CMPL

<Coliop|Coin> Mathematical Programming Language

Version 2.0.1

April 2022

Manual

M. Steglich, T. Schleiff

Table of contents

1 About CMPL

CMPL (<Coliop|Coin> Mathematical Programming Language) is a mathematical programming language and a system for mathematical programming and optimisation of linear optimisation problems.

The CMPL syntax is similar in formulation to the original mathematical model but also includes syntactic elements from modern programming languages. CMPL is intended to combine the clarity of mathematical models with the flexibility of programming languages.

CMPL executes CBC, GLPK, Gurobi, SCIP or CPLEX directly to solve the generated model instance. Because it is also possible to transform the mathematical problem into MPS or Free-MPS, alternative solvers can be used.

CMPL is an open-source project licensed under GPL. It is written in C++ and is available for most of the relevant operating systems (Windows, OS X and Linux).

The CMPL distribution contains **Coliop** which is CMPL's IDE (Integrated Development Environment). Coliop is an open-source project licensed under GPL. It is written in C++ and is as an integral part of the CMPL distribution.

The CMPL package also contains pyCMPL, jCMPL and CMPLServer.

pyCMPL is the CMPL application programming interface (API) for Python and an interactive shell and **jCMPL** is CMPL's Java API. The main idea of this APIs is to define sets and parameters within the user application, to start and control the solving process and to read the solution(s) into the application if the problem is feasible. All variables, objective functions and constraints are defined in CMPL. These functionalities can be used with a local CMPL installation or a CMPLServer.

CMPLServer is an XML-RPC-based web service for distributed and grid optimisation that can be used with CMPL, pyCMPL and jCMPL. It is reasonable to solve large models remotely on the CMPLServer that is installed on a high performance system. CMPL provides four XML-based file formats for the communication between a CMPLServer and its clients. (CmplInstance, CmplSolutions, CmplMessages, CmplInfo).

pyCMPL, jCMPL and CMPLServer are licensed under LGPLv3.

CMPL, Coliop, pyCMPL, jCMPL and CMPLServer are COIN-OR projects initiated by the Technical University of Applied Sciences Wildau.

2 CMPL Language reference manual

2.1 CMPL elements

2.1.1 General structure of a CMPL model

The structure of a CMPL model follows the standard model of linear programming (LP), which is defined by a linear objective function and linear constraints.

$c^T \cdot x \rightarrow max!$

$s.t.$

$A \cdot x \leq b$

$x \geq 0$

In such a model, four different types of objects can be distinguished:

`variables` `var`	Decision variables (columns within the linear programming model)
`objectives` `obj`	Objective functions (neutral rows within the linear programming model)
`constraints` `con`	Constraints (restricted rows within the linear programming model)
`parameters` `par`	Given values within the model

A CMPL model consists of definitions of objects of these four types. The model can be divided into sections, each introduced by the name of the object type, and containing the associated definitions. For example, a simple CMPL model can have the following structure:

```
par:
    // definition of the parameters
var:
    // definition of the variables
obj:
    // definition of the objective(s)
con:
    // definition of the constraints
```

A typical LP problem is the production mix problem. The aim is to find an optimal quantity for the products, depending on given capacities. The objective function is defined by the profit contributions per unit `c` and the variable quantity of the products `x`. The constraints consist of the use of the capacities and the ranges for the decision variables. The use of the capacities is given by the product of the coefficient matrix `A` and the vector of the decision variables `x` and restricted by the vector of the available capacities `b`.

The simple example:

$1 \cdot x_1 + 2 \cdot x_2 + 3 \cdot x_3 \rightarrow max!$

$s.t.$

$5.6 \cdot x_1 + 7.7 \cdot x_2 + 10.5 \cdot x_3 \leq 15$

$9.8 \cdot x_1 + 4.2 \cdot x_2 + 11.1 \cdot x_3 \leq 20$

$0 \leq x_n ; n \in \{1,2,3\}$

can be formulated in CMPL as follows:

```
par:
   c := ( 1, 2, 3 );
   b := ( 15, 20 );

   A := (( 5.6, 7.7, 10.5 ),
         ( 9.8, 4.2, 11.1 ));

var:
   x[defset(c)]: real;

obj:
   profit: c^T * x -> max;

con:
   A * x <= b;
   x >= 0;
```

2.1.2 Statements and expressions

A CMPL model consists of statements. Each statement is completed with a semicolon. The essential statement is the definition or assignment. A symbol on the left-hand side is assigned the value of the expression on the right-hand side. If the symbol on the left-hand side has not yet been defined, it is thereby defined as well. If the assignment is in a section for an object type, the expression on the right-hand side is converted into an object of this type.

The both most important operators are:

:=	Assignment (assigns the value of the right hand side expression to the CMPL symbol on the left hand side.)
:	Only allowed for the definition of a decision variable (representing a column in the LP problem matrix), an objective or a constraint (representing a row in the LP problem matrix). The name of the symbol on the left hand side is used as the name for the column or row in the LP problem matrix represented by the value on the right hand. The symbol is assigned an object which can be a decision variable, objective or constraint.

