LPL : A Mathematical Modelling Language

Tony Hürlimann
tony.huerlimann@unifr.ch
Departement of Informatics
University of Fribourg, Switzerland

December 8, 2011

Abstract

This paper describes the version 5.40 of the modelling language, named LPL (*Linear Programming Language* or *Logic Programming Language* or *Literate Programming Language*). It may be used to build, modify and document (linear and non-linear) mathematical and logical models. The LPL language has been successfully applied to formulate large models. The available LPL compiler translates LPL programs to the input code of a 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.

Keywords: Mathematical Modeling, Computer Languages, linear and non-linear optimization.

Contents

1 Introduction **2**

2 Fixed Income Portfolio (portfol[1]) **3**

3 Assign Players to Teams (soccer[1]) **6**

4 Book Acquisitions (library[1]) **15**

5 A Brief Overview of the LPL Language **20**

5.1 Index-sets 20
5.2 Numerical Data 21
5.3 Variables 22
5.4 Constraints 22
5.5 The Solver 23
5.6 Reports 24
5.7 Executable Statements 24

6 The Use of Indices **24**

7 Algebraic and Logical Expressions **26**

7.1 Constraints 27
7.2 Evaluating Tables 28
7.3 Sparse Tables 29

8 The Report and Input Generator **31**

8.1 Write Tables of Expressions 31
8.2 Writing with User-defined Masks 31
8.3 The Input Generator 33

9 Conclusion **33**

1 Introduction

This paper presents the last of the modelling language LPL. A first version of LPL was developed and implemented 15 years ago and was restricted to linear models (LPs). 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, string manipulation is also possible in a restricted way, goal and multi-stage programming are also supported to some extent, an open interface to most LP/MIP and non-linear solvers has been added, and finally, it has been extended to a real programming language. Many more modifications and improvements have taken place, but will not be mentioned here. Data used by the model can be in various format or read/write from/to databases.

Different modeling languages have been developed. Links are given at the LPL-Site (www.virtual-optima.com).

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

1. A simple syntax of models with indexed expressions close to the mathematical notation, and directly applicable for documentation,

2. Formulation of both small and large LP's with optional separation of the data from the model structure,

3. Availability of a powerful index mechanism, making model structuring very flexible.

4. An innovative and high-level Input and Report Generator, including connection to a database,

5. Intermediate indexed expression evaluation (much like matrix manipulation),

6. Tools for debugging the model (e.g. explicit equation listing),

7. Built-in text editor to enter the LPL model,

8. Fast production of the MPS file and other output-files, such as a TEX file which generates a embellished and documented version of the model. Generation of a SQL-script to create a database from the LPL-code specifications,

9. Open interface to most LP/MIP and non-linear solver packages. This means that any linear or non-linear solver can be called from within

the LPL code. The user can configure the communication between LPL and the solver,

10. Furthermore, LPL is a complete programming language.

This paper is organized as follows: Three simple examples demonstrate the overall structure of a model coded in LPL. The basic syntax of LPL is presented in section 5. Section 6 and 7 examine the indices and expressions, the most important constructs of LPL. Some aspects of the Report and Input Generator are discussed in section 8.

2 Fixed Income Portfolio (portfol[1])

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? (The problem is from [5]).

The model can be formulated as follows: *Explain*

```
MODEL Portfol "Fixed Income Portfolio (deterministic)";
  SET
    j      "A set of bonds or investment types";
    t      "A set of time periods";
  PARAMETER
    c{j}   "Current market price for bond j";
    f{j,t} "Cash flow produced by bond j in period t";
    q{j}   "Conditional minimum purchase quantity of bond j";
    Q{j}   "Maximum allowable purchase of bond j";
    a{t}   "Reinvestment rate (return at the end of period t";
    L{t}   "Cash available in period t";
  VARIABLE
    x{j}        "Quantity of bond purchased";
    s{t}        "Cash surplus at the end of a period";
    BINARY d{j} "d=1, if bond is selected else d=0";
  CONSTRAINT
    CashBal{t}: SUM{j} f*x + a*s[t-1] - s = L
      "Coupon return + cash surplus from t-1 = liabilities";
    C{j}: q*d <= x <= Q*d
```

```
      "Either x=0 or q<=x<=Q";
  MINIMIZE Invest: SUM{j} c*x + s[1]
    "Sum of purchased bonds plus initial cash";
END
```

This formulation contains different sections: Index-sets, numerical data-tables, variables (the unknowns),different constraints, and a minimization function. Note that this formulation 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. This can be formulated be another model as follows:

```
  MODEL data "define data here";
    INTEGER PARAMETER TMAX := 50;;
    j:=/A1 A2 A3 A4 A5/;
    c{j}:=[200 230 400 100 240];
    q{j}:=[ 10   .  20  15  20];
    Q{j}:=[500 700 700 800 900];
    t:=/T1 T2 T3 T4/;
    L{t}:=[ .   1200 1400  500];
    a{t}:=[ .9    .9   .9   .8];
    f{j,t}:=[. 4 4 4 , . 5.5 5 4 , . 5 3 6 , . 6 6 . , . 4 5 6];
  END
```

The data could also be read from a text file using the following data model instead:

```
  MODEL data;
    PARAMETER TMAX := 50;
    READ FROM 'portfol.dat' '%1:Table';
    READ '%1': ROW{j} (j,c,q,Q);
    READ '%2': ROW{t} (t,L,a);
    READ '%3': COL{t} t, ROW{j} (j, COL{t} f);
  END
```

The data must be displayed in a predefined format within the file `portfol.dat` as follows:

```
Table 1: j     c    q     Q
        A1   200   10   500
        A2   230    .   700
        A3   400   20   700
        A4   100   15   800
        A5   240   20   900
Table 2: t      L      a
        T1        .    90
        T2     1200    90
        T3     1400    80
        T4      500    80
Table 3 : f
         T1  T2   T3   T4
   A1     .  4    4    4
   A2     .  5.5  5    4
   A3     .  5    3    6
   A4     .  6    6    .
   A5     .  4    5    6
```

We shall see later, how this file is read in.

