

# ZIMPL

(Zuse Institute Mathematical Programming Language)

Thorsten Koch

for Version 2.00  
17. September 2003

## 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 [FGK03]. 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 advantages. 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:

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

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.

## ZIMPL

---

```
NAME          ex1.mps
ROWS
  N  OBJECTIV
  L  c1
COLUMNS
  x          OBJECTIV          2
  x          c1                1
  y          OBJECTIV          3
  y          c1                1
RHS
  RHS        c1                6
ENDATA
```

Another possibility is the LP format [ILO02], which is more readable<sup>1</sup> but is only supported by a few solvers.

```
Minimize
  cost:  +2 x +3 y
Subject to
  c1:  +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 c1: 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

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

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;
```

---

<sup>1</sup> The LP format has also some idiosyncratic restrictions. For example variables should not be named e12 or the like. And it is not possible to specify ranged constraints.

## 2 Invocation

To run ZIMPL on a 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 as if they were all one big file. If any error occurs while processing, ZIMPL will print out an error message and abort. In case everything goes well, the results will be written into two or three files, depending on the options specified.

The first file is the problem generated from the model in either LP or MPS format with extension `.lp` or `.mps`. The next one is the “table” file, which has the extension `.tbl`. This file 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. Also the the LP format restricts the length of names to 16 characters.

The third file is and optional CPLEX branching order file.

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` or `hum` which is only human readable.
- `-o name` Sets the base-name for the output files.  
Defaults to the name of the first input file striped of the path and extension.
- `-F filter` The output is piped through a filter. A `%s` in the string is replaced by the output filename. For example `-p "gzip -c >%s.gz"` would compress all the output files.
- `-n cform` Select the format for the generation of constraint names. Can be either `cm` which will number them  $1 \dots n$  with a ‘c’ in front. `cn` will use the name supplied in the `subto` statement and number them  $1 \dots n$  within the statement. `cf` will use the name given with the `subto`, then a  $1 \dots n$  number like in `cm` and then append all the local variables from the `forall` statements.
- `-v l..5` Set the verbosity level. 0 is quiet, 1 is default, 2 is verbose, 3 is chatter, and 5 is debug.
- `-b` Enables bison debugging output.
- `-f` Enables flex debugging output.
- `-h` Prints a help message.
- `-p` Does some presolve analysis to fix variables.
- `-r` Writes an CPLEX `ord` branching order file.

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`.

**If MPS-output is specified for a maximization problem, the objective function will be inverted.**

### 3 Format

Each ZPL-file consists of five types of statements:

- ▶ Sets
- ▶ Parameters
- ▶ Variables
- ▶ Objective
- ▶ 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.

### Expressions

ZIMPL works on the lowest level with two types of data: Strings and numbers.

Wherever a number or string is required, it is also possible to give a parameter of the right value type. Usually expressions are allowed instead of just a number or a string. The precedence of operators should be the usual one, but parenthesis can always be used to specify the order of evaluation explicitly. If in doubt use parenthesis to be save.

### Numeric expressions

A number in ZIMPL can be given in the usual format, e. g. as 2, -6.5 or 5.234e-12. Numeric expressions consist of numbers, numeric valued parameters, and any of the following operators and functions:

$a^b, a^{**}b$	$a$ to the power of $b$	$a^b$
$a+b$	addition	$a + b$
$a-b$	subtraction	$a - b$
$a*b$	multiplication	$a \cdot b$
$a/b$	division	$a/b$
$a \bmod b$	modulo	$a \bmod b$
$a \operatorname{div} b$	integer division	
$\operatorname{abs}(a)$	absolute value	$ a $
$\operatorname{floor}(a)$	round down	$\lfloor a \rfloor$
$\operatorname{ceil}(a)$	round up	$\lceil a \rceil$
$a!$	factorial	$a!$
$\min(S)$	minimum of a set	$\min_{s \in S}$
$\max(S)$	maximum of a set	$\max_{s \in S}$
$\min(a, b, c, \dots, n)$	minimum of a list	$\min(a, b, c, \dots, n)$
$\max(a, b, c, \dots, n)$	maximum of a list	$\max(a, b, c, \dots, n)$
$\operatorname{card}(S)$	cardinality of a set	$ S $
$\text{if } a \text{ then } b \text{ else } c$	conditional	

### String expressions

A string is delimited by double quotation marks ", e. g., "Hallo".