In the example, the following statement defines on the right-hand side a vector with three elements. In addition, a symbol c is defined to which the vector is assigned.

```
par:
   c := ( 1, 2, 3 );
```

A vector with two elements is assigned to a newly defined symbol b.

```
   b := ( 15, 20 );
```

Then a matrix with 2x3 elements is defined on the ride hand side and assigned to the newly defined symbol A.

```
   A := (( 5.6, 7.7, 10.5 ),
         ( 9.8, 4.2, 11.1 ));
```

In the variables section, a vector x of decision variables with the type real is defined.

```
var:
   x[defset(c)]: real;
```

The left-hand side defines a vector of variables with the name x. The function *defset*(c) ensures that this vector uses the same indices as the vector c. The expression on the right side is the data type real. Since the statement is in the var section, this expression is converted into a decision variable with data type real. Such a decision variable is created and assigned for any element of the vector x. The columns in the LP problem matrix for these decision variables are labelled x[1], x[2] and x[3].

In the following, the objective function with the name profit is defined.

```
obj:
   profit: c^T * x -> max;
```

The right-hand side (after the colon) is a formula expression, created from symbols defined before and the objective sense max. Because the statement stands in the obj section, this expression is converted into an objective function of the LP problem and assigned to symbol profit. Moreover, the row of this objective function in the LP problem matrix is also labelled profit.

The constraints are defined in the constraints section.

```
con:
   A * x <= b;
   x >= 0;
```

These statements consist of right-hand sides only. According to the constraint section, the formula expression is converted into a constraint of the LP problem. Since there is no left-hand side, no symbol or row name is defined. Therefore, the rows are given a default name.

The usual operators can be used in expressions. If the operands are vectors or matrices, a matrix operation is performed. Important operators are:

Arithmetical:

+	Sign or addition, also string concatenation
-	Sign or subtraction
*	Multiplication, If number is multiplied by a symbol, the multiplication operator can be omitted. (e.g 2x is identical to 2*x.
/	Division
^	To the power of

Transpose:

^T	Transpose vector or matrix (mainly used in matrix multiplication)

Comparison:

=	Equal to
>=	Greater than or equal
<=	Less than or equal to
<>	Unequal

`>`	Greater than
`<`	Less than

Construction operators:

`( … )`	• Array construction from elements (comma separated) or • Function call with the constructed array as parameter or • Arithmetical bracketing in expressions
`[ … ]`	• Tuple construction from elements or • Indexing operation using the constructed tuple as index or • Parameter for construction of a restricted type
`..`	• Interval construction between lower and upper bound • If used in a context where a set is expected, then the interval is converted to a set.

In addition, a CMPL model can contain comments at any point:

`//` `#`	Comment up to end of line
`/* … */`	Comment between `/*` and `*/`

2.1.3 Data types and arrays

2.1.3.1 Data types

Objects in CMPL have a data type in addition to the object type. There are simple data types (consisting of exactly one element) and composite data types (consisting of several objects, each of which has its own data type).

Almost all data types are only permitted for parameter objects. Some data types can be used for variables. Objective functions and constraints implicitly always have the data type `real`. In most cases, the data type only has to be explicitly specified for variables, for parameters it results automatically from the type of the assigned expression.

Simple data types usable also for variables:

`real`	• floating point number (uses internally C data type double) • literal value consists of digits, decimal point and optional exponent
`int`	• integer number (uses internally C data type long)
`integer`	• literal value consists only of digits
`bin`	• integer value which can only be 0 or 1
`binary`	• also usable as boolean value
	• literal values are `true` (value 1) and `false` (value 0)

Other important simple data types:

`string`	• character string • literal value is enclosed in double quotes

Composite data types:

`interval`	• Interval between two numeric values • If the lower or the upper bound is omitted, then the interval is unbounded on this side.
`tuple`	• Tuple of an arbitrary number of elements (also no element) with any data type. • A special kind of tuple is an index tuple, which consists only of elements of the data types `int` and `string`.
`set`	• Set of an arbitrary number of elements (also no element), all elements must be index tuples.
`formula`	• An expression of parameters and decision variables • Note that a formula is not a constraint, but an appropriate formula can be converted into a constraint.
`container`	• A value that contains other symbols (similar to struct or class in C)

As far as is reasonably possible, expressions of one data type can be converted into another data type. Such a data type cast has the form of a function call, where the data type name is used as the function name.

Some types can be further restricted by type parameters. Such type parameters are specified in the form of a tuple after the type name. The most important use is to restrict the range of validity for a decision variable.

Examples (within a `var:` section):

`x: real;`	Defines `x` as a real decision variable with default range ($x \geq 0$).
`x: real[0..100];`	Defines `x` as a real decision variable $0 \leq x \leq 100$.
`x: real[..];`	Defines `x` as a real decision variable with no ranges.
`y: integer[1..];`	Defines `y` as an integer decision variable $y \geq 1$.
`z: binary;`	Defines `z` as an integer decision variable $z \in \{0,1\}$.

