (zero values are ignored), 1 means full output (see also NextPosition function).

LPLgetName corresponds to the function GetName, see GetName(r,opt). It returns a string from the focused entity that is the instance name as a string. (see example below). The parameter opt is defined at GetName(r,a).

LPLgetValue corresponds to the function GetValue, see GetValue(r,attr). It returns a value from the focused entity. The parameter attr are defined at GetValue(r,a).

LPLgetValueS corresponds to the function GetValueS, see GetValueS(r,attr). It returns a string for the focused entity. The parameter attr is defined at GetValueS(r,a).

The following example runs through all variables and returns the names the values and the duals. The (pseudo)code would be as following:

```
Declare : name AS String, value and dual AS double
LPLsetFocus('');
while LPLnextFocus(3)=1 do begin
```

```
  while LPLnextPosition()=1 then begin
    LPLgetName(0,name);
    value = LPLgetValue(0);
    dual  = LPLgetValue(3);
    //// --- do something with the data ---
  EndWhile
EndWhile
```

LPLPivotSetParam sets a parameter for the pivot table generation. These parameters should be set before the LPLPivotInit function is called. If none is set, then default parameters are used. The parameter s specifies which of the pivot-table parameters has to be set. One can call the function several times to set various pivot-table parameters or one can call it once where the parameters are concatenated and separated by a semicolon. A pivot-table parameter is a string as follows:

```
Expa=x   x= 0,1 (table expanded?), default:  'Expa=1'
Head=x   x= 0,1 (head printed?), default:  'Head=1'
Vind=x   x= 0,1 (vert indexes printed?), default:  'Vind=1'
Hind=x   x= 0,1 (hori indexes printed?), default:  'Hind=1'
Isp=x    x= 0,1 (index name is sparse), default:  'Isp=1'
Dsp=x    x= 0,1 (datatable sparse), default:  'Dsp=1'
Inam=x   x= 0,1 (alias index names), default:  'INam=0'
Agg=x    x= 0,1,2,3 (aggregates, none,sum,count,avg) , default:
         'Agg=0'
Attr=x   x= 0..8 (corresponds to the number in LPLgetValue),
         default:  'Attr=0'
X=x      x>=0 (left position) , default:  'X=0'
Y=x      x>=0 (top position) , default:  'Y=0'
H=x      x>=0 (hori indexes) , default:  'H=1'
K=x      x>=0 (sliced out indexes) , default:  'K=0'
Perm=x   x is a permutation, default 'Perm=1,2,3,4,..'
FNa=x    a= 1..6, x= Fontname, default:  'FNa=Tahoma'
FSa=x    a= 1..6, x>0 Fontsize, default:  'FSa=8'
FBa=x    a= 1..6, x>0 Fontstyle, default:  'FBa=0'
FCa=x    a= 1..6, x>0 Fontcolor, default:  'FCa=..'
BCa=x    a= 1..6, x>0 Backcolor, default:  'BCa=..'
```

LPLPivotGetParam returns the value of pivot-table internal parameter. The function parameter which specifies which parameter to be returned:

0 the number of indexes (n)
1 the total number of (non-empty) pivot elements (nn)
2 the total number of rows of the pivot table (Rows)
3 the total number of columns of the pivot table (Cols)
4 the number of columns used for indexes (v)
5 the number of rows used for indexes (h)
6 the number of project on indexes (k)

LPLPivotInit create an internal data structure for a focused entity. It must be called before the next three functions. It returns the size of the pivot table (w,h)

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(w,h);
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, un-select or inverse the selection. By default all elements are selected.

LPLEncryptFile encrypts or decrypts the source code of a model file. The first parameter is the filename, the second parameter is a key (password). If a file is encrypted the same key must be used to decrypt it again. The last parameter is an integer: 0 (for decrypting) and 1 (for encrypting).

10.2. Using the Library

The library can be used in every programming environment that allows one to load true 32-dll (or a 64-bit dll library. Several examples are given.

10.2.1 Using the LPL Library from Delphi

The three executables `lplc.exe`, `lpls.exe` and `lplw.exe` can 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(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():integer; stdcall; external dllLpl;
  procedure LPLfree(); stdcall; external dllLpl;

  procedure MyCallBack; begin {write('.');} end;
  procedure MyWrite(s:pChar); stdcall; begin writeln(s); end;

begin
  LPLsetCallbacks(MyCallBack,MyWrite);
  LPLinitParam();
  LPLinit('alloy.lpl',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.

10.2.2 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" () 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 LPLgetParamS Lib "lpl" (ByVal n As Long, s As String)
```

```
Private Declare Sub LPLsetParamS Lib "lpl" (ByVal n As Long, s As String)

Public AbortBottonClick As Boolean 'this is a boolean set to true by a button'click (click

Private Sub MyWriteLog(s As String)  'LPL WriteToLog callback
  ' Note that s cannot be accessed in VB, so it must be read through LPLgetParamS 10
  Dim s1 As String
  Dim p As Long
  s1 = String(256, " ")
  LPLgetParamS 10, 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
  LPLsetParamS(2, "wvv")
  lRet = LPLcompile()
  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.

