

# **LPL : A MATHEMATICAL PROGRAMMING LANGUAGE**

**Tony Hürlimann**

**Working Paper**

**December 1996  
updated May 1997  
updated September 1997  
updated June 1998  
updated April 1999**

**(an former version was published in: OR Spektrum 15:43–55)  
(the same paper is available in German)**

---

*INSTITUT D'INFORMATIQUE, UNIVERSITE DE FRIBOURG* **iiUF**  
*INSTITUT FÜR INFORMATIK DER UNIVERSITÄT FREIBURG*

*Institute of Informatics, University of Fribourg*

*site Regina Mundi, rue de Faucigny 2, CH-1700 Fribourg / Switzerland*

*tony.huerlimann@unifr.ch*

*phone: ++41 26 300 2845 fax: ++41 26 300 9726*

---

This research is supported by the Swiss National Science Foundation and financed by the project no. 12-55989.98.

## **LPL : A Mathematical Programming Language**

Tony Hürlimann, Dr. lic. rer. pol.

Keywords: Modeling, linear and nonlinear optimization.

**Summary:** This paper describes the version 4.34 of the modeling language, named LPL (*Linear Programming Language* or *Logic Programming Language*). It may be used to build, modify and document (linear and nonlinear) mathematical and logical models. The LPL language has been successfully applied to generate automatically MPS input files and reports of large LP models. The available LPL compiler translates LPL programs to the input code of any LP/MIP solver, calls the solver automatically, reads the solution back to its internal representation, and the integrated Report Generator produces the user-defined reports of the model. Furthermore, an Input Generator can read the data from many formats.

Stichworte: Modellierung, lineare und nicht-lineare Optimierung.

**Zusammenfassung:** Dieser Artikel beschreibt die neue Version der Modellierungssprache LPL (*Linear Programming Language* oder *Logic Programming Language*), die sich dazu eignet, mathematische (lineare und nicht-lineare) Modelle aufzubauen, zu warten und zu dokumentieren. Die LPL-Sprache wurde zum Erstellen von MPS-Input-Dateien und Resultate-Tabellen größerer LP-Modelle erfolgreich eingesetzt. Der LPL-Compiler übersetzt ein LPL-Programm, das ein vollständiges Modell repräsentiert, in den Eingabecode eines LP/MIP-Lösungsprogramms, ruft den Lösungsalgorithmus auf, liest die Lösung, und ein integrierter Tabellengenerator gibt vom Benutzer definierte Resultate-Tabellen aus. Außerdem erlaubt ein Dateneingabe-Generator, die Daten in verschiedenen Formaten zu lesen.

## 1. INTRODUCTION

This paper presents the new version 4.34 of the modeling language LPL. A first version of LPL was developed and implemented ten years ago. It was described in [Hürlimann, Kohlas 1988]. It was restricted to linear models. The new version, exposed in this paper, has been improved in several respects: A powerful Input and Report Generator has been integrated, the expression syntax has been enlarged with several new functions and operators, units can be used to measure numerical entities, string manipulation is now possible in a restricted way, goal and multi-stage programming are also supported to some extent, and finally, an open interface to most LP/MIP and nonlinear solvers has been added. Many more modifications and, hopefully, improvements have taken place, but will not be mentioned here. A speciality of the old LPL version was its hierarchical indexing: Indices were nested lists which could be used as any other index within a model. To use nested lists for indexing is very intriguing, since many sets of objects can be arranged as a tree in a natural way. Practical experiences, however, have shown that they obscured the model structure – at least in the form they have been implemented in LPL. Furthermore, few modeling examples came around which used this option really. Many examples could be formulated more simply by using relations. It was, therefore, decided to drop hierarchical indexing from the LPL language. More research work should be done, before hierarchical indexing can be integrated in an advanced modeling language. A step in this direction was undertaken in [Hürlimann 1998b].

Different modeling languages have been developed. AIMMS [Bisschop 1998], AMPL [Fourer al. 1993], GAMS [Brooke al. 1988], LINGO [Schrage 1998] are closest to the LPL language.

To summarize briefly the main features of LPL, they are:

- A simple syntax of models with indexed expressions close to the mathematical notation, and directly applicable for documentation,
- Formulation of both small *and* large LP's with optional separation of the data from the model structure,
- Availability of a powerful index mechanism, making model structuring very flexible.
- An innovative and high-level Input and Report Generator.
- Intermediate indexed expression evaluation (much like matrix manipulation).
- Tools for debugging the model (e.g. explicit equation listing).
- Built-in text editor to enter the LPL model.

- Fast production of the MPS file and other output-files, such as a T<sub>E</sub>X file which generates a embellished and documented version of the model..
- Open interface to most LP/MIP and non-linear solver packages. This means that any linear or nonlinear solver can be called from within the LPL code. The user can configure the communication between LPL and the solver.
- LPL includes a Unit statement for measurement checks.
- Furthermore, LPL allows the user to generate alias-names for variables and restrictions.

<b>Indices</b>	
$j$	bonds ( $j = 1, \dots, N$ )
$t$	time periods ( $t = 1, \dots, T$ ) with $T \leq 50$
<b>Data Tables</b>	
$c_j$	current market price for bond $j$ (in 100\$)
$f_{jt}$	cash flow produced by bond $j$ at the end of period $t$ (in \$/piece)
$L_t$	cash requirement at the end of period $t$ (in \$)
$q_j$	conditional minimum purchase quantity of bond $j$ (in pieces)
$Q_j$	maximum allowable purchase of bond $j$ (in pieces)
$a_t$	re-investment rate for period $t$ (in % (percent))
<b>Variables</b>	
$x_j$	quantity of bond $j$ to be purchased here and now (in pieces)
$s_t$	cash surplus to be accumulated at the end of period $t$ (in \$)
$d_j$	is 1, if bond $j$ is selected for portfolio, else it is 0
<b>Minimize</b>	
$\sum_{j=1}^N c_j x_j + s_0$	the sum of bond purchase plus initial cash
<b>subject to</b>	
$\sum_{j=1}^N f_{jt} x_j + a_t s_{t-1} - s_t = L_t$	for $t = 2, \dots, T$ balance in each period: cash from coupon + cash surplus from previous period = liabilities)
$q_j d_j \leq x_j \leq Q_j d_j$	for $j = 1, \dots, N$ either $x = 0$ or $q \leq x \leq Q$
$x_j \geq 0, s_t \geq 0$	and $d_j \in \{0, 1\}$

**Figure 2-1**

This paper is organized as follows: A simple example demonstrates the overall structure of an model coded in LPL. The basic syntax of LPL is presented in section 3. Section 4 and 5 examine the indices and expressions, the most important constructs of LPL. Some aspects of the Report and Input Generator are discussed in section 6. Appendix A lists a complete assignment model containing 2700 0–1 variables (assign 180 players to

15 teams), which covers more interesting features of LPL.