2.1.3.2 Sets

A set is a collection of indices. Every element within a set is an `n`-tuple (pair of `n` entries) of integers or strings, where `n` is named the rank of the tuple. A tuple is constructed by the `[ … ]` operator, or for 1-tuples also contextually converted from a single integer or string.

Usage:

[*entry-1 [, entry-2, … , entry-n]* **]**	Construction of an n-tuple. The resulting tuple is an index tuple, if all entries are `int` or `string`.

There are two special tuples:

`[null]`	0-tuple. This special index tuple is the only tuple with rank 0. It can be element within a set as other index tuples.
`[]`	Empty tuple. Note that this is **not** the 0-tuple, it is not even an index tuple. Because it's not an index tuple it cannot be element within a set. But it can be converted to a set, and means then the full set (infinite set which contains every possible index tuple.)

In addition to the elements they contain, sets in CMPL are also characterised by the order of these elements. This order is relevant in iterations when the keyword `ordered` is used. The order of the elements is determined during the construction of a set.

Set construction:

set(*tuple1, tuple2, …*)	Constructs a set from the given index tuples. Instead of index tuples also int or string values are allowed, which will be converted to 1-tuples. The order of the elements is determined during the construction of a set. e.g.: `set("a", "b", "c")` `set(1, [1,2], [1,"a"])`
set(*interval*)	Constructs a set of the 1-tuples of all integer values within the interval. If the interval has no lower or no upper bound, the resulting set will be infinite. The elements are ordered from the lowest to the highest value. e.g.: `set(1..10)` `set(0..)`
set()	Empty set (without any element)
start(*increment*)*end*	Constructs a set of 1-tuples of integers which starts with `start`, is incremented or decremented in each step by `increment` and ends with value `end`. The values `start`, `increment` and `end` must be integers. For the `increment` the value 0 is not allowed, but it can be negative. If `increment` is positive, the elements are ordered from the lowest to the highest. If `increment` is negative, the elements are ordered from the highest to the lowest. e.g.: `1(1)10` is equal to `set(1..10)` `10(-3)0` is equal to `set(10, 7, 4, 1)`
*****_tuple_	Constructs a set with only one element. Instead of an index tuple, an `int` or `string` value is also allowed, which is converted into a 1-tuple.
interval	If an interval value is used where a set is expected, then the interval will contextually converted to a set.
`[]`	Empty tuple. If it is used where a set is expected, it means the full set. This is the infinite set containing all possible index tuples.
`[ … ]`	If the tuple contains at least one element that is an interval or a set, and if it is used where a set is expected, then the tuple is converted to a set. If the tuple contains elements not suitable for this conversion, an error is generated. e.g.: `[1..2, 1..2]` `[set("a","b","c"), 1]`

Infinite sets are important for indexing arrays and for matching operations. They are usually composed with `[ ... ]`. Important infinite components are:

`*`	All index tuples of rank 1. e.g.: `[*]` set of all index tuples of rank 1 `[*,*]` set of all index tuples of rank 2 `[1,*]` set of all index tuples of rank 2 which have 1 as first part
`/`	Also all index tuples of rank 1, but marks the part to be discarded in the match operation. e.g.: `[*,/]` set of all index tuples of rank 2, but return in match operation only the first part of the matching tuples
`..`	All 1-tuples that consists of an integer value e.g.: `[1,..]` set of all index tuples of rank 2 which have 1 as first part and any integer as second part
`<empty>`	All index tuples of all ranks (also rank 0) e.g.: `[]` set of all possible index tuples including the 0-tuple `[*,]` set of all index tuples of at least rank 1 (i.e. all possible index tuples save the 0-tuple) `[,1,]` set of all index tuples of all ranks, which have a 1 as part at arbitrary position (because `[]` contains the 0-tuple, the 1 can be also at the first or the last position in the index tuples)

The most important operations and functions for sets are:

`t` **`in`** *`s`*	Checks whether an index tuple `t` is element of the set `s`, result is a binary value. (Also used for iterating over all index tuples within `s`.) e.g.: `3 in 1..10` results true `[1,1] in [1,*]` results true `[1,1] in [*]` results false
`s1` **`*>`** *`s2`*	Match operation. Builds the intersection between the two sets and then removes from the tuples of the result set the parts that correspond to parts of `s2`, but are not a set or are marked with `/`. e.g.: `s1 *> [1,*]` Finds all 2-tuples in `s1` with the first entry equal to 1 and returns a set only consisting of the second entries of the tuples found. `s1 *> [*1,*]` Finds all 2-tuples in `s1` with the first entry equal to 1 and returns this set of tuples. (also the first part is included, because `*1` is a set). `s1 *> [*,/]` Finds all 2-tuples in `s1`, and returns a set of only consisting of the first parts of the tuples found.

len(*s*)	Returns the count of elements within the set `s`. If `s` is an infinite set, the result is the special value `inf`.
rank(*s*)	Returns the rank of set `s`. If all tuples within `s` have the same rank, the result is this rank. Otherwise the result is an interval from the minimum rank to the maximum rank of the tuples. The result for the empty set is 1.