The following is either a numeric or a string expression, depending if *expression* is a string or a numeric expression.

*if boolean-expression then expression else expression end.*

At this time no further functions or operators for strings are implemented.

### Boolean expressions

These evaluate either to *true* or *false*. For numbers the relational operators  $<$ ,  $\leq$ ,  $=$ ,  $\neq$ ,  $\geq$ , and  $>$  are defined. For strings only  $=$  and  $\neq$  are available. Combinations of boolean expressions with *and*, *or*, and negation with *not* are possible.

The expression *tuple in set-expression* can be used to test set membership of a tuple.

### Sets

Sets consist of tuples. The tuples in a sets are unordered, i. e., each tuple can only be once in a set. Sets are delimited by braces, { and }, resp. Tuples consist of components. The components are ordered. Each tuple of a specific set has the same number of components. The components are either numbers or strings. The type of the  $n$ -th component of each tuple of a set must be the same. A tuple starts and ends with  $<$  and  $>$ , resp., e. g.  $\langle 1, 2, "x" \rangle$ . The components are separated by commas. If tuples are one-dimensional, it is possible to omit the tuple delimiters in a list of elements, but then they must be omitted from all tuples of the set, e. g.  $\{1, 2, 3\}$ .

Sets can be defined with the set statement. This consists of the keyword `set`, the name of the set, an assignment operator (`:=`) and a valid set expression.

Sets are referenced by use of an *template* tuple, consisting of placeholders that are replaced by the value of the components of the respective tuple. For example a set  $S$  consisting of two dimensional tuples could be referenced by `<a,b> in S`. If any of the placeholders are actual values, only those tuples will be extracted that match these values. For example `<1,b> in S` will only get those tuples whose first component is 1. Please note that if one of the placeholders is the name of an already defined parameter, set or variable, it will be substituted which will either result in an error or an actual value.

### Examples

```
set A := { 1, 2, 3 };
set B := { "hi", "ha", "ho" };
set C := { <1,2,"x">, <6,5,"y">, <787,12.6,"oh"> };
```

For set expressions these functions and operators are defined:

$A * B$ ,		
<code>A cross B</code>	Cross product	$\{(x, y)   x \in A \wedge y \in B\}$
$A + B$ ,		
<code>A union B</code>	Union	$\{x   x \in A \vee x \in B\}$
<code>A inter B</code>	Intersection	$\{x   x \in A \wedge x \in B\}$
$A \setminus B$ , $A - B$ ,		
<code>A without B</code>	Difference	$\{x   x \in A \wedge x \notin B\}$
<code>A symdiff B</code>	Symmetric difference	$\{x   (x \in A \wedge x \notin B) \vee (x \in B \wedge x \notin A)\}$
$\{n..m\}$ ,		
<code>{n to m by s}</code>	Generate, (default $s = 1$ )	$\{x   x = n + is \leq m, i \in \mathbb{N}_0\}$
<code>proj(A, t)</code>	Projection $t = (e_1, \dots, e_n)$	The new set will consist of $n$ -tuples, with the $i$ -th component being the $e_i$ -th component of $A$ .

### Examples

```
set D := A cross B;
set E := { 6 to 9 } union A without { <2>, <3> };
set F := { 1 to 9 } * { 10 to 19 } * { "A", "B" };
set G := proj(F, <3,1>)
# will give: <"A",1>, <"A",2> ... <"B",9>
```

## Conditional sets

It is possible to restrict a set to tuples that satisfy a boolean expression. The expression given by the `with` clause is evaluated for each tuple in the set and only tuples for which the expression evaluated to *true* are included in the new set.