10.2.3 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)(void);
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();
      // Compile LPL model with specified options.
if (nRes != 0) {
// Error compiling model.
LPLfree();
FreeLibrary(hDll);
return 3;
}
```

```
LPLfree();  // Finalize LPL.

FreeLibrary(hDll);
return 0;
}
//---------------------------------------------------------------------------
```

10.2.4 Using the LPL Library from Java

A java program must use the library *lplj.dll* (instead of *lpl.dll*). Both libraries are identical 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 and open the folder.
2. Copy *lplj.dll* and *alloy.lpl* into that folder also (if they are not there already). Also you need the licence file *lpl.lic*.
3. Run the batch job *compile.bat* or *compile64.bat*.
4. Run the batch job *run.bat* or *run64.bat*.

10.2.5 Using LPL in Jupyter Notebook

It is easy to run LPL code from a Python Jupyter Notebook. A small notebook has been prepared, the user can download it at LPLS.ipynb. A additional Jupyter Notebook for the Julia kernel is available at: LPLSJ.ipynb

Chapter 11

Appendix A: LPL Syntax

The complete LPL syntax is presented as the extended Backus-Naur form. The following symbols are meta-symbols (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 boldface. The starting symbol is `Model`.

```
Model =          model ModelHeader StatSeq end
ModelHeader =    Id [Parameters] [Attrs] ";"
StatSeq =        { NEntity | Model | IEntity | ForStat | WhileStat | IfStat |
                 Expr | AssignStat | TableStat }
NEntity =        [Type] [NKeyword] Id IList Attrs ";"
IEntity =        IKeyword [Id] [IList] Attrs ";"
Type =           integer | binary | alldiff | string | date
NKeyword =       set | parameter | variable | expression | constraint
IKeyword =       maximize | minimize | solve | addconst
IList =          "{" Index {"," Index} ["|" Expr] "}"
Index =          [LocIds] QualId ["[" Id {"," Id} "]"] | [LocIds] Expr ".."
                 Expr
LocIds =         [Id | "(" Id {"," Id } ")"] in
Attrs =          { frozen | default (Number|String) | "[" Expr "]" | Comment |
                 String | if Expr | subject_to SubjExpr | priority Expr | AssOp
                 Expr }
AssignStat =     QualId IList Attrs ";"
TableStat =      "*" Id Ilist AssOp Table ";"
ForStat =        for IList do StatSeq end
WhileStat =      while Expr do StatSeq end
IfStat =         if Expr then StatSeq [else StatSeq] end
SubjList =       [not] QualId { "," [not] QualId }
SubjExpr0 =      if "(" BoolExpr "," SubjList { "," BoolExpr "," SubjList } ")"
SubjExpr =       SubjList | SubjExpr0
Expr =           SimpleExpr {Relation SimpleExpr}
SimpleExpr =     [Indexing] Term {MulOp Term}
Term =           [ "+" | "-" | "#" | "~" | "$" ] Factor
Factor =         Number | Id within QualId | "(" Expr ")" | QualId ["[" Expr
                 "]"] | "«" Expr "»" | Funct [ Ilist ] "(" Expr ")" | String
Relation =       "," | ":=" | ".." | "->" | "<-" | "<->" | xor | or | nor | and
                 | nand | "=" | "<>" | "<" | "<=" | ">" | ">=" | "+" | "-" |
                 "&"
MulOp =          "*" | "/" | "^" | "%" | "&&" | "||"
Indexing =       [IndexOp] IList
IndexOp =        or | xor | nor | and | nand | exist | for | forall |
                 prod | max | min | argmax | argmin | sum | count |
                 (atleast|atmost|exactly) "(" Int ")" | for
Funct =          a-function-name, see functions
Table =          TableA | TableB | TableC
TableA =         "[" {Data} "]"
TableB =         "/" {SubTable} "/"
TableC =         "/" "1" ":" Int "/"
SubTable =       [("(" | "(" TemplateOpt ")" | ")")] [ColonOpt] ListOpt
TemplateOpt =    EleOrStar { "," EleOrStar }
ColonOpt =       ":" ["(tr)"] {Element} ":"
ListOpt =        { {Element} {Data} ["/"] }
Data =           Number | "." | String | "(" TemplateOpt ")"
Element =        Number | Id | String
EleOrStar =      element | "*"
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
```

Bibliography

[1] Knuth D.E. Literate Programming. *Computer Journal*, 27,2(May):97–111, 1984.

[2] Hürlimann T. Index Notation in Mathematics and Modeling Language LPL: Theory and Exercises. https://matmod.ch/lpl/doc/indexing.pdf.

[3] Hürlimann T. Logical modeling. https://matmod.ch/lpl/doc/logical.pdf.

[4] Hürlimann T. Various Model Types. https://matmod.ch/lpl/doc/variants.pdf.