2.1.3.3 Arrays

All data in CMPL is organised as an array. Indices of an array are index tuples. The set of all index tuples that are indices in a given array represent the definition set of the array.

Scalar values can be considered as an array with one element whose index is the 0-tuple.

An array is constructed by `( ... )`. If a separating comma is contained within, the corresponding definition set is `set(1..n)`, where `n` is the number of specified elements. A comma can also be placed after the last element. If there is no comma, the created array does have a 0-tuple set as the definition set. In this case, the brackets therefore act like simple arithmetic brackets.

Examples

`(0.5, 1, 2, 3.3, 5.5)`	Array of 5 numbers. Definition set is: `set(1..5)`
`(3)`	Array with only one element. Definition set is: `set([null])`
`(3,)`	Also array with only one element. Definition set is: `set(1)`
`(4,,7)`	Array of two numbers. Definition set is: `set(1,3)`
`()`	Empty array (with no element). Definition set is: `set()`
`null`	Also an empty array

In addition to numbers, elements in an array can also be data objects of any type, i.e. tuples, sets, decision variables or constraints. It is also possible to mix data objects of different types within an array.

Arrays themselves, however, cannot be elements in another array. Instead, in a nested array construction, the arrays are combined into an entire array with the corresponding definition set.

Examples

`((1, 2), (3, 4))`	Array of four numbers. Definition set is: `set([1,1], [1,2], [2,1], [2,2])` which is equivalent to: `set([1..2, 1..2])`
`((3))`	Array with only one element. Definition set is: `set([null, null])` which is equivalent to: `set([null])`
`(1, (2, (3, 4)))`	Array of 4 numbers. Definition set is: `set([1], [2,1], [2,2,1], [2,2,2])`

The individual elements of an array and partial arrays can be accessed with indexing. To do this, a tuple must be specified directly after the name of the array. This tuple must either be an index tuple or the tuple used for indexing must be convertible into a set.

In the first case, the single element to be retrieved from the array is the one belonging to this index tuple. If the definition set of the array does not contain the specified index tuple, an error occurs.

In the second case, a partial array is retrieved. To do this, a matching operation is internally performed with the definition set of the array as the first operand and the set specified as the tuple as the second operand. The result is an array of the elements of the matching index tuple, with the set resulting from the match operation as the definition set. If the match operation results in the empty set, this is not an error, but the result is an empty array.

Examples

`a := ((11, 12), (13, 14));`	Given example arrays
`b := (21, (22, (23, 24)));`	
`c := 31;`	
`a[1]`	results an error
`b[1]`	results: `21`
`c[1]`	results an error
`a[2,1]`	results: `13`
`b[2,1]`	results: `22`
`c[2,1]`	results an error
`a[`**`null`**`]`	results an error
`b[`**`null`**`]`	results an error
`c[`**`null`**`]`	results: `31`
`a[]`	results array `a` itself
`b[]`	results array `b` itself
`c[]`	results array `c` itself
`a[2,]`	results array: `(13, 14)`
`b[2,]`	results array: `(22, (23, 24)`
`c[2,]`	results empty array
`a[2,1..]`	results array: `(13, 14)`
`b[2,1..]`	results: `22` but with definition set: `set(1)`
`c[2,1..]`	results empty array
`a[*2,*1]`	results: `13` but with definition set: `set([2,1])`
`b[*2,*1]`	results: `22` but with definition set: `set([2,1])`
`c[*2,*1]`	results empty array

Indexing has a slightly different meaning when it is applied to the left-hand side of an assignment. Then the indexing determines which elements of the array are assigned values. If an element under the corresponding index tuple does not yet exist in the array, it is added.

Examples

`a := (11, 12, 13);`	No left hand side indexation. The specified array is assigned, and all previous content of `a` are overwritten.

`a[] := (11, 12, 13);`	Indexation with the full set. The given array is assigned, and inserts or overwrites the elements with index tuples `[1], [2], [3]`. All other elements of `a` remain unchanged.
`a[4] := 14;`	The element with index tuple `[4]` is inserted or overwritten. All other elements of `a` remain unchanged.
`p := set("str1","str2");` `a[p] := (1, 2);`	Assigns `a["str1"] := 1` and `a["str2"] := 2`
`a[4] := (14, 15);`	Error because an attempt is made to assign an array to a single element.
`a[4..] := (14, 15);`	Assigns `a[4] := 14` and `a[5] := 15`
`A[4, 1..] := (14, 15);`	Assigns `a[4,1] := 14` and `a[4,2] := 15`

