Thorsten Koch

Zimpl **User Guide**

ZIMPL
(Zuse Institute Mathematical Programming Language)

Thorsten Koch

11. March 2001

Abstract

ZIMPL is a little language to translate the mathematical model of a problem into a linear or (mixed-)integer mathematical program expressed in `lp` or `mps` file format which can be read by a LP or MIP solver.

May the source be with you, Luke!

1 Introduction

Most of the things in ZIMPL (and a lot more) can be found in the wonderful book about the modelling language AMPL from Robert Fourer, David N. Gay and Brian W. Kernighan [FGK93]. Indeed if not the guys at ILOG had needed more than three months just to tell me the price of AMPL for CPLEX I would probably use AMPL today. On the other hand, having the source code of a program has its atvantages. The possibility to run it regardless of architecture and operating system, the ability to modify it to suite the needs and not having to hassle with license managers may make a much less powerful program the better choice. And so ZIMPL came into being.

A linear program (LP) might look like this:

$$\begin{aligned} \min \quad & 2x + 3y \\ \text{subject to} \quad & x + y \leq 6 \\ & x, y \geq 0 \end{aligned}$$

The usual format to store the description of such a problem is `mps` invented by IBM [IBM97] long ago. Nearly all available LP and MIP solvers can read this format. While `mps` is a nice format to punch into a punch card and at least a reasonable format to read for a computer it is quite unreadable for humans.

```
NAME          ex1.mps
ROWS
 N  OBJECTIV
 L  c0
COLUMNS
    x         OBJECTIV          2
    x         c0                1
    y         OBJECTIV          3
    y         c0                1
RHS
    RHS       c0                6
ENDATA
```

Another possibility is the `lp` format [ILO00], which is more readable but is only supported by a few solvers.

```
Minimize
 cost:  +2 x +3 y
Subject to
 c0:   +1 x +1 y <= 6
End
```

But since each coefficient of the matrix A must be stated explicitly it is also not a desirable choice to develop a mathematical model.

Now, with ZIMPL it is possible to write this:

```
var x;
var y;
minimize cost: 2 * x + 3 * y;
subto c0: x + y <= 6;
```

and have it automatically translated into `mps` or `lp` format. While this looks not much different from what is in the `lp` format, the difference can be seen, if we use indexed variables. Here is an example. This is the LP

$$\min \quad 2x_1 + 3x_2 + 1.5x_3$$
$$\text{subject to} \quad \sum_{i=1}^{3} x_i \leq 6$$
$$x_i \geq 0$$

And this is how to tell it ZIMPL

```
set I       := { 1 to 3 };
param c[I] := <1> 2, <2> 3, <3> 1.5;
var   x[I] >= 0;
minimize value: sum <i> in I : c[i] * x[i];
subto     cons: sum <i> in I : x[i] <= 6;
```

2   Invocation

To run ZIMPL on the model given in the file `ex1.zpl` type the command

```
zimpl ex1.zpl
```

The general case is `zimpl [options] <input-files >`. It is possible to give more than one input file. They are read one after the other and then executed as if they were all in one big file. The result of running ZIMPL will be either an error message or two files.

The two files are the problem generated from the model in either `lp` or `mps` format with extension `.lp` or `.mps` and a file with the extension `.tbl` which lists all variable and constraint names used in the model and there corresponding name in the problem file. The reason for this name translation is that the `mps` format can only handle names up to eight characters long. And in LP format the length of the names is also restricted to 16 characters.

The following options are possible (only the first two are normally of interest):

-t *format*	Selects the output format. Can be either `lp` which is default or `mps`.
-o *name*	Sets the base-name for the output files.
	Defaults to the first input file striped of the extension.
-b	Enables bison debugging output.
-d	Enables zimpl debugging output.
-f	Enables flex debugging output.
-h	Prints a usage message.
-v	Enables verbose mode.

A typical invocation is for example:

```
zimpl -o hardone -t mps data.zpl model.zpl
```

This reads the files `data.zpl` and `model.zpl` and produces `hardone.mps` and `hardone.tbl`.

3   Format

Each ZPL-file consists of five types of statements: Sets, parameters, variables, an objective and constraints. Each statement ends with a semicolon (`;`). Everything from a number-sign (#) to the end of the line is treated as a comment and is ignored.
If a line starts with the word `include` followed by a filename in double quotation marks, this file is read instead of this line.

ZIMPL works on the lowest level with two types of data: Strings and numbers. A string is delimited by double quotation marks (`"`). A number can be given as 2, -6.5 or 5.234e-12. Wherever a number or string is required, it is also possible to give a

parameter of the right value type. Also numeric expressions are allowed instead of just a number. The precedence of numeric, boolean and set operators should be the usual one. If in doubt use parenthesis to be save.

Numeric expressions

The following functions are currently implemented: $\wedge$,+,-,*,/,mod,div. $\wedge$ stands for *to the power of*. mod is the modulo function and div gives the integer part of a division. ** can be used as synonym for $\wedge$. With min and max it is possible to find the minimum/maximum member of an one dimension set of numeric values.
The following is also a numeric expression:
if *boolean-expression* then *numeric-expression* else *numeric-expression* end.