## 2. A SIMPLE EXAMPLE

To illustrate the modeling language LPL, let's define the following simple portfolio optimization problem:

Cash in a company has to be invested into bonds at each period in such a way that the requirements and forecasted liabilities can be satisfied. The purchasing strategy should be to buy the bonds at minimal costs which permit to fulfil the cash requirements. Which types of bonds and how much has to be bought at each period?

Figure 2-1 displays the algebraic formulation of a simple portfolio model [Shapiro 1988, p. 589ff]. This formulation can be translated directly into an LPL model formulation as shown in Figure 2-2.

The algebraic formulation of Figure 2-1 contains different sections: Index-sets, numerical data-tables, variables (the unknowns), a minimization function, and different constraints. Note that this formulation in Figure 2-1 represents only a model *structure* not a specific, *instantiated* model, since no data are defined. To produce a specific model, it must be supplemented by the values of all index-sets and data-tables.

```

MODEL PORTFOL1 "Fixed income portfolio selection, a deterministic model";

SET
  j      "bonds";
  t      "time periods";

UNIT    -- units of measurement
  dollar "units of money";
  percent "Percent (%)" = 1/100;
  pieces "number of bonds";
  uPrice "price per piece" = dollar/pieces;

PARAMETER
  c{j} UNIT [100*uPrice] "current market price for bond j";
  f{j,t} UNIT [uPrice] "cash flow produced by bond j in period t";
  q{j} UNIT [pieces] "conditional minimum purchase quantity of bond j";
  Q{j} UNIT [pieces] "maximum allowable purchase of bond j";
  a{t} UNIT [percent] "reinvestment rate (return at the end of period t)";
  L{t} UNIT [dollar] "cash available in period t";

VARIABLE
  x{j} UNIT [pieces] "quantity of bond purchased";
  s{t} UNIT [dollar] "cash surplus";
  BINARY d{j} " =1 if bond is selected else =0";

CONSTRAINT
  Invest UNIT [dollar] "sum of purchased bonds plus initial cash"
    : SUM{j} c*x + s[1];
  CashBal{t|t>1} UNIT [dollar]
    "coupon return + cash surplus from t-1 = liabilities"
    : SUM{j} f*x + a*s[t-1] - s = L;
  C{j} UNIT [pieces] "either x=0 or q <= x <= Q"
    : q*d <= x <= Q*d;
  initS UNIT [dollar] "initial cash"
    : s[1] = 0[dollar];

```

```

(*$I 'PORTFOL1.INC' *)      -- read the data
MINIMIZE Invest;          -- solve the model
WRITE Invest; x; s; d;    -- report the results
END

```

**Figure 2-2**

Figure 2-2 shows the entire model structure in LPL syntax. The different sections of index-sets, data, variables, minimizing function, and constraints are headed by the reserved words SET, PARAMETER, VARIABLE, CONSTRAINT, and MINIMIZE. Units can be defined using the UNIT statement. Comments can be added anywhere between quotes "... " or within (\* ...\*) or using a double dash (--). Comments between quotes are qualified, that is, they are remembered by the compiler. The other comments can also be used to document a model, but they do not belong to the formal part of the model.

Since an LPL model can be processed directly by the LPL compiler, its formulation has some particularities compared to the algebraic notation: The sigma sign  $\Sigma$  is replaced by the reserved word SUM; subscript indices are listed between parentheses; a semicolon must end every declaration. It is also possible to use multi-letter names in place of single-letter names. Hence,  $c(j)$  might be replaced by *marketprice(bond)*.

Four additional instructions are found in the LPL model in Figure 2-2:

- *{\$I...* reads the model data from file PORTFOL1.INC written also in LPL syntax (see Figure 2-3),
- *MINIMIZE...* calls the solver and reads the solution back to the LPL internal representation,
- *WRITE...* prints the required tables to the output file, called NOM-file,
- the *UNIT* statements used within the model (see 3.7).

```

-- portfol1.inc: a data set for the Portfolio model
SET
  j =/ | c    q    Q |
-----
  A1  2    10   500
  A2  2.3  .    700
  A3  4    20   700
  A4  1    15   800
  A5  2.4  20   900  /;

  t =/ | L    a |
-----
  T1    .    90
  T2  1200   90
  T3  1400   80
  T4   500   80  /;

PARAMETER f = /
          : T2  T3  T4:
-----
  A1      4    4    4

```

```

A2      5.5  5    4
A3      5    3    6
A4      6    6    .
A5      4    5    6  /;

INTEGER
  TMAX [0,100] = 50;  -- new data not define in the main program

-- check data consistency
CHECK This{j} UNIT [pieces]: q < Q;  -- each q must be smaller than Q
CHECK This: #t <= TMAX;  -- the cardinality of t must be smaller than TMAX
-- end of data --

```

### Figure 2-3

A specific data set for this model is shown in Figure 2-3 written in LPL syntax. The data-tables can also be in plain text. LPL includes a flexible Input Generator to read external data definition (see section 6).

(The files PORTFOL1.LPL and PORTFOL1.INC can be found in the folder 'models' of the LPL-site: <ftp://ftp-iiuf.unifr.ch/pub/lpl/>.)

## 3. A BRIEF OVERVIEW OF THE LPL LANGUAGE

In this section, the different components of an LPL model are briefly presented.

### 3.1 INDEX-SETS

An index-set is an unordered or ordered collection of objects. They are called *elements*. The set of bonds  $j$  in our example is such a set. A set is declared in LPL as

```
SET j "bonds" ;
```

The elements themselves of this set are not defined in this place, they are part of the model data defined in file PORTFOL1.INC (Figure 2-3). The modeler is, however, free to declare a set and to assign its elements to the same place within the model. The set  $j$  could have been defined as

```
SET j = /1:5/;  -- j is a set of five bonds
```

The two elements  $1$  and  $5$  separated by a colon define a range of elements, which is the same as

```
SET j = /1 2 3 4 5/;
```

where every element is mentioned explicitly. The element-names  $1, \dots, 5$  might also be replaced by more meaningful names as in

```
SET j = /World_bank88 Treasury_bills Sandoz_baby EFF ATT87/;
```

The comment "bonds" is a statement attribute belonging to the specification of the set  $j$ . It comments briefly the entity.