2.1.3.4 Special values

There are some special values in CMPL:

inf	Infinite value of data type `real` Can be used in interval construction, e.g. `-inf..inf`. Can be used in numeric expressions, e.g. `inf + 1` results to `inf`
invalid	Is not a real value, but a marker for a non existing value. Can be used in assignment and in array construction. e.g. `(1,invalid,2)` is an array with 3 elements and definition set `set(1, 2, 3)`, in which the second element has not a value yet. A value can be checked with the function `valid` for validity.
null	Empty array whichhas special semantics in certain contexts: • Array construction: Marks a non existing element. Note that this is different from an existing element with no value (marked with `invalid`) e.g. `(1,null,2)` is an array with 2 elements and definition set `set(1, 3)`. • Tuple construction: Converted into a 0-tuple. So `[1,null,2]` is equivalent to `[1,2]`. • Arithmetic addition: Converted into value 0. So `null+2` results to `2`. (also `invalid+2` results to `2`.) • Arithmetic multiplication: Converted into value 1. So `null*2` results to `2`. (also `invalid*2` results to `2`.) • String concatenation: Converted to empty string. So `null+"abc"` results to `"abc"` (also `invalid+"abc"` results to `"abc"`.)

<empty>	An omitted value has special semantics in certain contexts: • Array construction: Marks a non existing element (equivalent to `null`). • Tuple construction: Converted to the full set of all possible index tuples of all ranks. • Interval construction: Converted to the infinite value (equivalent to `-inf` (on the left side of operator `..`) or to `inf` (on the right side of `..`)

2.1.3.5 Functions and operations for arrays

Important functions and operators for arrays are:

`defset``(a)`	Returns the definition set of array `a`
`validset``(a)`	Returns the set of all index tuples of array `a`, for which a value exists in the array (i.e. for which the value is not `invalid`) If the array contains only valid values, then `validset` results the same as `defset`.
`t` **`of`** `a`	Checks whether an index tuple `t` is an indexing tuple with a valid value in the array `a` Equivalent to: `t` **`in validset`**`(a)` Like `in` also `of` can be used for iterations.

Arrays can be used as operands in operations:

`+` **`-`**	• One operand is a non-empty array, the other a scalar value: Performs the operation for every element of the array and the scalar value, the result is an array with the same definition set as the operand array. e.g.: `(1, 2, 3) + 1` results in `(2, 3, 4)` • Both operands are arrays: The definition set of both arrays must be equal, otherwise it is an error. It performs the operation for every pair of elements of the operand arrays with the same indexing tuple. e.g.: `(1, 2) + (3, 4)` results in `(4, 6)`
`*`	• One operand is a non-empty array, the other a scalar value: Performs the operation for every element of the array and the scalar value. The result is an array with the same definition set as the operand array. e.g.: `(1, 2, 3) * 2` results in `(2, 4, 6)` • First operand is a transposed array of rank 1, second operand is a non-transposed array of rank 1: The definition set of both arrays must be equal, otherwise it is an error. Performs matrix multiplication of a row vector with a column vector. e.g.: `(1, 2)^T * (3, 4)` results in `11`

	• First operand is a non-transposed array of rank 1, second operand is a transposed array of rank 1: Performs matrix multiplication of a column vector with a row vector. e.g.: `(1, 2) * (3, 4)^T` results in `((3, 4), (6, 8))` • First operand is a rectangular array of rank 2, second operand is a non-transposed array of rank 1: The definition set of the second operand must match the second part of the definition set of the first operand, otherwise it is an error. Performs a multiplication of a matrix with a column vector. e.g.: `((1, 2), (3, 4)) * (5, 6)` results in `(17, 39)` • First operand is a transposed array of rank 1, second operand is a rectangular array of rank 2: The definition set of the first operand must match the first part of the definition set of the second operand, otherwise it is an error. Performs multiplication of a row vector with a matrix. e.g.: `(1, 2)^T * ((3, 4), (5, 6))` results in `(13, 16)` • Both operands are rectangular arrays of rank 2: The second part of the definition set of the first operand must match the first part of the definition set of the second operand, otherwise it is an error. Performs multiplication of a matrix with another matrix. e.g.: `((1, 2), (3, 4)) * ((5, 6), (7, 8))` results in `((19, 22), (43, 50))`
= >= <= <> > <	• One operand is an array, the other a scalar value: Performs the operation for every element of the array and the scalar value. The result is an array with the same definition set as the operand array. e.g.: `(1, 2, 3) >= 2` results in `(false, true, true)` e.g.: if `x` is defined as `var x[3];` then `x >= 0;` is equivalent to `x[1] >= 0; x[2] >= 0; x[3] >= 0;` • Both operands are arrays: The definition set of both arrays must be equal, otherwise an error occurs. Performs the operation for every pair of elements of the operand arrays with the same indexing tuple. e.g.: `(1, 2, 3) = (1, 2, 3)` results in `(true, true, true)` e.g.: if `x` is defined as `var x[3];` then `x >= (4, 5, 6);` is equivalent to `x[1] >= 4; x[2] >= 5; x[3] >= 6;`
== !=	Both operands can be arbitrary values or arrays. The full operands are checked for equality (or non-equality). The result is either `true` or `false`. e.g.: `(1, 2, 3) == (1, 2, 3)` results to `true` e.g.: if `x` is defined as `var x[3];` then `x == 0` results in `false` (an array of decision variables is not the same as a numeric 0)

2.1.4 Object definitions

2.1.4.1 Assignment attributes

A CMPL model essentially consists of definitions of data objects in the form of assignments. The semantics of such an assignment is controlled by attributes. These attributes can precede the assignment in any order.