The different sections of the model structure: index-sets, data, variables, constraints, and the minimizing function are headed by the reserved words SET, PARAMETER, VARIABLE, CONSTRAINT, and MINIMIZE. 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 and can be used later on the documentation sheets. The other comments can also be used to document the source code of the 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 $\sum$ 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}`.

Solution

Running this model generates an output file below, giving an optimal solution to this problem. The optimal solution says to buy 272.36 units of bond type A4. We shall need an total investment of 27236.84 to buy these bonds, and the return flows over the time periods will guaranty that the liabilities can be fulfilled at each period. Cash surplus must be hold in the second and third period. You can solve the model over the Internet, try portfol[1][1].

[1] `http://diuflx71.unifr.ch/lpl/Solver.jsp?name=/portfol`

3 Assign Players to Teams (soccer[1])

Problem: Assign 180 players to 15 teams for a soccer tournament in such a way that the following conditions are fulfilled (The problem is from [3]):

1. A player must be only in one team.
2. The total player count per team must be 12.
3. The total skill level of a team must be at least 59.
4. The total team age must be at least 124.
5. Certain players must be in certain teams.
6. Certain players are rejected from certain teams.
7. Certain players must be together in the same team.
8. Certain players must not be together in the same team.

You can download and solve the model through Internet: see soccer[1][2].

Modeling Steps using LPL [2]

All we need to know is whether a player p is in team t or not. Hence, we first introduce the two sets $p \in P$ (players) and $t \in T$ (teams).

1. Now we introduce a binary variable x for each (p, t)-combination, with the meaning $x_{p,t} = 1$ if player p is in team t, and $x_{p,t} = 0$ if player p is not in team t.
2. We also add lists that map player to teams, exclude player from teams, collect players who must be together or must not be together.
3. To formulate the 8 requirements is now straightforward. For example, to formulate requirement 1 that a player must only be in one team can be coded as follows: For each player p we have $\sum_t x_{p,t} = 1$.
4. The last two requirements are a little bit harder to find. If two player $i \in P$ and $j \in P$ must be in the same team then $x_{i,t} = x_{j,t}$, $t \in \{1 \ldots 15\}$. Why? Suppose $x_{i,t} = 0$ for a particular team t, that means player i is not in team t, then $x_{j,t} = 0$ too, that is player j is not in

[2] `http://diuflx71.unifr.ch/lpl/Solver.jsp?name=/soccer`

team t as well. Suppose on the contrary $x_{i,t} = 1$ for a particular team t, that means player i is in team t, then $x_{j,t} = 1$ too, that is player j is also in team t. Hence, both are in the same team if $x_{i,t} = x_{j,t}$. We could also formulate this as a logical constraint:

$$x_{i,t} \leftrightarrow x_{j,t}, \qquad \forall_{t,i,j|(i,j)\in tGroups}$$

5. To require that two players i and j are not in the same team, we only need to exclude that both variables $x_{i,t}$ and $x_{j,t}$ are true for each team t. That is $x_{i,t} + x_{j,t} \leq 1$. One could also formulate this a s a logical constraint:

$$x_{i,t} \rightarrow \overline{x_{j,t}}, \qquad \forall_{t,i,j|(i,j)\in nGroups}$$

Note the asymmetry of the last two constraints. For the first we have the requirement: If player i and j must be together, then if player i is not in a team t then player j must not be in team t neither. This is not the case for two players i and k, which must not be together: if player i is not in team t, then player k can or cannot be in team t.

The complete formulation is as follows:

```
MODEL Soccer "Assign Player to Teams";
  SET
    t                 "the list of teams";
    p                 "the list of players";
    mustBeIn{p,t}     "player p must be in team t";
    rejectFrom{p,t}   "player p is rejected from a team t";
    tGroups{p,p}      "players who must be together in a team";
    nGroups{p,p}      "players who must never be together";
  PARAMETER Skill{p} "skill of player p";
            Age{p}   "age of player p";
  BINARY VARIABLE x{p,t} "=1 if player p is in team t else =0";
  CONSTRAINT
    Bounds{p}: SUM{t} x = 1
      "player p must be only in one team";
    Heads{t}: SUM{p} x = 12
      "total player count per team t must be 12";
    SkillLevel{t}: SUM{p} Skill*x >= 59
      "total skill level of team t must be at least 59";
    TeamAge{t}: SUM{p} Age*x >= 124
      "total team age of team t must be at least 124";
```

```
    Must{i=mustBeIn}: x[i] = 1
      "player p must be in team t";
    Reject{i=rejectFrom}: x[i] = 0
      "player p is rejected from a team t";
    Same{t,tGroups[i,j]}: x[i,t] - x[j,t] = 0
      "players which must be in the same team";
    Never{t,nGroups[i,j]}: x[i,t] + x[j,t] <= 1
      "players which must not be in the same team";
  MAXIMIZE obj: SUM{p,t} x "maximize the assignment";
END
```

A particular feature of this formulation is its relations defined: `mustBeIn` or `rejectFrom`. These relations are tuple of their corresponding basic sets. There relations then can be used as sets in the subsequent part of the model specification.

Another particularity is the `WRITE` statement. It defines a mask which has to be filled up. The solution output (the file soccer.nom) shows the result of this instruction.