Boolean expressions

For numbers $<, <=, ==, !=, >=, >$ are defined. For strings only $==$ and $!=$ are available. The expression *tuple* in *set-expression* can be used to test set membership of a tuple. Combinations of boolean-expressions with and, or and negation with not are possible.

Sets

The elements of a set are tuples. Each tuple of a set has the same number of components. The components are either numbers or strings. The type of the n-th component of each tuple must be the same. A tuple starts and ends with $<$ and $>$, resp. The components are separated by commas. If tuples are one-dimensional, it is possible to omit the braces in a list of elements, but then they must be omitted from all tuples of the set.

Set functions

A cross B	Cross product	$\{(x, y) \mid x \in A \wedge y \in B\}$
A union B	Union	$\{x \mid x \in A \vee x \in B\}$
A inter B	Intersection	$\{x \mid x \in A \wedge x \in B\}$
A without B	Difference	$\{x \mid x \in A \wedge x \notin B\}$
A symdiff B	Symmetric difference	$\{x \mid (x \in A \wedge x \notin B) \vee (x \in B \wedge x \notin A)\}$
$\{$n to m *by s*$\}$	Generate, (default $s = 1$)	$\{x \mid x = n + is \le m, i \in \mathbb{N}_0\}$

It is possible to write * instead of cross, + instead of union, \ or - instead of without and .. instead of to.

Examples

```
set A := { 1, 2, 3 };
set B := { "hi", "ha", "ho" };
set C := { <1,2,"x">, <6,5,"y">, <787,12.6,"oh"> };
set D := A cross B;
set E := { 6 to 9 } union A without { <2>, <3> };
```

Parameters

Parameters can be declared with or without an indexing set. Without indexing the parameter is just one value, which is either a number or a string. For indexed parameters there is one value for each member of the set. There are not much uses for string valued parameters, so normally they are all numbers.

Parameters are declared in the following way: The keyword param is followed by the name of the parameter optionally followed by the indexing set. Then after the assignment sign comes a list of pairs. The first element of each pair is a tuple from the index set, the second element is the value of the parameter for this index.

Examples

```
param q := 5;
param u[A] := <1> 17, <2> 29, <3> 12;
param w[C] := <1,2,"x"> 1/2, <6,5,"y"> 2/3;
```

In the example, no value is given for index <787,12.6,"oh"> of parameter w, that is assignments need not to be complete. This is correct as long as it is never referenced.

Variables

Like parameters, variables can be indexed. A variable has to be one out of three possible types: Continuous (called *real*), binary or integer. The default is real. Variables may have lower and upper bounds. Defaults are zero as lower and infinity as upper bound. Binary variables are always bounded between zero and one. It is possible to compute the value of the lower or upper bounds depending on the index for the variable (see last declaration in the example).

Examples

```
var x1;
var x2 binary;
var y[A] real >= 2 <= 18;
var z[<a,b,c> in C] integer
    >= a * 10
    <= if b <= 3 then p[b] else 10 end;
```

Objective

There must be exactly one objective statement in a model. The objective can be either `minimize` or `maximize`. Following the keyword is a name, a colon (`:`) and then a term consisting of variables.

Example

```
minimize cost: 12 * x1 -4.4 * x2
   + sum <a> in A : u[a] * y[a]
   + sum <a,b,c> in C with a in E and b > 3 : -a/2 * z[a,b,c];
```

Constraints

The general format for a constraint is `subto name:  term sense rhs`. Name can be any name starting with a letter. The term is defined as in the objective. Sense is one of `<=`, `>=` and `==`. The rhs is any numeric expression. Many constraints can be generated with one statement by the use of the `forall` instruction, see below.

Examples

```
subto time : 3 * x1 + 4 * x2 <= 7;
subto space: sum <a> in A : 2 * u[a] * y[a] >= 50;
subto weird: forall <a> in A :
   sum <a,b,c> in C : z[a,b,c] == 55;
```

Details on `sum` and `forall`

The general forms are
    `forall` *index* `do` *term* and `sum` *index* `do` *term*.
The general form of *index* is
    tuple `in` *set* `with` *boolean-expression*.
It is allowed to write a colon (`:`) instead of `do` and a vertical bar (`|`) instead of `with`. The number of components in the *tuple* and in the components of the mebers of the *set* must match. The `with` part of an *index* is optional. The *set* can be any expression giving a set.

Examples

```
forall <i,j> in X cross { 1 to 5 } without { <2,3> }
   with i > 5 and j < 2 do
      sum <i,j,k> in X cross { 1 to 3 } cross Z do
         p[i] * q[j] * w[j,k] >= if i == 2 then 17 else 53;
```

Note that in the example *i* and *j* are set by the `forall` instruction. So they are fixed for all invocations of `sum`.