Index-sets may be indexed, in which case they define a tuple-list of its indices. If the sets  $p$  and  $t$  are defined in the following way (see Appendix A: the soccer model):

```
SET p = / 1:180 /;                -- a list of 180 players
SET t = /T1 T2 T3 T4 T5 T6 T7 T8 T9 T10 T11 T12 T13 T14 T15/;  -- 15 teams
```

then an indexed set *Must\_be\_in* can be specified as a list of three elements

```
SET Must_be_in{p,t} = / 1 T2 , 2 T6 , 34 T7 /;
```

This tuple-list defines a relation *Must\_be\_in* between the players  $p$  and the teams  $t$ . It tells us, that player  $P1$  belongs to team  $T2$ , player  $P2$  to team  $T6$ , and player  $P3$  to team  $T7$ . It is important to note, that this relation assigns only a sparse sublist of the whole (Cartesian) tuple-list. Another example is to define sublists of players as

```
SET pla1{p} = / 1 45 56 67 78 122 /;
SET pla2{p} = / 2 67 123 145 12 178 /;
```

Indexed sets may also be assigned by logical expressions. The union, the intersection and the difference of set *pla1* and *pla2* may be defined as

```
SET union{p} = pla1 or pla2;
SET intersection{p} = pla1 and pla2;
SET difference{p} = pla1 and not pla2;
```

Index-set is one of the most important components in any large-scale LP model. LPL offers a broad variety of different set-types.

### 3.2 NUMERICAL DATA

Numerical data within the model are collected in *coefficients*. Its declaration is headed by the reserved words INTEGER and PARAMETER. The simplest coefficient consists of only one value:

```
INTEGER PARAMETER TMAX [0,100];
```

TMAX is declared as a coefficient, but its value is not yet known. The declaration, however, tells us that TMAX must be an integer value within the range [0,100]. The LPL compiler tests these conditions and reports an error, if they are violated. LPL offers also the CHECK statement which can check the data consistency using more complex conditions. The instruction

```
CHECK This{j} UNIT [pieces]: q < Q; -- q must be smaller than Q for every j
```

checks for every  $q$  over  $j$ , if  $q$  is strictly smaller than  $Q$ .

Coefficients can also be indexed, and their values are collected in tables.  $c\{j\}$  is such a coefficient in our example. It declares a marketprice  $c$  for every bond  $j$ . The conditional minimum purchase quantity  $q$  and the allowable purchase  $Q$  are also indexed over  $j$ . Note that LPL does distinguish lower and upper-case letter by default, therefore,  $q$  and

$Q$  are distinct in this model.

The first table of Figure 2-3 is a convenient way to define the set  $j$  together with the three coefficients  $c$ ,  $q$ , and  $Q$ . This is possible, because all three coefficients run over the same index  $j$ . This is, however, not the only way to enter the data. The modeler is free to declare and define the data within the model structure as

```
SET j = /1:5/;
PARAMETER c{j} = [ 200 230 400 100 240 ];
PARAMETER q{j} = [ 10 . 20 15 20 ];
PARAMETER Q{j} = [ 50 70 70 80 90 ];
```

LPL offers different table formats for the data. But the most flexible way to get the data from external formats is to use the Input Generator.

Indexed coefficients are not restricted to one-dimensional tables (that is, with one index). They can be two- three- or higher-dimensional.  $f\{j,t\}$ , for example, declares a two-dimensional coefficient, where the cash flow  $f$  is defined for every bond  $j$  within every time period  $t$ . Data tables are reassignable.

```
PARAMETER a = 10; (* 10 is assigned to the coefficient a *)
PARAMETER b = a; (* the value of a is copied to b (10) *)
PARAMETER a = 20; (* a gets a new value 20, but b is still 10 *)
```

### 3.3 VARIABLES

Variables have the same properties as coefficients. They can also be defined as multi-dimensional objects and numerical values can be assigned to them. The only differences are, that their declarations are headed by the reserved word `VARIABLE` and their values are usually assigned under the solver's control. A typical variable declaration is

```
VARIABLE x{j}; -- purchased quantity x of bond j
```

The variable  $x$  is declared over  $j$  and may be interpreted as a numerical one-dimensional table of unknown values. But the modeler may also assign values to them. They are used in nonlinear models as initial values from where a solver can start searching. It is also possible to restrict the values of the variables. Lower and upper bounds on variables are often used. Sometimes variables must be integers. These options can be added to any variable declaration as in

```
INTEGER VARIABLE d{j};
```

The declaration of  $d$  declares a variable over set  $j$ . If the reserved word `INTEGER` is replaced by `BINARY`, its values can only be the integers 0 or 1. These restrictions are automatically translated by the LPL compiler as `BOUNDS` and `INT-MARKERS` in the `BOUND-` and `COLUMN-`Section of the MPS input-code for a LP/MIP solver, a standard file for storing linear models.

### 3.4 CONSTRAINT

The model restrictions are declared in the `CONSTRAINT` statement. Each restriction begins with a name. The optimizing function can be included within the list. The restriction name is followed by a colon and a linear expression. Restrictions can also be indexed.

The restriction

```
CONSTRAINT C{j} UNIT [pieces] : q*d <= x <= Q*d;
```

is defined for every element of the set  $j$ . This generates as many single restrictions as  $j$  has elements. Any algebraic or logical expression is allowed for defining constraints. Summations begin with the reserved word `SUM`. The term

```
... SUM{j} c[j]*x[j] ...
```

for example, sums the product  $c*x$  over the set  $j$ . This is close to the mathematical notation. The indices of  $c[j]$  and  $x[j]$  can also be dropped if no ambiguities arise from the expression. This simplifies the term to

```
... SUM{j} c*x ...
```

The declaration of restrictions can also be separated from their definition. The declaration

```
CONSTRAINT
  Invest UNIT [dollar] "sum of purchased bonds plus initial cash";
  CashBal{t|t>1} UNIT [dollar]
    "coupon return + cash surplus from t-1 = liabilities";
  C{j} UNIT [pieces] "either x=0 or q <= x <= Q";
  inits UNIT [dollar] "initial cash";
```

is perfectly correct. The assignment of the restriction may take place in a different model part. Constraints can also be made inactive using the keyword `INACTIVE`. Such constraints are not passed to the solver. This makes it possible – using the solver statement (`MINIMIZE` or `MAXIMIZE`) in several places within the model – to call the solver repeatedly within an LPL model with different model variants. Another useful application of this option is multi-goal programming.