A further point is interesting: The relation `nGroups` is only a partial list. To complete it, one must construct the transitive closure of it (if a player A and B must not be in the same team and player B and C must not be in the same team then we can deduce that players A and C also must not be in the same team). The code is given in the data model as follows:

```
  MODEL data;
    t  := /T1 T2 T3 T4 T5 T6 T7 T8 T9 T10 T11 T12 T13 T14 T15/;
    p  := /1:180/;
    mustBeIn{p,t}    := / 1 T2 , 2 T6 , 34 T7/;
    rejectFrom{p,t} := / 10 T1, 20 T2, 64 T1, 64 T12,
      166 T1, 166 T3, 166 T4, 166 T5, 166 T6, 166 T7, 166 T8, 166 T9/;
    tGroups{p,p} := / 2 3 , 112  76 , 89 9 , 34 135 ,
                          4 35, 4 47, 4 81, 4 98 /;
    nGroups{p,p} := / 21 22 , 55 56 , 1 9 , 1 10,
       11 35, 11 45, 11 56, 11 67, 11 78, 11 89, 11 90, 11 21/;
    Skill{p} := trunc(rnd(3,9));
    Age{p}   := trunc(rnd(10,12));

    -- must calculate the transitive closure
    parameter dummy := {p0=p,p1=p,p2=p|p0<p1<p2}
```

```
    (IF(nGroups[p0,p1] and nGroups[p0,p2],nGroups[p1,p2]:=1),
     IF(nGroups[p0,p1] and nGroups[p1,p2],nGroups[p0,p2]:=1),
     IF(nGroups[p0,p2] and nGroups[p1,p2],nGroups[p0,p1]:=1));
END
```

Solution

This example is a pure 0-1 problem with 2700 binary variables and 1230 constraints. One may wonder why it was so easily solvable. If the problem has a solution, then certainly all 180 players must be assigned, therefore, the solution has a trivial upper and lower bound of 180. Since this is the optimal value (supposing there is a feasible solution), it is easy to check for optimality. If, on the other hand, the model had no feasible solution, it could be probably extremely hard to prove this.
A solution is as follows:

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

This table is generated by the instruction (this will be explained later):

```
  WRITE 'team | players in the team
------------------------------------------------------
%4s | %4s\n'  : ROW{t} (t , COL{p|x} p);
```

The same solution is given graphically in Figure 1 and 2 (This graphs have also been generated by a LPL code). The player are numbered vertically and the the teams horizontally.

Green circles at the left within these black rectangles indicate that indeed this is a `Must` constraint. Hence, one can verify on the fly that *all* green point are not outside a black rectangle, so these constraints are all fulfilled (we did not expect anything else, if it would be otherwise, the model constraints were wrong!).

White circles at the right within these black rectangles indicate that indeed all `Same` constraints are fulfilled. Example, `tGroups` says that player 2 and 3 must be in the same team. Looking at the graph, we can verify this fact, they are in team T6 together.

Red circles must all be outside these black rectangles, since they model the `rejectFrom` relation. These position are forbidden. We can verify at a glance that all red circles are outside (fulfilling the `Reject` constraints).

Yellow circles must also be all outside the black rectangles, otherwise they would violate the `Never` constraint.

It can be easily verified in the graph of Figure 1 and 2 that all other constraints also hold. So, the constraint that a player can only play in one single team (the `Bounds` constraint, can be verified by looking at all number in the right most column, these numbers are all 1, saying that a particular player is only in one team.

The constraint that all teams must have 12 players (`Heads` constraints) can be verified by the third data row from the bottom, these numbers are all 12; the constraint `SkillLevel` by the second data row from the bottom, and finally the last data row shows that the constraints `TeamAge` is fulfilled.

This example shows that sometimes with a relatively few number of instructions the whole solution can be visualized and verified easily. This is very important in such problems. If a solution is not "acceptable", then one can easily generate variants which then can again be visualized and rejected or accepted on a quick "vision test".

Question 1 *(Answer to 1)*

---

1. *What is the solution, if we require in addition that player* `3` *must be in team* `T7`*?*
2. *In addition to all requirements we want to balance the team skills as much as possible. How would we formulate this?*

Answer 1 *(Question of 1)*

---

1. *The table* `mustBeIn` *changes to* `mustBeIn{p,t}= /... , 3 T7/;` *Solving the problem reveals that it is infeasible. Inspecting the data shows why:* `mustBeIn` *requires that player* `2` *is in team* `T6` *and player* `3` *in team* `T7`. *This contradicts our requirement that player* `2` *and* `3` *must be together in the same team (see* `tGroups`*).*

2. *Different approaches exist! A straightforward is as follows: The previous maximizing function is transform to a constraint as follows:*

```
Assign: SUM{p,t} x=180;
```

(since we know that all players can be assigned). Then we introduce a new objective function. That is to minimize the sum of all positive difference in team skill between the teams. However this leads to additional constraints and the model is no longer solvable.

Another solution is as follows: Keep the original model and raise the right- hand-side (RHS) constant of the constraint `SkillLevel` *as much as possible: from 59 to 60, then 61, 62, 63, 64. At 65, the model becomes infeasible. At each higher value of the team skill the skills are more balanced. This is because teams with high team skill must move skilled player to teams that have low team skill.*

At 64, the highest team skill is only 67. We now force a upper bound on the team skill of 65 to see if the balance can be augmented. Indeed, the solution now displayed teams only with a team skill of 64 and 65, which is optimal. Hence, in the original model, we replaced the constraint:

```
SkillLevel{t}: SUM{p} Skill*x >= 64;
```

by the new constraint:

```
SkillLevel{t}: 64<= SUM{p} Skill*x <= 65;
```

This is a efficient way to solve the problem. The disadvantage is that the model must be solved several times and we must "try" lower and

upper bounds. Of course, we could model a loop to find automatically the lower and upper bounds.