Initialising sets and parameters from a file

It is possible to load the values for a set or a parameter from a file. The syntax is

`read` *filename* `as` *template* [`skip` *n*] [`use` *n*] [`fs` *s*] [`comment` *s*]

filename is the name of the file to read.

template is a string with a template for the tuples to generate. Each input line from the file is split in fields. The splitting is done according to the following rules: Whenever a space, tab, comma, semicolon or double colon is encountered a new field is started. Text that is enclosed in double quotes is not split, the quotes are allways removed. When a field is split all space and tab charaters around the splitting are removed. If the split is due to a comma, semicolon or double colon, each occurence of these characters starts a new field.

Examples

All these lines have three fields:

```
Hallo;12;3
Moin    7  2
"Hallo, Peter"; "Nice to meet you" 77
,,2
```

For each component of the tuple the number of the field to use for the value is given, followed by either a `n` if the field should be interpreted as a number or `s` for a string. Have a look at the example, it is quite obvious how it works. After the template some optional modifiers can be given. The order does not matter.

`skip` *n* instructs to skip the first *n* lines of the file.

`use` *n* limits the number of lines to use to *n*.

`comment` *s* sets a list of characters that start comments in the file. Each line is ended when any of the comment characters is found.

When a file is read, empty lines are skiped and not counted for the `use` clause. They are counted for the `skip` clause.

Examples

```
set P := { read "nodes.txt" as <1s> };

nodes.txt:
   Hamburg    ->    <"Hamburg">
   München    ->    <"München">
   Berlin     ->    <"Berlin">
```

```
set Q := { read "blabla.txt" as "<1s,5n,2n>" skip 1 use 2 };

blabla.txt:
   Name;Nr;X;Y;No      ->   skip
   Hamburg;12;x;y;7   ->   <"Hamburg",7,12>
   Bremen;4;x;y;5      ->   <"Bremen,5,4>
   Berlin;2;x;y;8      ->   skip

param cost[P] := read "cost.txt" as "<1s> 4n" comment "#";

cost.txt:
   # Name Price    ->   skip
   Hamburg 1000    ->   <"Hamburg"> 1000
   München 1200    ->   <"München"> 1200
   Berlin  1400    ->   <"Berlin">  1400

param cost[Q] := read "haha.txt" as "<3s,1n,2n> 4s" fs ":";

haha.txt:
   1:2:ab:con1     ->   <"ab",1,2> "con1"
   2:3:bc:con2     ->   <"bc",2,3> "con1"
   4:5:de:con3     ->   <"de",4,5> "con1"
```

4 Error messages

Here are some of the incomprehensible error messages ZIMPL can produce:

Comparison of different types

It is not possible to compare a number with a string.

xxx of incompatible sets

The members of the two sets involved in operation *xxx* have not the same number of components.

Illegal element xxx for symbol

In the initialisation of a parameter, tuple *xxx* is not a member in the index set.

WITH not allowed here

When initialising the bounds of variables by index, the `with` clause to exclude some of the elements is not allowed.

Comparison of different dimension tuples

Two tuples were compared which have a different number of components. This is never a good idea, since such tuples are always different.

Comparison of elements with different types

Two tuples were compared which have a different type for the n-th element. This is never a good idea, since such tuples are always different.

Type error, expected xxx got yyy

In some context type *xxx* was expected, but the interpreter found type *yyy*. The meaning of the numbers could be looked up in `code.h`.

5 Remarks

ZIMPL is licensed under the GNU general public licence version 2. For more information on free software see `www.gnu.org`. The latest version of ZIMPL can be found at `www.zib.de/koch/zimpl`. If you find any bugs you can email me at `koch@zib.de`. Please include an example that shows the problem. If somebody extends ZIMPL, I am interested in getting patches to put them back in the main distribution.

References

[FGK93] R. Fourier, D. M. Gray, and B. W. Kernighan. *AMPL: A Modelling Language for Mathematical Programming*. boyd & fraser publishing company, Danvers, Massachusetts, 1993.

[IBM97] *IBM Optimization Library Guide and Reference*. IBM Corp., 1997.

[ILO00] *ILOG CPLEX 7.0 Reference Manual*. ILOG, 2000.

[vH99] Pascal van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, Cambridge, Massachusetts, 1999.

[XPR99] *XPRESS-MP Release 11 Reference Manual*. Dash Associates, 1999.

1.3

Search User Guide. Simple Modules. Modules in FUEL can range from simple forms to a complex, interactive, multi-page interface. Simple modules don't require additional views and controllers to be created. Common examples of simple modules include events, news and job postings. Â The name of the permission that users must be subscribed to to have access to the module. edit_method. The default is find_one_array filtered on the id value passed to the form. Select Compute Coordinates Kodi Engineering Services Pte Ltd Page 3 Simple User Guide of Spectrum Link Process Finish Once all points are coordinated, you now will be able to save as DXF or DWG file. You can also save your data to other text format for print out of information. Note: DXF / DWG you are require to do some setting before export. Save As Kodi