The objective function begins with the reserved word `MINIMIZE` or `MAXIMIZE`, depending on whether the function is minimized or maximized. This instruction calls the solver directly.

### 3.5 THE SOLVER

LPL has no integrated solver, but can call an external solver automatically. The interface between most available LP/MIP solvers and LPL can even be specified by the modeler. For nonlinear solver an interface must be programmed, details can be found in [Hürlimann 1998g and 1998h]. The command `MINIMIZE` or `MAXIMIZE` instructs

the LPL compiler to produce the MPS-file, the standard solver input file for linear models, or some other files, then to call the solver itself with the right parameters, and reads the solution back to the LPL internal data structure. The interface is explained in detail in the reference manual [Hürlimann 1998]. By default, the LPL's internal simplex solver is called, but other solver packages such as XA, CPLEX, MOPS or LINDO work too. LPL even has a heuristic solver integrated for certain scheduling problems. Non-linear solvers must be added.

### 3.6 REPORTS

LPL does not only allow one to formulate a complete model, but can also produce model reports. This is an integrated part of LPL. The reserved word WRITE is used to generate even most complex reports. The simplest reports are generated using the syntax shown in Figure 2-2 as

```
WRITE Invest; x; s; d;
```

which prints four tables in a predefined format.

A more complex WRITE statement is found in the Appendix A, and is explained in § section 6.

### 3.7 UNITS

Every declaration of numerical entities can be extended by indicating the units. This increases the reliability and the readability. Furthermore, explicit declaration of units gives the compiler additional checking power that may reduce the number of syntactical as well as semantic errors. Every expression is checked of unit commensurateness before it is evaluated. Automatic unit conversion takes place without the intervention of the user.

Units must be declared using the Unit Statement beginning with the keyword UNIT. *Basic units* (such as 'meter') are just declared by their name. *Derived units* (such as 'speed=meter/sec') are declared by their unit expression

```
UNIT
meter;          (* a basic length measure *)
kilo = 1000;    (* a derived unit commensurable to type number *)
km = kilo*meter; (* a derived measure, commensurable to 'meter' *)
cm = m/100;     (* a derived measure, also commensurable to 'meter' *)
speed = meter/sec; (* another derived unit, not commensurable to 'meter' *)
```

The use of units within the model structure as well as within the data tables (defined in LPL) is simple: just extend the declaration with the reserved word UNIT and add a unit expression within brackets. A number within an expression must be extended by [*<unit expression>*]. The Report Generator too accepts units. The table *invest* might be written in \$1000 instead of \$ to the output device. To do this, one may add the unit expression as follows

```
WRITE invest UNIT [1000*dollar];
```

Note that units are optional. The modeler may or may not use them within the model.

The different parts of an LPL model have now been described very briefly. LPL contains many other features, such as logical modeling, but we shall not go into these more advanced features. Several fundamental aspects of LPL will be explained now in greater detail in the subsequent sections. Most examples are from the soccer model, listed in Appendix A.

#### 4. THE USE OF INDICES

The indices are used to define multi-dimensional items, such as coefficients, variables, restrictions, or indexed sets. They are called *tables*.

```
PARAMETER a{i,j} = 1;
VARIABLE x{i,j,k,l};
CONSTRAINT r{i};
SET s{i,j};
```

$a$  declares a two-dimensional numerical table – a matrix – and assigns the value 1 to every table-entry,  $x$  is a four-dimensional variable,  $r$  is a restriction-vector, and  $s$  is a two-dimensional indexed set. In the same manner, the indices are used for the different *index-operators* (SUM, PROD,...). They are operators, which iterate over index-sets.

```
....SUM{i,j} a[i,j]...
....PROD{i,j,k,l} x[i,j,k,l]...
```

The first example returns the sum of all values in the table  $a[i,j]$ , the second returns the product of all  $x[i,j,k,l]$ .

In the last examples, the indices play an *active* part. In the expression *PARAMETER*  $a\{i,j\}=1$ , for example, the indices  $i$  and  $j$  determine how many entries the table  $a$  contains, and the same indices decide how many terms the summation extends in the expression *SUM* $\{i,j\} a[i,j]$ . One may compare this syntax with the nested loops in a imperative programming language. The view is, however, not quite correct. Since LPL is a declarative language, the assignment is in fact a definition: all elements of the matrix  $a$  are 1.

Indices are also used in algebraic expression (within the brackets  $[...]$ ) in the above examples, where they play a *passive* part. In the expression

```
PARAMETER a{i,j} = b[i,j] + 1;
```

every table-entry of  $b$  plus one will be copied to the corresponding table-entry of  $a$ . The indices in  $[i,j]$  play a passive part. Every passive index must be bound to an active index. In the last example, the index  $i$  in  $[i,j]$  will be bound to the index  $i$  in  $\{i,j\}$ , and  $j$

in  $[i,j]$  is bound to  $j$  in  $\{i,j\}$ . The binding guarantees a unique expression evaluation. In the assignment

```
PARAMETER a{i,i,j} = b[i,j]; (* not an error, but... *)
```

the index  $i$  in  $[i,j]$  could be bound to the first or second  $i$  within the index-list( $\{i,j\}$ ). Therefore, a unique binding is not possible. (By default LPL will bind it to the last, but this may be implementation-dependent).

Binding is flexible in LPL. It is, however, necessary that the bound and the binding indices represent the same index-set. An expression such as

```
PARAMETER a{i,j} = b[i,k]; (* binding error! *)
```

produces an error. It must be replaced by

```
PARAMETER a{i,j} = b[i,j IN k]; (* correct *)
```

$j$  IN  $k$  returns the position of a specific element of  $j$  within the set  $k$ . (In this case,  $k$  is regarded as an ordered set). If the specific element of  $j$  is not in  $k$ , the whole expression  $b[i,j$  IN  $k]$  returns zero. If the intersection of  $j$  and  $k$  is a real subset of  $k$ , then the table  $a$  will be filled up only partially by the assignment, the rest of the table  $a$  will remain unchanged.