Still another way – that avoids the loop in the previous solution – is to add two integer variables as:

```
INTEGER VARIABLE xx [64-4,64]; yy[65,65+4];
```

for the unknown lower and upper bound in the previous model. We know from the data that upper bound of xx cannot be larger than 64 (since the average level of team skill is 64.8667) and the lower bound of yy cannot be smaller than 65. It is important to bound xx and yy as much as possible. Then we add the constraint and the minimizing function as follows:

```
Assign: SUM{p,t} x=180;
MINIMIZE obj: yy-xx;
```

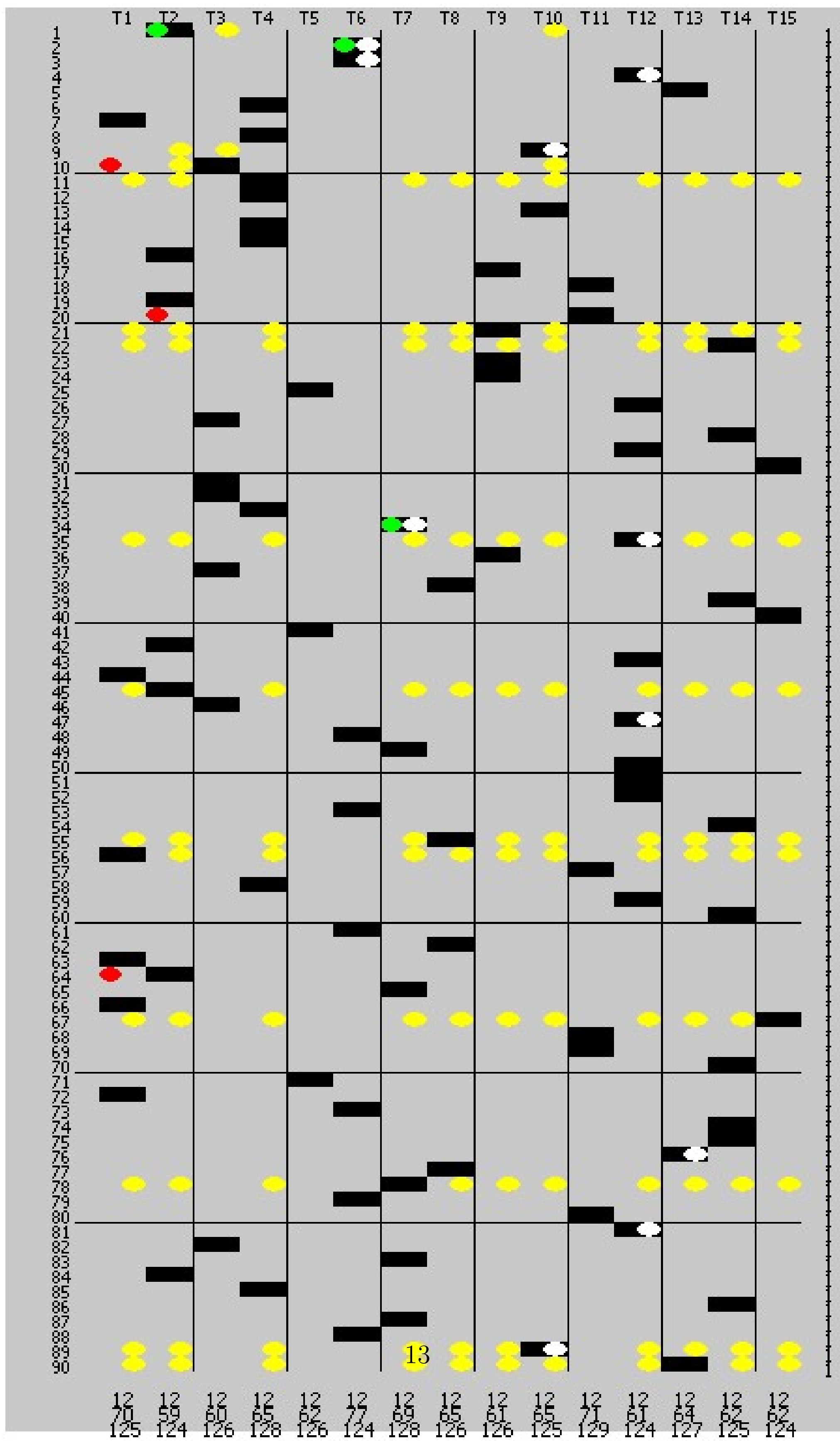

Figure 1: The solution (player 1–90)