Important attributes are:

object types	The value of the right-hand side of an assignment is converted to the specific object type. e.g.: `var x: real;` Calls the convert function to `var` with the data type `real` as parameter, meaning the construction of a new decision variable of the data type `real`, and assigns it to the symbol `x`.
data types	The value of the right-hand side of the assignment is converted to the given data type. e.g.: `set s := (1, 2, 3);` is equivalent to: `s := set(1, 2, 3);` Note that the data type is only used for the conversion of the right-hand side, but the symbol is not restricted to values of this data type.
const	The assigned symbol is write protected. Any try to reassign occurs an error. e.g.: `const i := 42;` Assigns symbol `i` that cannot be reassigned.
ref	Creates a reference to another symbol. e.g.: `a := 1; ref b := a; a := 2;` Then also `b` has the value 2.
public **private** **local**	Specifies the validity scope of the symbol defined in the assignment. Used primarily within code blocks.
new	Even if the assigned symbol already exists, define a new symbol hiding the original one. Used primarily within code blocks, in combination with `private` or `local` validity scope.
ordered	Execution one after the other in the order of the affected set or the definition set of the affected array in a single thread. Currently only implemented for iterations in a code block.

2.1.4.2 Sections

A section is an area of the CMPL model in which specified defaults apply to the assignment attributes. A section begins with the section header followed by a colon. All subsequent statements belong to the section until another section header starts the next section.

A section header consists of a number of assignment attributes that can be specified in any order. These attributes are then used by default for all assignments within the section. An attribute specified directly in the individual assignment overrides the section's default.

Examples

`const par:` `a := (1, 2, 3);`	Defines a parameter array of three numbers that cannot be changed afterwards.
`set:` `s := (1, 2, 3);`	Defines a set of three numbers.
`var bin:` `x, y, z;`	Defines three decision variables with the data type `bin`.

There are special assignment operators that do not respect the attributes of the section but have special default attributes. These assignment operators are:

`::=`	Works like an assignment `:=` but the default validity scope is `local` instead of `public`. The attributes from the current section are not taken into account.
`+=` `-=` `*=` `/=`	Works like `::=` but performs in addition the specific operation. e.g. `a += b;` is equivalent to `a ::= a + b;`

These special assignment operators are particularly useful for parameters used for control, in order to be able to assign them easily in sections for variables or restrictions.

2.1.4.3 Special forms of assignments

An assignment can have several left-hand sides. These left-hand sides are specified before the assignment operator, separated by commas. The assignment is carried out for each left-hand side.

An assignment can have an array on the left side and a scalar value on the right side. In this case, this value is assigned to each array element on the left side.

An assignment can be made without a right-hand side and assignment operator. A default value is then used as the right-hand side, which is `invalid` per default..

But, if an attribute determines the data type or the object type, the default value of the data type or object type is used. For example, for a numeric type this is `0`, for `string` the empty string, and for `set` the empty set.

For the object type `var`, the standard is a decision variable with the type `real[0..]`. The assignment respectively definition is done with the assignment operator `:`. For objective functions and restrictions, how-

ever, an assignment without right-hand side is not possible. If an objective function and restrictions are defined without a left-hand side, an automatic name is assigned.

2.1.4.4 Examples for definitions of parameters and variables

Examples for parameters (within a `par:` section):

`k := 10;`	Parameter `k` with value 10
`k := ( 0.5, 1, 2, 3.3, 5.5 );` `k[] := ( 0.5, 1, 2, 3.3, 5.5 );` `k[1..] := ( 0.5, 1, 2, 3.3, 5.5 );` `k[1..5] := ( 0.5, 1, 2, 3.3, 5.5 );` `n:= 1..5;` `k[n] := ( 0.5, 1, 2, 3.3, 5.5 );`	All the same vector of parameters with five elements
`A[]:= ( 16, 45.4 );`	Definition of a vector with two integer values `a[1]`=16 and `a[2]`=45.4
`a[,] := (( 5.6, 7.7, 10.5 ),` `( 9.8, 4.2, 11.1 ));`	Dense matrix with two rows and three columns
`b[] := ( 22 );`	Definition of the vector `b` with only one element.
`products := set( "bike1", "bike2");` `machineHours[products] := (5.4, 10);`	Defines a vector for machine hours based on the set `products`.
`myString := "this is a string";`	String parameter
`q := 3;` `g[1..q] := ( 1, 2, 3 );`	Parameter `q` with value 3 Usage of `q` for the definition of the parameter `g`
`x := 1(1)2;` `y := 1(1)2;` `z := 1(1)2;` `cube[x,y,z]:= ( ( (1,2),(3,4) ) ,` `( (5,6),(7,8 ) ) );`	Definition of a parameter cube that is based on the sets `x`, `y` and `z`
`a := set( [1,1],[1,2],[2,2],[3,2] );` `b[a] := ( 10, 20, 30, 40 );`	Definition of a sparse matrix `b` that is based on the 2-tuple set `a`.