In general, the LPL compiler tries to bind the passive indices automatically, identified by their names. The modeler, however, has the possibility to force a binding using *dummy-indices*. The corresponding active indices must be headed by a dummy-index, followed by an equal sign or the IN reserved word.

```
PARAMETER a{d1=i,d2=j} = b[d1,d2];
PARAMETER a{d1 IN i,d2 IN j} = b[d1,d2]; (* the same *)
```

The dummy-indices  $d1$  and  $d2$  can then be used instead of the passive indices within the expressions. This forces the binding from  $d1$  to  $i$  and from  $d2$  to  $j$ . Every active index within an LPL model can be extended by a dummy. The same dummies may be used in different expressions since they have only local effect. (They are really treated like local variables in programming languages). In some situations, dummies are necessary, but in most contexts they may be dropped. It is a matter of style, of whether the modeler wants to use them systematically or not. LPL even makes it possible to drop all passive indices, if they can be bound uniquely. Since this relaxed use of indices within LPL models might be dangerous, LPL also offers a compiler switch, which restricts this practice (see [Hürlimann 1998] for details).

Indices can also be used as terms within expressions. In this case, they return the position of an element within the index-set. The expression

```
PARAMETER a{i} = i;
```

assigns the values 1, 2, 3, ... , and  $n$  to the table  $a$ , if  $n$  is the cardinality of index-set  $i$ .

If the index, however, is used within a WRITE expression, then not the element itself is returned by the expression (as string) not the position. An example is:

```
WRITE{i} : i , a[i];
```

In this instruction  $i$  and  $a[i]$  are written to a file one by one. Not the element position is written but the element name.

## 5. ALGEBRAIC AND LOGICAL EXPRESSIONS

Algebraic and logical expressions are an important part of any LPL model. They are used in three different contexts:

- to define model restrictions
- to evaluate and to output intermediate expressions and tables
- to declare sparse, indexed tables.

Expressions are built using the usual mathematical notation with arithmetical (+ - \* / ^) and logical (*and*, *or*, *not*) operators, coefficients, numbers and functions (Figure 5-2). They may also contain *index-operators* (Figure 5-3), which are iterated over index-sets. The sum-operator  $\Sigma$ , replaced by the reserved word SUM, is such an index-operator. LPL, like the programming language C, does not distinguish between algebraic and logical expressions. In logical expressions zero is used as FALSE and any other number is interpreted as TRUE. Figure 5-1 gives an overview of all operators in LPL in order of decreasing precedence. Every expression returns a numerical value.

Coefficients or variables within an expression return the corresponding value. If a coefficient or variable has not been assigned before in the model, a default value is taken. The default value can be entered through the reserved word DEFAULT following a declaration as in

```
PARAMETER a{i} DEFAULT 2;
```

If no default value has been declared by the modeler, it is zero.

Examples of expressions are

```
4.6e5 + 7^9 - SIN(879.8)
(((4+5)*4)^2 - 12 ) + IF(a>0,3,7)
a OR b AND NOT (3*4)
SUM{i} (a[i]+b[i])/2
PROD{i} x[i,j]
```

The last expression contains an unbound index  $j$ . Therefore, it returns not one *single* numerical value, but a one-dimensional table of values over  $j$ . The index can be bound, if the expression is assigned to a table  $b$ , or if another index-operator is added to the expression:

```
PARAMETER b{j} = PROD{i} x[i,j];
... + SUM{j} (PROD{i} x[i,j] ...;
```

SIN LOG ...	functions
+ - NOT # IN	unary operators
^	power
* / %	product and division
SUM PROD ...	index-operators
+ -	addition and subtraction
= <> < > <= >= ~	relational operators
AND NAND	logical AND
OR NOR	logical OR
XOR	exclusive OR
, (or  )	enumeration operator
:=	assign operator

**Figure 5-1 (Operators)**

max(x,y)	returns x, if x>y, else returns y
min(x,y)	returns x, if x<y, else returns y
abs(x)	returns the absolute value of x
ceil(x)	returns next integer greater than x
floor(x)	returns next integer less than x
trunc(x)	returns the truncated x
sin(x)	returns the sinus of x
cos(x)	returns the cosines of x
log(x)	returns the nat. logarithm of x
sqrt(x)	returns the root of x
arctan(x)	returns the arctan of x
rnd(x,y)	returns a uniform distributed number
rndn(x,y)	returns a normal distributed number
if(x,y,z)	returns y, if x is TRUE, else returns z

**Figure 5-2 (Functions)**

SUM{i} a[i]	sum all a over i
PROD{i} a[i]	multiply all a over i
MIN{i} a[i]	the smallest a[i]
MAX{i} a[i]	the biggest a[i]
EXIST{i} a[i]	tests, whether all a[i] is FALSE
FORALL{i} a[i]	tests, whether all a[i] are TRUE
XOR{i} a[i] ...	several logical operators
PMAX{i} a[i]	position i of the biggest a[i]
PMIN{i} a[i]	position i of the smallest a[i]
COL{i} ...	horizontal table-expander
ROW{i} ...	vertical table-expander

**Figure 5-3 (Index-operators)**

It is important to see, that LPL allows one to evaluate and to return indexed expressions and not only single values. Insofar, LPL is more a table-manipulator language than a

modeling language.

### 5.1 CONSTRAINTS

Expressions allow the definition of model constraints. The syntax of an expression representing a constraint is very general. Logical operators are allowed in constraints, which are – using a sophisticated algorithm – translated to 0–1 constraints by default. Furthermore, the expression may be nonlinear relative to the defined variables. Constraints may also be defined as ranges. The same variable may occur several times within the expression. If the constant expression of a variable evaluates to zero, then the corresponding variable is automatically eliminated from the restriction. Constraints containing only one variable are automatically transformed to bounds.

```

CONSTRAINT R: x + y = SUM{i} z[i];    (* defines a restriction R *)
CONSTRAINT B: a <= x+y <= b;         (* defines a range *)
CONSTRAINT L{i}: x[i] + y = 5;       (* defines a restriction over i *)
CONSTRAINT S: x + 3*y = 4*(x+y);     (* x and y occurs twice *)
CONSTRAINT T: x + z -y = (4-4)*x +x; (* x will be eliminated *)

```

The restriction *S* will be automatically simplified to  $-3*x - y = 0$ , and in *T* the variable *x* is eliminated.