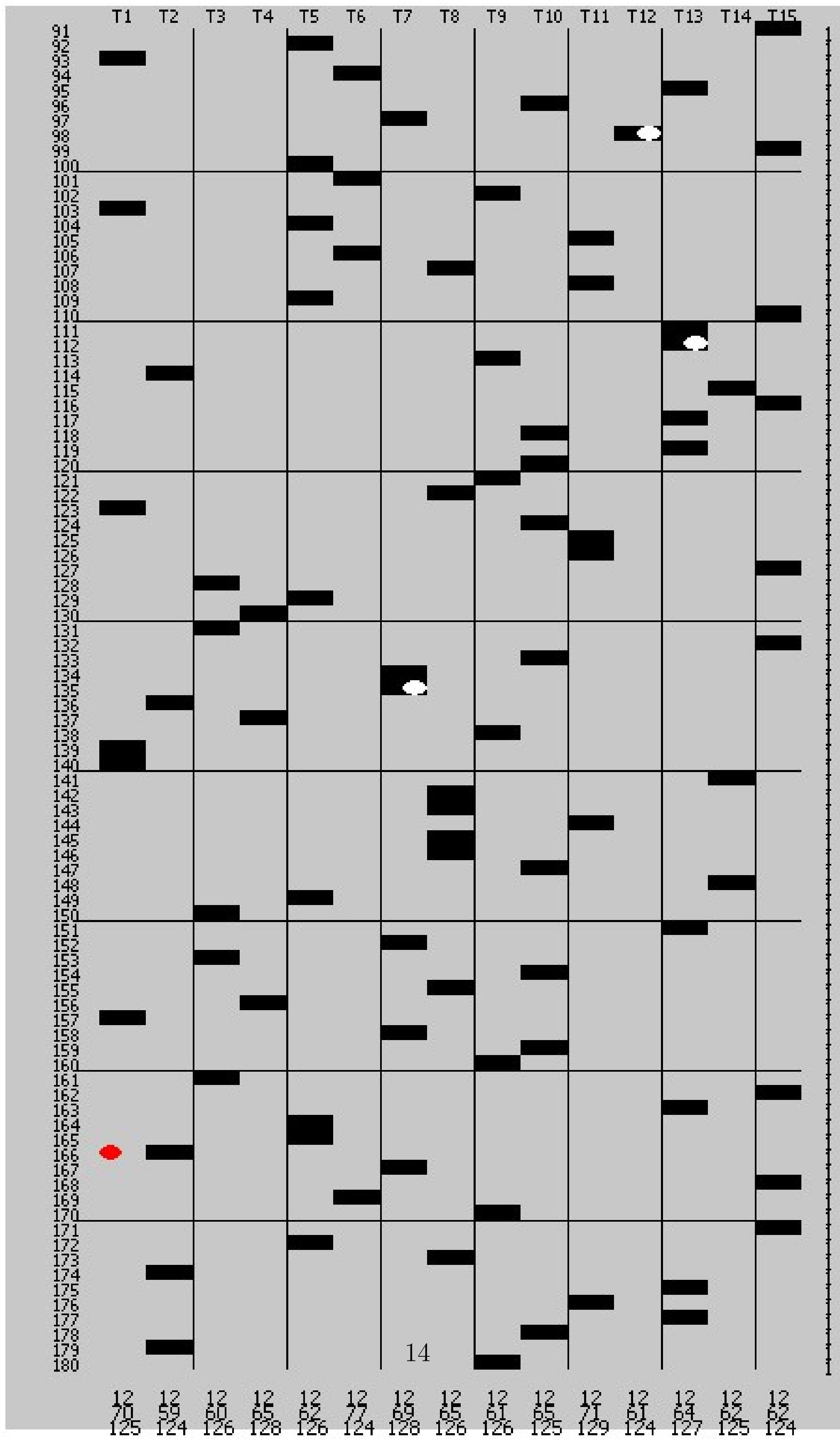

Figure 2: The solution (player 91–180)

4 Book Acquisitions (library[1])

Problem: The budget of an academic library has to be allocated to different book types and faculties. Various conflicting goals should be achieved, such as the global expenditures that should be in a specified range, a given percentage should go into periodicals, at least a particular number of titles should be purchased, etc. How should the budget be allocated while minimizing deviations from all these goals? (See [4].) The 9 goals we like to achieve are:

1. Acquire between 7500 and 10700 titles.
2. Observe the total budget of 2000000.
3. Limit periodical expenditures to 60% of the total acquisitions.
4. Limit periodical acquisitions to the level supported for a five-year period.
5. Allocate titles by subject according to circulation data.
6. Allocate titles by subject according to enrolment data
7. Limit research acquisitions to 15% of the total acq. and allocate on the basis of departmental research productivity.
8. Limit retrospective acquisitions to 5% of total acquisitions and allocate on the basis of retrospective subject needs.
9. Meet desired subject acquisition ranges.

You can download and solve the model through Internet: see library[1][3].

Modeling Steps using LPL [2]

A mathematical formulation of this problem is given below. All goals are defined as so called soft-constraints, that is, the constraint should hold if possible. If it is not possible, then a deviation should be minimal. To achieve this, positive and a negative slack variables are introduced for every constraint. For example, `sum{i,j} x[i,j] >= 7500` is formulated as `sum{i,j} x[i,j] + n = 7500`, where `n` is a positive variable. Hence, n "measures" the negative deviation from 7500. These kind of variables are

[3] `http://diuflx71.unifr.ch/lpl/Solver.jsp?name=/library`

minimized in an objective function. Typically, the objective function sums all or a subset of the slack variables each multiplied by a weight (the more important the goal, the higher the weight).

The model formulation is as follows:

```
MODEL LIBRARY "Book Acquisitions";
  SET
    i := /Books, Periodicals/  "Book types";
    j := /Humanities, SocialScience, Sciences, Education,
          Interdisciplinary/ "Faculties";
  PARAMETER
    aC{i,j}  "Average cost of a title";
    pC{j}    "Projected average costs for periodicals for 5 years";
    cir{i,j} "Circulation data";
    e{i,j}   "Enrolment perc. of all book and periods titles";
    pr{j}    "Research productivity";
    ret{j}   "Retrospective data";
    low{i,j} "Minimum percent level of acquisition";
    up{i,j}  "Maximum percent level of acquisition";
   VARIABLE x{i,j}  "Number of titles assigned";
    Ng1; Pg1a; Pg2; Ng2; Pg3; Ng3; Pg4; Ng4; Pg5; Ng5; Pg6{i,j};
    Ng6{i,j}; Pg7; Ng7; Pg8; Ng8; Pg9{i,j}; Ng9{i,j}; Pg9a{i,j};
    Ng9a{i,j};
  CONSTRAINT
    g1 : SUM{i,j} x + Ng1 >= 7500;              -- goal 1
    g1a: SUM{i,j} x - Pg1a <= 10500;            -- goal 1a
    g2 : SUM{i,j} aC*x +Ng2-Pg2 = 200000;       -- goal 2
    g3 : SUM{j} aC[2,j]*x[2,j] +Ng3-Pg3
          = 0.6*(SUM{j} aC[2,j]*x[2,j]);        -- goal 3
    g4 : SUM{j} pC*x[2,j] +Ng4-Pg4 = 193261;    -- goal 4
    g5 : SUM{i,j} cir*x +Ng5-Pg5 = 550;         -- goal 5
    g6{i,j|e}: x[i,j] +Ng6-Pg6 = e*(SUM{j|e} x[i,j]);      -- goal 6
    g7 : SUM{j|pr} pr*x[1,j] +Ng7-Pg7 = 0.15*(SUM{i,j} x); -- goal 7
    g8 : SUM{j} ret*x[1,j]+Ng8-Pg8 = 0.05*(SUM{i,j} x);    -- goal 8
    g9{i,j}: x +Ng9-Pg9 = low*(SUM{i,j} x);     -- goal 9
    g9a{i,j}: x  +Ng9a-Pg9a = up*(SUM{i,j} x);  -- goal 9a
  MINIMIZE  M1:  .... --- weighted sum;
END
```