Examples for decision variables (within a `var:` section):

`x: real;`	`x` is a non-negative real decision variable
`x;`	`x` is also a non-negative real decision variable (because data type `real` is the default for decision variables, if not given in the section)
`x: real[..];`	`x` is a real decision variable with no ranges
`x: real[0..100];`	`x` is a real decision variable, $0 \le x \le 100$
`x[1..5]: int[10..20];`	vector with 5 elements, $10 \le x_n \le 20 ; n \in \{1,2,...,5\}$
`x[1..5,1..5,1..5]: real[0..];`	A three-dimensional array of real decision variables with 125 elements identified by indices, $x_{i,j,k} \ge 0 ; i,j,k \in \{1,2,...,5\}$

par: prod := **set**("bike1", "bike2"); **var**: x[prod]: **real**[0..];	Defines a vector of non-negative real decision variables based on the set `prod`
y: **bin**;	x is a binary variable $y \in \{0,1\}$
par: a:=**set**([1,1],[1,2],[2,2],[3,2]); **var**: x[a]: **real**[0..];	Defines a sparse matrix of non-negative real decision variables based on the set `a` of 2-tupels.
x[1..10], y[1..5]: **real**;	Defines two vectors of real decision variables
x[1]: **real**; x[2]: **int**;	Defines `x[1]` as real decision variable, but `x[2]` as integer decision variable.
x[1..2]: (**real**, **int**);	Defines `x[1]` as real decision variable, but `x[2]` as integer decision variable.
const type: my_real := **real**[0..100]; **var**: x[1..10]: my_real;	Defines own data type for real values within the range 0 to 100 Defines decision variables of that type

2.1.5 User messages

During executing the CMPL code, outputs can be made to the console. These outputs can be used in to log the processing of CMPL. However, they cannot be used to display the optimisation result, as the optimisation only runs when the CMPL code has been completely processed and the model instance has been created.

The following functions are available for output:

echo(*a***)**	Console output of the argument value. If the argument is an array, the values are separated by space. The output is finished with a line break.
error(*s***)**	Outputs an error message with the argument string and ends the execution of the CMPL model.

All objects in CMPL have a string representation that is used for output. For decision variables and restrictions, this cannot be the corresponding result value from the optimisation, as this is not yet known at the time the output is executed. Instead, an internal representation of the CMPL object is output.

For parameters, the parameter value is displayed. In order to output numerical values in particular in the desired form, they can be converted into a string by specifying the desired formatting.

format(*f,u,…* **)**	Creates a formatted string from the values of the arguments `u,…`, using the format string *f*. The format follows the syntax of the C function `sprintf(…)`.

For each of the specified arguments `u,...` the format string must contain a formatting specification that matches the type of the argument value. Such a formatting specification has the following structure (for further details see C documentation):

```
%<flags><width><.precision>specifier
```

specifier	
d	Data type `int`
f	Data type `real`
s	Data type `string`

If the type of the corresponding argument does not match the type of the `specifier`, then the argument is converted to the matching type.

flags	
-	Left-justify
+	Forces the result to be preceded by a plus or minus sign (`+` or `-`) even for positive numbers. By default only negative numbers are preceded with a `-` sign.
width	
number	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The `width` is not specified in the `format` string, but as an additional integer value argument preceding the argument that has to be formatted.
.precision	
.*number*	For integer specifiers `d`: precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of `0` means that no character is written for the value `0`. For `f`: this is the number of digits to be printed after the decimal point. For `s`: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. When no precision is specified, the default is `1`. If the period is specified without an explicit value for precision, `0` is assumed.
.*	The *precision* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted.

Examples:

```
a:=66.77777;
echo(format("%10.2f", a));
i:=7; j:=9;
echo(format("%d of %d", i, j));
```

outputs: `66.78`

outputs: `7 of 9`

2.1.6 Code blocks

2.1.6.1 Overview

A code block is a part of the CMPL code enclosed in curly brackets. The structure in the simplest case is:

```
{ header: body }
```

The code block body consists of any other CMPL code, which can be statements or an expression. A code block header is formally always a Boolean expression. In addition to a normal Boolean expression, the definition of code block symbols can be used. In this case, the expression is considered satisfied if there is at least one valid assignment of values to the code block symbols. If there are several valid assignments, then the execution is carried out for each of these assignments, so that the code block acts as a loop.

Instead of a single header, there can also be any number of headers separated by commas:

```
{ header1, header2, …: body }
```

As far as the headers act as conditions, they must all be satisfied. As far as they represent a loop, they act as a nested loop.

The header can also be empty, in which case it acts as a condition that is always satisfied.

```
{: body }
```

A code block can be separated into several parts using `|`.

```
{ header1: body1 | header2: body2 | … }
```

In this case, the first body for which its headers are satisfied is executed. All subsequent parts are not evaluated.