### 5.2 EVALUATING TABLES

Expressions are also used to produce intermediate results.

```

PARAMETER a{i,j} = b[j,i];
PARAMETER d{i,j} = a[i,j] + b[i,j];
CONSTRAINT INACTIVE e{i,k} = SUM{j} a[i,j]*c[j,k];
WRITE{i,j} : SUM{k} c[j,k] + a[i,j];

```

The first assignment copies the transposed table *b* into the table *a*. The second assigns the addition of the two matrices *a* and *b* to the table *d*. The third defines a matrix-multiplication, but the statement is not executed at this point, the evaluation is delayed instead: each time *e* is used in another expression the term at the right hand side is evaluated. The fourth calculates and outputs a two-dimensional table.

### 5.3 SPARSE TABLES

Another important application of expressions is the definition of sparse tables. This is explained by an example. Suppose one has to define a transportation model; the following model components are needed:

```

SET i ALIAS j;                (* a list of locations *)
SET links{i,j};              (* a list of links between the locations *)
PARAMETER costs{i,j};       (* the transportation costs of the links *)
VARIABLE x{i,j};            (* the unknown transportation quantities *)

```

Note first that the set *i* has two names (*i* and *j*) which point to the same set, but are

distinct names. The indexed set *links* defines which transportation links exist between the different locations *i*. Normally, this list is a (sparse) subset of all possible link-combinations. Since the links exist between some but not all locations, it is clear that the transportation costs make sense on the existing links only. Suppose, furthermore, that a quantity *x* is only transported on links with costs less or equal to 1000. In LPL it is possible to define such sparse tables in the following way:

```
PARAMETER costs{i,j | links[i,j]};
VARIABLE x{i,j | links[i,j] and costs[i,j]<=1000};
or even better:
PARAMETER costs{links};
VARIABLE x{links | costs<=1000};
```

*Costs* is defined over  $\{i,j\}$  if *links*[*i,j*] is TRUE. If the expression – headed by a '|' – evaluates to TRUE, the tuple exists; otherwise it is not defined. The subsequent *use* of the variable *x* restricts the list automatically to the existing ones. Therefore, the expression

```
... + SUM{i,j} x[i,j] + ...
```

does not sum *all*  $\{i,j\}$ -tuples of *x*, but only the existent ones defined in *links* and restricted by the expression *costs*<=1000.

The same syntax may be used for the iteration process of the index-operators too. An example is the following expression: Sum the values of the table *a*(*i*), such that  $i \leq 5$

```
SUM{i | i<=5} a[i];
```

The portfolio model used this option in the *Cash\_bal* constraints

```
CONSTRAINT Cash_bal{t|t>1}: SUM{j} f*x + a*s[t-1] - s = L;
```

This constraint is only defined for  $t > 1$ , but will not be generated for the period  $t = 1$ . Another example is the restriction *Must* in the soccer model of Appendix A

```
Must{p,t | Must_be_in}: Work = 1;
```

which would produce 2700 (=180x15) single restrictions without the condition *Must\_be\_in*, but actually produces only the following three constraints

```
Must1: Work[1,2]=1
Must2: Work[2,6]=1
Must3: Work[34,7]=1
```

since *Must\_be\_in* is defined as

```
SET Must_be_in{p,t} = / P1 2 , P2 6 , P34 7 //;
```

which is an indexed set containing three tuples.

Still a better method is to define the constraint *Must* as follows:

```
Must{i=must_be_in} : work[i] = 1;
```

This expression means that *Must* will not run through all {p,t} tuple just to record that 2697 of all 2700 constraints to not need to be generated as the first declaration does. It will only run through three constraints and generate them all. The speed difference in evaluating the two expressions is of course substantial.

## 6. THE REPORT AND INPUT GENERATOR

The WRITE statement allows the output of tables. They are written to a file, which may be further edited with the modeler's favoured text-processor. The simplest statements are the following, where LPL uses a standard layout of the output tables.

```
WRITE a;                (* write the table a in standard format *)
WRITE b '%2';          (* write the table b with length 2 *)
WRITE c '%10.2';       (* write table c with 10 positions and 2 decimals *)
WRITE a; b; c;         (* write all three tables *)
WRITE a '%6:10:10';    (* write the table a in blocks of 10x10 *)
```

### 6.1 WRITE TABLES OF EXPRESSIONS

LPL also allows us to output tables of any expression. In this case, the reserved word WRITE is separated from the expression by a colon. The layout is, once again, chosen by the LPL compiler.

```
WRITE      : a*b '%12';    (* write a value with length 12 *)
WRITE{j,i} : a[i,j];      (* write the transposed matrix a *)
WRITE      : 'this text';  (* write a text line *)
WRITE{j}   : SUM{i} a[i,j]; (* write the column totals of a *)
```

The reserved word WRITE is followed by an index-list, if the expression is indexed.

### 6.2 WRITING WITH USER-DEFINED MASKS

For more complex layouts, the modeler can define a layout mask through a Mask instruction. The Mask instruction is a quoted comment of a WRITE statement. The contents of the mask may be any string of characters. The two characters # and \$, however, have a special signification and are called *place-holder*. They are replaced by the numerical or string expressions of a subsequent WRITE statement.

Example:

```
WRITE
" Month: $$$$$$ : #####.##%  #####.#### gallons
"
: 'April' , 17.15*2 , 2378.567321 ;
```

These two statements produce the following output

```
Month:   April   :   34.30%   2378.5673 gallons
```

The WRITE statement consists of the reserved word WRITE, the comment (the mask content), and an expression-list separated by a colon. A comma separates the different

expressions within the list. The expressions are filled into the mask from left to right and from top to bottom at the different place-holders.

The mask feature, however, becomes most powerful together with the two index-operators ROW and COL and reveals the real power of the Report Generator. *The index-operator ROW {COL} can expand a place-holder vertically (horizontally) over index-sets.* The combination of both index-operators allows the user to produce rather complex tables. The syntax of both operators is the same as for any other index-operators.