The method we use is called *goal programming*. Goal programming is a

simple technique to handle several (maybe conflicting) goals in a mathematical way. Here is a simple example.

Suppose that three (conflicting) requirements (goals) are as follows: (1) Do not exceed the budget. (2) At least 70% of the budget must go to A. (3) Exactly 50% must go to B. However, the three goals cannot be attained at the same time. Of course in this simple example this fact is evident and straightforward. In practice, things are more complicated and it is not evident – especially when the goals are formulated by different actors – that the goals conflict. A mathematical formulation would then simply generate a unsolvable model and nothing could be deduces from such a model.

We have to make compromises in attaining the goals. A solution is – as already shown – to introduce additional variables which 'absorb' the deviations from the goals. A weighted sum of these variables can then be minimized to attain the goals "as much as possible".

Instead of minimizing a weighted sum, one could also rank the goals according to their importance. Then the deviations in the order of the ranked list of goals are minimized. After each optimization the attained goal are fixed, that is, the values of the minimized deviations variables are fixed in the next round of minimization. This method was used in this application: First, `Ng1+Pg1a` is minimized, that is the negative deviation from goal `g1` and the positive deviation from goal `g1a`. Next the positive deviation of goal `g2` (the positive deviation of goal `g2`) is minimized, and so on. The objective function above is replaced by the instructions:

```
MINIMIZE  M1: Ng1+Pg1a;
Freeze(Ng1,Pg1a);
MINIMIZE  M2: Pg2;
Freeze(Pg2);
MINIMIZE  M3: SUM{i,j} (Ng9+Pg9a);
Freeze(Ng9,Pg9a);
MINIMIZE  M4: Ng5;
Freeze(Ng5);
MINIMIZE  M5: SUM{i,j|e} IF(j<=2,Pg6,Ng6);
Freeze(Pg6,Ng6);
MINIMIZE  M6: Pg7+Pg8;
Freeze(Pg7,Pg8);
MINIMIZE  M7: Pg3+Pg4;
Freeze(Pg3,Pg4);
```

The optimal budget allocation for the following data **Solution**

```
MODEL data;
  aC{i,j}:= [13.01, 12.55, 19.32, 10.53, 13.10, 37.64,
             23.12, 114.0, 24.18, 35.0];
  pC{j}:= [75.71, 37.92, 229.29, 55.32, 58.98];
  cir{i,j} := [0.028, 0.028, 0.033, 0.066, 0.004, 0.033,
               0.099, 0.075, 0.209, 0.037];
  e{i,j} := [0.278, 0.273, 0.186, 0.263, 0, 0.278,
             0.273, 0.186, 0.263, 0];
  pr{j}  := [0.175, 0.450, 0.600, 0.210, 0.000];
  ret{j} := [0.252, 0.475, 0.042, 0.185, 0.046];
  low{i,j}:= [0.15, 0.25, 0.05, 0.15, 0.05, 0.03, 0.10,
              0.03, 0.05, 0.10];
  up{i,j} := [0.25, 0.35, 0.10, 0.25, 0.10, 0.08, 0.15,
              0.05, 0.10, 0.15];
END
```

is as follows:

	Books	Periodicals	SUM
Humanities	1447	289	1736
SocialScience	2412	965	3377
Sciences	482	295	777
Education	1549	761	2310
Interdisciplinary	482	965	1447
SUM	6372	3275	9647

Table 1: TEXT

	Books	Periodicals	SUM
Humanities	18826.05	10893.35	29719.40
SocialScience	30267.35	22303.78	52571.13
Sciences	9318.97	33598.39	42917.36
Education	16312.47	18396.50	34708.97
Interdisciplinary	6318.76	33764.37	40083.13
SUM	81043.60	118956.40	200000.00

Table 2: Budget Allocation (Amount of money)

5 A Brief Overview of the LPL Language

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

5.1 Index-sets

An index-set is an ordered collection of objects. They are called *elements*. The set of bonds j in the first example is such a set. A set is declared in LPL as follows:

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

The elements themselves of this set are not defined at this place, they are part of the model data defined in the file `portfol.dat` or in a `MODEL data ...` specification. The modeler can also declare a set and assign its elements at the same place within the model. The set j could have been defined then 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,...,5 might also be replaced by more meaningful names as follows:

```
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 j.

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:

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

then an indexed set $MustBeIn$ can be specified as follows (as shown in the soccer example):

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

This tuple-list defines a relation *MustBeIn* between the players p and the teams t. It tells us, that player P1 must belong 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 would be to define subsets of players as follows:

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

*pla*1, for example, defines a subset of the set p containing the six elements (1, 45, 56, 67, 78, 122). Indexed sets may also be assigned by logical expressions. The union, the intersection and the difference of set *pla*1 and *pla*2 may be defined as