## Examples

```
set F := { <i,j> in Q with i > j and i < 5 };
set A := { "a", "b", "c" };
set B := { 1, 2, 3 };
set V := { <a,2> in A*B with a == "a" or a == "b" };
# will give: <"a",2>, <"b",2>
```

## Indexed sets

It is possible to index one set with another set. Here is a list of functions that use this:

<code>powerset(A)</code>	Generates all subsets of $A$	$\{x x \subseteq A\}$
<code>subset(A,n)</code>	Generates all subsets of $A$ with $n$ elements	$\{x x \subseteq A \wedge  x  = n\}$
<code>indexset(A)</code>	The index set of $A$	$\{1 \dots  A \}$

Indexed sets are accessed by adding the index of the set in brackets [ and ], like `S[ 7 ]`. There are three possibilities how to assign to an indexed set:

- The assignment expression is a list of comma separated pairs, consisting of a tuple from the index set and a set expression to assign.
- A set reference expression is given as index, then the assignment expression is evaluated for each index tuple.
- By use of a function that returns a indexed set.

## Examples

```
set I           := { 1..3 };
set A[I]        := <1> { "a", "b" },
                  <2> { "c", "e" }, <3> { "f" };
set B[<i> in I] := { 3 * i };
set P[]         := powerset(N);
set J           := indexset(P);
set S[]         := subset(I, 2);
```

## 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. It is possible to declare a *default* value. 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 default 99;
param w[C] := <1,2,"x"> 1/2, <6,5,"y"> 2/3;
param x[<i> in I with i mod 2 == 0] := 3 * i;
```

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). Bounds can also be set to `infinity` and `-infinity`.

### 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;
```

Remember: if nothing is specified a lower bounds of zero is assumed.



## Objective

There must be at most 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 term`. Name can be any name starting with a letter. The term is defined as in the objective. Sense is one of `<=`, `>=` and `==`. 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;
subto c21: 6 * (sum <i> in A : x[i]
    + sum <j> in B : y[j]) >= 2;
subto c40: x[1] == a[1] +
    2 * sum <i> in A do 2*a[i]*x[i]*3 + 4;
```

## Details on sum and forall

The general forms are

`forall index do term` and `sum index do term`.

It is possible to nest several forall instructions. 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 members 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:
```