Figure 6-1 shows the result of the WRITE statement using a mask of the model defined in Appendix A (some lines have been cut).

```

Results per player
*****

player Skill Age  in Team
-----
1       7   10   T2
2       7   10   T6
3       3   10   T6
4       3   10   T14
5       3   10   T4
6       5   10   T14
7       3   10   T11
... most lines have been cut here ...
171     6   11   T3
172     5   11   T1
173     3   10   T3
174     6   10   T2
175     3   10   T6
176     4   11   T3
177     7   11   T12
178     6   10   T15
179     5   10   T13
180     7   11   T13

Results per team
*****

team  skill  age  Av.Skill  Av.Age
-----
T1    60  128   5.000    10.667
T2    59  125   4.917    10.417
T3    59  125   4.917    10.417
T4    59  124   4.917    10.333
T5    61  126   5.083    10.500
T6    60  125   5.000    10.417
T7    59  127   4.917    10.583
T8    60  127   5.000    10.583
T9    59  125   4.917    10.417
T10   60  125   5.000    10.417
T11   60  125   5.000    10.417
T12   60  130   5.000    10.833
T13   60  127   5.000    10.583
T14   60  124   5.000    10.333
T15   60  125   5.000    10.417

Player to Team assignment
*****
team | players in the team
-----

```

T1	22	28	42	51	101	113	128	153	159	164	168	172
T2	1	29	45	58	62	93	115	130	132	151	160	174
T3	43	48	52	57	61	116	123	125	149	171	173	176
T4	5	9	17	41	60	74	80	86	89	114	126	144
T5	27	46	70	83	85	100	103	106	124	146	152	158
T6	2	3	13	25	30	56	87	92	134	136	170	175
T7	8	23	34	67	72	95	129	135	138	140	142	157
T8	12	14	16	31	32	36	66	79	105	119	145	161
T9	18	24	37	38	50	53	64	97	111	117	120	143
T10	49	63	73	75	76	91	110	112	118	162	166	167
T11	7	20	26	39	40	54	71	78	88	94	127	150
T12	10	11	21	55	59	102	109	121	133	148	154	177
T13	33	44	68	69	77	82	84	90	104	165	179	180
T14	4	6	35	47	65	81	98	108	131	147	155	169
T15	15	19	96	99	107	122	137	139	141	156	163	178

**Figure 6-1**

The mask contains 11 place-holders. The first four are defined on line 5 as

```
$$$$$ ## ## $$$
```

They are filled by the expression

```
(p , skill , age , COL{t|work} t)
```

in the WRITE statement as follows. The first place-holder is replaced by the name of player  $p$ , the second and third by the *Skill* and *Age* value of player  $p$ . The expression  $COL\{t|work\} t$  runs over all  $t$  for a specific  $p$  and writes all team names  $t$  horizontally for every  $work[p,t]$  that is TRUE. Since a player  $p$  can only be in one team, this expression writes only one team name per line. Because the whole expression is also defined over  $ROW\{p\}$ , the output of this line is repeated for every player  $p$ , which will produce 180 lines in our case. The next five place-holders are defined on line 13 of the mask. They are filled by the expression:

```
ROW{t} (t , SUM{p|work} Skill , SUM{p|work} Age ,
        (SUM{p|work} Skill)/12 , (SUM{p|work} Age)/12 ) ,
```

The first place-holder is filled by the team name  $t$ , the next by the total skill per team, calculated by the expression  $SUM\{p|work\} Skill$ . The other expressions are calculated similarly.

The last two place-holders in line 19 – the last line of the mask – are filled by the expression:

```
ROW{t} (t , COL{p|work} p);
```

Note that this simple line produces a whole two-dimensional table. The players are collected per line. The iteration of the ROW and COL index-operators can also be restricted by a condition. This allows us to select any subset of tables and to output the result in most complex layouts.

## 6.3 THE INPUT GENERATOR

The Input Generator is represented by the READ statement. It is similar to the WRITE statement, but instead of output data, it allows one to read data from text files. As an example, suppose the data of the portfolio model in Figure 2-3 are not organized in LPL syntax, but in a plain text file shown in Figure 6-2.

```
(* PORTFOL1.DAT: data file for the portfolio model PORTFOL1 *)
(* three tables are defined as follows *)

Table 1 : data of the different bond types
(*   bond   price   minimal   maximal
   type   (in $100) purchase   purchase
----- *)
      A1      2      10      500
      A2     2.3      .      700
      A3      4      .      700
      A4      1     15      800
      A5     2.4     20      900
Endtable 1

Table 2 : data per period
(*   Period   estimated   rate of
      Liabilities   reinvestment
----- *)
      T1      -      -
      T2     12000     90 %
      T3     14000     80 %
      T4      5000     80 %
Endtable 2

Table 3 : Coupon data
(*           T1   T2   T3   T4
----- *)
      A1      .   4   4   4
      A2      .  5.5  5   4
      A3      .   5   3   6
      A4      .   6   6   .
      A5      .   4   5   6
Endtable
(* end of data file *)
```

**Figure 6-2**

Now the data specification of the model (that is, the include statement in our example, see Figure 2-2) can be replaced by the four READ statements

```
READ FROM 'portfol.dat' ':Table:Endtable';
READ '%1' : ROW{j} ( j , c , q , Q );
READ '%2' : ROW{t} ( t , L , a );
READ '%3' : ROW{j} ( j , COL{t} f[j,t] );
```

The first READ statement indicates the filename of the input file and the *read-block-delimiters*, between which are used by subsequent read statements. The second Read jumps to the first occurrence of 'Table' (beginning on a new line within the read file) and reads lines until a 'Endtable' string, beginning at a new line, is encountered. On each line four tokens are read, if possible. The instruction *ROW{j}...* has two functions: it reads lines repeatedly, and synchronizes the read. This means: if a line

contains more than four tokens only the first four are read, if it contains less, less are read. In either case, a fresh read of four tokens will begin on a new line. The instruction  $COL\{t\}...$  repeats to read tokens on the same line until the line ends.

Note that eventually the index set of the active indices of the two operators COL and ROW does not exist yet. The statement:

```
READ '%1' : ROW{j} ( j , c , q , Q );
```

therefore, cannot mean: “read four words (token) from the file repeatedly as many elements the index set  $j$  has”, since the index set  $j$  is read only through this READ statement. In  $ROW\{j\}$  the  $j$  has a passive role insofar as the index set is constructed. It still binds the passive indices of  $c$ ,  $q$ , and  $Q$ , however. The READ statement could be compared to boot-strapping: generate the set by itself!

It is also possible to read from different files. The three tables could have been divided into three files. In this case, no block indication would have been necessary. The Input Generator skips empty lines or lines containing only separators between tokens, such as spaces or tabs automatically. Experiences with the Input Generator are promising: For an LP model with 1300 constraints and 1500 variables, a Pascal program of 32 pages had been written to manipulate the data; using the LPL Input Generator, it was possible to code the same program in 2 pages.