```
SET unionP{p}  := pla1 OR pla2;
SET intersectionP{p} := pla1 AND pla2;
SET differenceP{p} := pla1 AND ~pla2;  -- ~ being the not-operator
```

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

5.2 Numerical Data

Numerical data within the model are collected in parameters. Its declaration is headed by the reserved word `PARAMETER`. The simplest parameter consists of only one value:

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

$TMAX$ is declared as a parameter of type integer, but its value is not yet known. The declaration, however, tells us that $TMAX$ must be an integer value within the range $[0, 100]$.

Parameters can also be indexed, and their values are collected in tables. c_j (`c{j}` in LPL), for example declares a vector, which length is the cardinality of the set j. It declares a market price c for every bond j. The conditional minimum purchase quantity q and the allowable purchase Q are also indexed over j. (Note that LPL distinguishes lower and upper-case letter by default, therefore, q and Q are distinct identifiers.) The file `portfol.dat` defines a convenient way to define the set j together with the three parameters c, q, and Q. This is possible, because all three parameters 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, for example:

```
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. However, the most flexible way to get the data from external formats is to use the Input Generator. Indexed parameters can be two- three- or higher-dimensional. `f{j,t}`, for example, declares a two-dimensional parameters (a matrix), where the cash flow f is defined for every bond j within every time period t.

5.3 Variables

Variables have the same properties as parameters. 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 non-linear models as initial values from were 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 or Boolean. These options can be added to its declaration as in

```
INTEGER VARIABLE d{j}  [1,10];
```

The declaration of d declares a integer variable over set j with a lower bound 1 and an upper bound 10. If the reserved word `INTEGER` is replaced by `BINARY`, its values can only be the integers 0 or 1. These constraints 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. Hence, LPL generates a MIP-problem (mixed integer problem).

5.4 Constraints

The model constraints are declared in the `CONSTRAINT` statement. Each constraint begins with a name. The optimizing function can be included within the list. The constraint name is followed by a colon and an expression. Constraints can also be indexed like parameters and variables. The constraint

```
CONSTRAINT  C{j}: q*d <= x <= Q*d;
```

for example, is defined for every element of the set j. This generates as many single constraints as j has elements. Any algebraic or logical expression is allowed for defining constraints. The objective function also can be declared in the same way as a constraint:

```
MAXIMIZE  D : SUM{j} c[j]*x[j];
```

Summations begin with the reserved word SUM. The term

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

for example, sums the dot product cx over the set j. This corresponds to the mathematical notation of $\sum_j c_j x_j$. 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 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. Constraints can also be made inactive or the values of variables can be fixed using the keyword FREEZE:

```
VARIABLE x FREEZE := 10;
CONSTRAINT R FREEZE: a*x + b*y <= 19;
```

Inactive constraints are not send to the solver and fixed variables are treated as if they were parameters. Frozen entities, of course, does only make sense if in the course of actions they can be "unfrozen" using the keyword UNFREEZE. In the model *library.lpl*, the keyword FREEZE is used in a multistage (multi goal) optimization problem. Certain slack variables are minimized and then their values are fixed, another subset of slack variable is minimized and their values again fixed, etc.

5.5 The Solver

LPL has an integrated linear solver for small problems (up to 800x800 matrix problems). However, for any serious problem a commercial solver should be used that LPL can call automatically.[4] The interface between most available LP/MIP solvers and LPL can even be specified by the modeler. For non-linear solver an interface must be programmed. The command MINIMIZE or MAXIMIZE instructs the LPL compiler to generate an instanciated model and to send it to a specified solver while LPL running system is waiting. The interface specification to a solver are explained in detail in the reference manual of LPL. By default, the LPL's internal simplex solver is called, but other solver packages such as gurobi, CPLEX, MOPS, Xpress, Mosek, and Glpk work too. Gurobi and CPLEX can also be called as dynamic link library (DLL). LPL even integrates a heuristic solver to solve certain scheduling problems. Non-linear solvers must be added by the user.

[4]Their exist several freely available solvers in the Internet. One of the most powerful are lp_solve and GLPK. They are available at our Virtual-Optima site.

5.6 Reports

LPL not only allows one to formulate a complete model, but can also produce output. The reserved word `WRITE` is used to generate result reports. The simplest reports are generated using the following syntax:

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

which prints four tables in a predefined format (see `portfol.nom` file output). A more complex `WRITE` statement is found in the soccer model. It is explained below.

5.7 Executable Statements

An LPL model consists of declarative statements and executable statements. The executable statements may contain:

1. loop statements (FOR, WHILE)
2. conditional statement (IF ... THEN ... ELSE ...)
3. FREEZE/UNFREEZE statement (to fix/unfix variables or inactivate/activate constraints
4. EMPTY statement (to clear data)
5. MINIMIZE/MAXIMIZE/WRITE/READ statements
6. assignment statement (to re-assign data and sets)
7. model call statement (to execute another 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 somewhat greater detail in the subsequent sections. Most examples are from the model `soccer.lpl`.