## ZIMPL

---

```
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";

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 is a list of the incomprehensible error messages ZIMPL can produce:

### 101 Bad filename

The name given with the `-o` option is either missing, a directory name, or starts with a dot.

### 102 File write error

Some error occurred when writing to an output file. A description of the error follows on the next line. For the meaning consult your OS documentation.

### 103 Output format not supported, using LP format

You tried to select another format than `lp`, `mps`, or `hum`.

### 104 File open failed

Some error occurred when trying to open a file for writing. A description of the error follows on the next line. For the meaning consult your OS documentation.

### 105 Duplicate constraint name “xxx”

Two `subto` statements have the same name.

### 106 Empty LHS, constraint trivially violated

One side of your constraint is empty and the other not equal to zero. This mostly happens if a set to sum up is empty.

### 107 Range must be $l \leq x \leq u$ , or $u \geq x \geq l$

If you specify a range you must have the same comparison operators on both sides.

### 108 Empty Term with nonempty LHS/RHS, constraint trivially violated

The middle of your constraint is empty and either the left- or right-hand side of the range is not zero. This mostly happens if a set to sum up is empty.

### 109 LHS/RHS contradiction, constraint trivially violated

The lower side of your range is bigger than the upper side, e.g.  $15 \leq x \leq 2$ .

### 110 Division by zero

You tried to divide by zero. This is not a good idea.

### 111 Modulo by zero

You tried to compute a number modulo zero. This does not work well.

### 112 Exponent value xxx is too big or not an integer

It is only allowed to raise a number to the power of integers. Also trying to raise a number to the power of more than two billion is prohibited.<sup>2</sup>

---

<sup>2</sup>The behaviour of this operation could easily be implemented as `for(i;i)` or more elaborate as `void f(){f(i);}`.

**113 Factorial value xxx is too big or not an integer**

You can only compute the factorial of integers. Also computing the factorial of a number bigger than two billion is generally a bad idea. See also Error 115.

**114 Negative factorial value**

To compute the factorial of a number it has to be positive. In case you need it from a negative number, remember that for all even numbers the outcome will be positive and for all odd number negative.

**115 Timeout!**

You tried to compute a number bigger than 1000!. See also the footnote to Error 112.

**116 Illegal value type in min: xxx only numbers are possible**

You tried to build the minimum of some strings.

**117 Illegal value type in max: xxx only numbers are possible**

You tried to build the maximum of some strings.

**118 Comparison of different types**

You tried to compare apples with oranges, i.e., numbers with strings.

**119 Union of incompatible sets**

To unite two sets, both must have the same dimension tuples, i.e., the tuples must have the same number of components.

**120 Minus of incompatible sets**

To subtract two sets, both must have the same dimension tuples.

**121 Intersection of incompatible sets**

To intersect two sets, both must have the same dimension tuples.

**122 Symetric Difference of incompatible sets**

To build the symmetric difference of two sets, both must have the same dimension tuples.

**123 “from” value xxx in range too big or not an integer**

To generate a set, the “from” number must be an integer with an absolute value of less than two billion.

**124 “upto” value xxx in range too big or not an integer**

To generate a set, the “upto” number must be an integer with an absolute value of less than two billion.

**125 “step” value xxx in range too big or not an integer**

To generate a set, the “step” number must be an integer with an absolute value of less than two billion.

**126 Zero “step” value in range**

The given “step” value for the generation of a set is zero. So the “upto” value can never be reached.

**127 Illegal value type in tuple: xxx only numbers are possible**

The selection type in a call to the `proj` function can only contain numbers.

**128 Index value xxx in proj too big or not an integer**

The value given in a selection tuple of a `proj` function is not an integer or bigger than two billion.

**129 Illegal index xxx, set has only dimension yyy**

The index value given in a selection tuple is bigger than the dimension of the tuples in the set.

**130 Duplicate index xxx for initialisation**

In the initialisation of a indexed set, two initialisation elements have the same index. E.g, set `A[] := <1> { 1 }, <1> { 2 };`

**131 Illegal element xxx for symbol**

The index tuple used in the initialisation list of a index set, is not member of the index set of the set. E.g, set `A[{ 1 to 5 }] := <1> { 1 }, <6> { 2 };`

**132 Values in parameter list missing, probably wrong read template**

As the message said, probable the template of a read statement looks like "`<1n>`" instead of "`<1n> 2n`".

**133 Unknown local symbol xxx**

A local symbol was used, that is not defined anywhere in scope.

**134 Illegal element xxx for symbol**

The index tuple given in the initialisation is not member of the index set of the parameter.

**135 Index set for parameter xxx is empty**

The attempt was made to declare an indexed parameter with the empty set as index set. Most likely the index set has a `with` clause which has rejected all elements.

**136 Lower bound for var xxx set to infinity – ignored**

In the zimpl code something like `≥ infinity` must have been appeared. This makes no sense and is therefore ignored.