## 7. CONCLUSION

This paper is a brief report on the new version of the LPL modeling language. The Reference Manual [Hürlimann 1998] gives a detailed description of the language. A model library of about 160 examples written in LPL is available from the LPL-site.

The development of LPL was initially motivated by practical use. Different models with 1500-2000 restrictions, 2000-3500 variables, and a matrix density of 7500-12000 non-zero elements are under continuous use and development at our Institute for Informatics. Most of them are formulated in LPL. Until the mid eighties, these models had to be solved by a solver package on a mainframe computer. Therefore, the important task of the LPL was to produce the MPSX solver input file. The hardware of the new generation of PCs as well as the solver software packages such as XA, CPLEX, and others, allow us now to solve and manipulate the mentioned models locally on the PC. LPL has, therefore, been extended to a point, where different modeling tasks (model formulation, solving and report writing) are supported. Our practical experiences are, that a typical modeling-cycle (modification – resolving – reporting results) has been reduced from several hours or even days to several minutes. The soccer model with 2700 0–1 variables (represented in the Appendix A) was generated under MS/DOS on a 80386 by LPL in about 2 minutes and solved by XA in 30 minutes, six year ago. Actually (June 1998), the same model is generated in 5

seconds and solved in one minute by a new XA under NT4.0 on a Pentium 200Pro. By these figures one may estimated the progress made to manipulate and solve bigger models. All the more important become the modeling tools that allow the modeler to build, to manipulate, to modify, and to document the model.

The LPL compiler has been implemented using Turbo Pascal and Delphi 3 from Borland under MS/DOS and Windows 95/NT. The version 4.30 is actually available at the LPL-site: **<ftp://ftp-iiuf.unifr.ch/pub/lpl/>**, completely with a documentation and model examples.

## APPENDIX A

The following model (SOCCER.LPL) was written in a draft form by J. Byer the developer of XA. I have adapted this model for use in this paper. The model is a simple assignment problem which assigns 180 players to 15 teams. The only variable is *Work* that is indexed over the players  $p$  and the teams  $t$ . It declares a total of 2700 ( $=180*15$ ) single 0–1 variables.  $work(p,t)$  has only the values 1 or 0, depending on whether the player  $p$  is assigned to team  $t$ .

```

MODEL Soccer "An assignment problem of 180 player to 15 teams (0-1 IP-
problem)";
  (* Ref: BYER James, Sunset Software, personal communications *)

OPTION randSeed := 1;

SET
  t          "the list of teams";
  p          "the list of players";
  must_be_in{p,t} "player p must be in team t";
  reject_from{p,t} "player p is rejected from a team t";
  t_groups{p,p} "groups of players who must be together in a team";
  n_groups{p,p} "groups of players who must never be together";

PARAMETER Skill{p} "skill of player p";
          Age{p}   "age of player p";

BINARY VARIABLE work{p,t} "=1 if player p is in team t else =0";

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

```

```

(*----- data -----*)
SET
t = /T1 T2 T3 T4 T5 T6 T7 T8 T9 T10 T11 T12 T13 T14 T15/;
p = /1:180/;
must_be_in{p,t} = / 1 T2 , 2 T6 , 34 T7 /;
reject_from{p,t} = / 10 T1 , 20 T2 ,
                    [166,*] T1 T3 T4 T5 T6 T7 T8 T9 ,
                    [64,*] T1 T12 /;
t_groups{p,p} = / 2 3 , 112 76 , 89 9 , 34 135 ,
                 [4,*] 35 47 81 98 /;
n_groups{p,p} = / 21 22 , 55 56 ,
                 [11,*] 35 45 56 67 78 89 90 21 /;

PARAMETER Skill{p} = trunc(rnd(3,8));
          Age{p} = trunc(rnd(10,12));

(* check that a player p can be only in one team *)
CHECK this{p}: sum{t | must_be_in}1 <= 1;
(*----- end data -----*)

MAXIMIZE obj;
WRITE
"          Results per player
*****

player Skill Age in Team
-----
$$$$$  ##  ##  $$$$

          Results per team
*****

team skill age Av.Skill Av.Age
-----
$$$$$ ##### ### ###.### ###.###

          Player to Team assignment
*****
team | players in the team
-----
$$$$ | $$$$
"
: ROW{p} (p , Skill , Age , COL{t|work} t) ,
          ROW{t} (t , SUM{p|work} Skill , SUM{p|work} Age ,
                  (SUM{p|work} Skill)/12 , (SUM{p|work} Age)/12 ) ,
          ROW{t} (t , COL{p|work} p);
END

```

**REFERENCES**

- BISSCHOP J., [1998], AIMMS, The Modeling System, Paragon Decision Technology B.V.
- BROOKE A., KENDRICK D., and MEERAUS A. [1988], GAMS, A User's Guide, The Scientific Press.
- FOURER R., GAY D.M., KERNIGHAN B.W., [1993], AMPL, A Modeling Language For Mathematical Programming, The Scientific Press, San Francisco.
- HÜRLIMANN T., [1998], Reference Manual for the LPL Modeling Language (Version 4.30), Institute of Informatics, Working Paper, June, Fribourg.
- HÜRLIMANN T., [1998b], Hierarchical Index-sets in Modeling Languages, Institute of Informatics, Working Paper, March, Fribourg.
- HÜRLIMANN T., [1998g], The LPO-file, an interface of the LPL modeling language to an arbitrary solver, Institute of Informatics, Working Paper, June, Fribourg.
- HÜRLIMANN T., [1998h], Linking the Nonlinear Solver CFSQP to the Modeling Language LPL, Institute of Informatics, Working Paper, June, Fribourg (this paper).
- HÜRLIMANN T., KOHLAS J., [1988], LPL: A Structured Language for Linear Modeling, OR Spectrum, Vol.10, p.55–63, Springer Verlag.
- SHAPIRO J.F. [1988], Stochastic Programming Models for Dedicated Portfolio Selection, NATO ASI Series, Vol. F48, Mathematical Models for Decision Support, Edited by G. Mitra, Springer Verlag, Berlin, S.587-611.
- SCHRAGE L., [1998], Optimization modeling with LINGO, Lindo Systems, Inc., (see WWW at: <http://www.lindo.com>).