6 The Use of Indices

The indices are used to define multi-dimensional entities, such as parameters, variables, constraints, or indexed sets. They are called *tables*.

```
PARAMETER a{i,j} := 1;
VARIABLE  x{i,j,k,l};
CONSTRAINT r{i} : SUM{j} x[i,j] = 1;
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 constraint-vector, and s is a two-dimensional indexed set. In the same way as in these declarations, the indices are also used for the different index-operators (`SUM, PROD,...`). These are operators, which iterate over index-sets, called index-operators.

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

The first expression above returns the sum of all values in the table `a[i,j]`, it is in fact the same as

$$\sum_{i,j} a_{i,j}$$

. The second expression returns the product of all `x[i,j,k,l]`. In the previous 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 over 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 an 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 previous 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,i,j}`. Therefore, a unique binding is not possible. Binding is very 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 in set j within the set k. (Remember, all index-sets in LPL are 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 non-empty 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. 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 error proned, LPL also offers a compiler switch, which restricts this practice (for details see the reference manual). 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 the element name is returned by the expression (as a string) not the position.

7 Algebraic and Logical Expressions

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

1. to define model constraints
2. to evaluate and to output intermediate expressions and tables
3. to declare sparse, indexed tables

Expressions are built using the usual mathematical notation with arithmetical (`+ - * / ^`) and logical (`and, or, ~`) operators, parameters, numbers and functions. They may also contain index-operators, which are iterated over index-sets. The sum-operator $\sum$, replaced by the reserved word `SUM`, is such an index-operator. LPL 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.

Parameters or variables within an expression return the corresponding value. If a parameter 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 as follows:

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

It is important to see, that LPL allows one to evaluate and to return indexed expressions and not only single values.

7.1 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 non-linear 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 constraint. Constraints containing only one variable are automatically transformed to bounds.

SIN LOG ...	math. functions
+ - # IN	unary operators
/ %	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

Table 3: Operators in LPL

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

Table 4: Some Functions in LPL

```
CONSTRAINT R: x + y = SUM{i} z[i];   -- defines a constraint R
CONSTRAINT B: a <= x+y <= b;         -- defines a range
CONSTRAINT L{i}: x[i] + y = 5;       -- defines a constraint 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 constraint S will be automatically simplified to $-3 * x - y = 0$, and in T the variable x is eliminated.

7.2 Evaluating Tables

Expressions are also used to produce intermediate results.

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

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
ARGMAX{i} a[i]	position i of the biggest a[i]
ARGMIN{i} a[i]	position i of the smallest a[i]
COL{i} ...	horizontal table-expander
ROW{i} ...	vertical table-expander

Table 5: Index-operators in LPL

```
PARAMETER d{i,j} := a[i,j] + b[i,j];
PARAMETER 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 (this is called a it definition in LPL). The fourth calculates and outputs a two-dimensional table.

7.3 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, j;                -- a list of locations (two names!)
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 (and j). 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 and shorter:

```
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 then 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<=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 constraint `Must` in the soccer model.

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

which would produce 2700 (=180x15) single restrictions without the condition $MustBeIn$, but actually produces only the following three constraints

```
Must1: Work[1,T2]=1
Must2: Work[2,T6]=1
Must3: Work[34,T7]=1
```

since $MustBeIn$ is defined as

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

which is an indexed set containing three tuples. Still a better method is to define the constraint `Must` as follows:

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

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

8 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 favored 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 a, b, c;          -- write all three tables
```

8.1 Write Tables of Expressions

LPL also allows the user 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    '%12f'  : a*b;     -- write a value with width 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.

8.2 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 the WRITE statement. The contents of the mask may be any string of characters. The characters %, however, have a special signification and is called place-holder. It replaced by the numerical or string expressions of a subsequent WRITE statement. Example:

```
WRITE
'  Month: %8s  :  %7.2f\%   %9.4f gallons\n'
:  '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. The soccer model contains such a mask write.

The mask (in model *soccer.lpl*) contains 11 place-holders. The first four are defined on line 5 as

```
%6s    %2d   %2d    %4s
```

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 10 of the mask as follows:

```
%5s  %5d  %3d  %7.3f  %7.3f\n
```

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 13 (the last line of the mask):

```
%4s | %4s\n"
```

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.

8.3 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 are organized in a plain text file (*portfol.dat*). The data in this file is read by the four READ statements, defined in the MODEL DATA submodel as following:

```
READ FROM 'portfol.dat' '%1:Table';
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 declares the filename of the input file and the read-block-delimiters, 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 'Table' string, beginning at a new line, is encountered. On each line, four tokens are read, if possible. The instruction `ROW{j}...` has two effects: 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 than four, 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 in "real-time". It still binds the passive indices of c, q, and Q, however. 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 highly readable pages of LPL code.

9 Conclusion

This paper is a brief – necessarily incomplete – introduction of the LPL modeling language. The Reference Manual gives a detailed description of the language. The

development of LPL was initially motivated by practical use. Different models with 8000 constraints, 20000 variables, and a matrix density of 70000 non-zero elements are under continuous use and development at the Departement of Informatics at the University of Fribourg. Most of them are formulated in LPL. 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 Cplex, Xpress, Glpk, 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 modelling tasks (model formulation, solving and report writing) are supported. The practical experiences are, that a typical modelling-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 was generated under MS/DOS on a 80386 by LPL in about 2 minutes and solved by XA in 30 minutes (in 1992). 2004, the same model is generated in 10 seconds and solved in 30 seconds by the CPLEX 6.5 under Windows XP. Actually, LPL uses 4secs to generate the model and Gurobi 2 takes 0.1 secs to solve the model! By these figures one may estimated the progress made to manipulate and solve bigger models. All the more important become the modelling tools that allow the modeler to build, to manipulate, to modify, and to document the model.

The LPL compiler has been implemented using Delphi 2006 from Inprise under Windows XP. It can be compiled also under freePascal (a freeware compiler: www.freePascal.com) A restricted version of LPL is available at the LPL-site free of charge. It comes with a complete documentation and many model examples at:

www.virtual-optima.com

References

[1] T. Hürlimann. Model Library. http://diuflx71.unifr.ch/lpl/mainmodel.html.

[2] T. Hürlimann. Reference Manual for the LPL Modelling Language. http://www.virtual-optima.com.

[3] Byer J. A Soccer problem. private communication, Sunset Software.

[4] Beilby M.H. and Mott T.H. Academic Library Acquisitions Allocation based on Multiple Collection Development Goals. *Compututation and Operations Research*, 10:4:335–343, 1983.

[5] J.F. Shapiro. *Stochastic Programming Models for Dedicated Portfolio Selection*, pages 587–611. NATO ASI Series Vol. F48, Springer-Verlag, 1988.