**137 Upper bound for var xxx set to -infinity – ignored**

In the zimpl code something like `≤ -infinity` must have been appeared. This makes no sense and is therefore ignored.

**138 Priority/Startval for continous var xxx ignored**

There has been set a priority or a starting value for a continous (real) variable. This is not possible and therefore ignored.

**139 Lower bound for integral var xxx truncated to yyy**

An integral variable can only have an integral bound. So the given non integral bound was ajusted.

**140 Upper bound for integral var xxx truncated to yyy**

An integral variable can only have an integral bound. So the given non integral bound was ajusted.

**141 Infeasible due to conflicting bounds for var xxx**

The upper bound given for a variable was smaller than the lower bound.

**142 Unknown index xxx for symbol yyy**

The index tuple given is not member of the index set of the symbol.

**143 Size for subsets xxx is too big or not an integer**

The cardinality for the subsets to generate must be given as an integer smaller than two billion.

**144 Tried to build subsets of empty set**

The set given to build the subsets of, was the empty set.

**145 Illegal size for subsets xxx, should be between 1 and yyy**

The cardinality for the subsets to generate must be between 1 and the cardinality of the base set.

**146 Tried to build powerset of empty set**

The set given to build the powerset of, was the empty set.

**147 use value xxx is too big or not an integer**

The use value must be given as an integer smaller than two billion.

**148 use value xxx is not positive**

Negative or zero values for the use parameter are not allowed.

**149 skip value xxx is too big or not an integer**

The skip value must be given as an integer smaller than two billion.

**150 skip value xxx is not positive**

Negative or zero values for the skip parameter are not allowed.

**151 Not a valid read template**

A read template must look something like "<1n , 2n>". There have to be a < and a > in this order.

**152 Invalid read template syntax**

Apart from any delimiters like `<`, `>`, and commas a template must consists of number charater pairs like `1n`, `3s`.

**153 Invalid field number xxx**

The field numbers in a template have to be between 1 and 255.

**154 Invalid field type xxx**

The only possible field types are `n` and `s`.

**155 Invalid read template, not enough fields**

There has to be at least one field inside the delimiters.

**156 Not enough fields in data**

The template specified a field number that is higher than the actual number of field found in the data.

**157 Not enough fields in data (value)**

The template specified a field number that is higher than the actual number of field found in the data. The error occured behind the index tuple in the value field.

**158 Read from file found no data**

Not a single record could be read out of the data file. Either the file is empty, or all lines are comments.

**159 Type error, expected xxx got yyy**

The type found was not the expected one, e.g. subtracting a string from a number would result in this message.

**160 Comparison of elements with different types xxx / yyy**

Two elements from different tuples were compared and found to be of different types. Probably you have a index expression with a constant element, that does not match the elements of the set. Example: `set B:={<i,"a"> in {<1,2>}}`;

**161 Line xxx: Unterminated string**

This line has an odd number of `"` characters. A String was started, but not ended.

**162 Line xxx: Trailing "yyy" ignored**

Something was found after the last semicolon in the file.

**163 Line xxx: Syntax Error**

Either a new statement was not started with one of the keywords: `set`, `param`, `var`, `minimize`, `maximize`, `subto`, and `print`, or no name was given after the keyword.



**164 Duplicate element xxx for set rejected**

An element was added to a set that was already in it.

**165 Comparison of differen dimension sets**

Two sets were compared, but have different dimension tuples. (This means they never had a chance other then beeing empty sets to be equal.)

**166 Duplicate element xxx for symbol yyy rejected**

An element that was already there was added to a symbol.

**167 Comparison of different dimension tuples**

Two tuples with different dimensions were compared. Probably you have a index expression which does not match the dimension of the tuples in the set. Example:  
`set B:={<1> in {<1,2>}};`

**168 No program statements to execute**

No ZIMPL statements were found in the files loaded.

**169 Execute must return void element**

This should not happen. If you encounter this error please email the .zpl file to <mailto:koch@zib.de>.

**800 parse error: expecting xxx (or yyy)**

Parsing error. What was found was not what was expected. The statement you entered is not valid.

**801 Parser failed**

The parsing routine failed. This should not happen. If you encounter this error please email the .zpl file to <mailto:koch@zib.de>.

## 5 Remarks

ZIMPL is licensed under the GNU general public licence version 2. For more information on free software see <http://www.gnu.org>. The latest version of ZIMPL can be found at <http://www.zib.de/koch/zimpl>. If you find any bugs you can email me at <mailto: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.
- [FGK03] R. Fourier, D. M. Gray, and B. W. Kernighan. *AMPL: A Modelling Language for Mathematical Programming*. Brooks/Cole—Thomson Learning, second edition, 2003.
- [IBM97] *IBM Optimization Library Guide and Reference*. IBM Corp., 1997.
- [ILO02] *ILOG CPLEX 8.0 Reference Manual*. ILOG, 2002.
- [vH99] Pascal van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, Cambridge, Massachusetts, 1999.
- [XPR99] *XPRESS-MP Release 11 Reference Manual*. Dash Associates, 1999.