A code block can be executed immediately. Or it defines an object of the data type `function` that can be assigned and called later. For this purpose, a `&` must be placed directly in front of the code block. In this case, the code block receives an array as a function parameter, accessible as `$arg`.

A code block always returns an array as a result. This result array can be used if the code block represents an expression. If it is an statement, the result array is usually an empty array and is not used.

Examples:

`{ @i in 1..3: a[i] := 2*i; }`	Assigns the value `2` to `a[1]`, `4` to `a[2]` and `6` to `a[3]`. The code block contains an iteration over `set(1..3)` and the assignment is made in the code block body
`a[] := { @i in 1..3: 2*i };`	Assigns the value `2` to `a[1]`, `4` to `a[2]` and `6` to `a[3]`. The code block constructs an array `(2, 4, 6)` with the definition `set(1..3)`. Afterwards, this array is assigned to `a`.

`sum{ @i in 1..2, @j in 1..2: x[i,j] }`	This expression is equivalent to: `x[1,1] + x[1,2] + x[2,1] + x[2,2]` The code block constructs an array `(x[1,1], x[1,2], x[2,1], x[2,2])` and then the function `sum` is called with this array as function parameter.
`sum{ [@i,@j] in [1..2,1..2]: x[i,j] }` `sum{ @t in [1..2,1..2]: x[t] }` `{ k > 0: a := 100; }`	Both expressions are equivalent to previous example. Executes the assignment to symbol `a` only if `k>0`.
`a := { k > 0: 1 | k < 0: -1 |: 0 };`	The parameter `a is` assigned the sign of `k`.
`{ k > 0: a[1] := 1; |: a[2] := 1; }`	Assignment to either `a[1]` or `a[2]` depending on `k`.
`a[{k > 0: 1 |: 2}] := 1;`	Equivalent to previous example
`my_sum := &{:` `  local a := null;` `  { @i of $arg: a := a + $arg[i]; }` `  return a;` `};`	Defines a function object equivalent to the built-in function `sum` and assigns it to `my_sum`.

2.1.6.2 Code block symbols

New symbols can be defined within a code block header. Such symbols get their value in the header and cannot be changed by any assignment. The symbols have local validity that ends with the end of the associated code block body.

In general, a header is always to be understood as a Boolean expression. If new symbols are defined in this expression, they are assigned values so that the Boolean expression is fulfilled. If the new symbol is the left-hand operand of `in` or `of`, then all possible assignments are used, resulting in an execution like in a loop.

E.g. the expression `{ @i in s.: ... }` can be understood as: `for all i which are element in set s`.

The `@` marking of a code block symbol in its definition is optional. It serves the readability of the CMPL code, and the prevention of errors by not taking into account that a symbol may already be defined in an external context.

If `i` is not defined in the outer context, then

`{ @i in s: … }`

and

`{ i in s: … }`

are identical. In both cases `i` is defined as a code block symbol and the code block body is executed for each element in `s`.

If `i` is already defined externally, then with

```
i := 1; { @i in s: … }
```

this iteration is executed as above. Inside the code block, the outer `i` is hidden by the code block symbol `i`. However, with

```
i := 1; { i in s: … }
```

no code block symbol is defined, but it is checked whether the value of the outer `i` is element of `s`, and if so, the code block body is executed once.

Although a code block header always formally represents a boolean expression, a code block symbol to be defined may not be placed anywhere in it. The following uses are permitted:

- as the left side of a comparison with `=` or `==`

  (because of the semantic similarity with an assignment, the assignment operator `:=` may then also be used instead of the comparison operator)
- as the left-hand side of the operators `in` and `of`
- within a tuple construction expression that stands in place of the simple code block symbol

Examples of code block headers

`@i = 1`	Executes the code block body once, with `i` containing the value `1`
`@i = (1, 2, 3)`	Executes the code block body once, with `i` containing the array `(1, 2, 3)`
`@i = set(1, 2, 3)`	Executes the code block body once, with `i` containing the set `set(1, 2, 3)`
`@i = set()`	Executes the code block body once, with `i` containing the empty set
`@i in set(1, 2, 3)`	Executes the code block body three times, with `i` first `1`, then `2`, then `3`
`@i in set([1,1],[2,1],[3,2])`	Executes the code block body three times, with `i` first the tuple `[1,1]`, then `[2,1]`, then `[3,2]`
`@i in set()`	The code block body is not executed. (if a next alternative code block part exists, then execution goes to it)
`[@i, 1] = [2, 1]`	Executes the code block body once, with `i` containing the value `2`
`[@i, 1] = [1, 2, 1]`	Executes the code block body once, with `i` containing the tuple `[1,2]`
`[@i, 1] = 1`	Executes the code block body once, with `i` containing the null tuple `[null]`
`[@i, 1] = [1, 2]`	The code block body is not executed. (if a next alternative code block part exists, then execution goes to it)

`[@i, 1] in set([1,1],[2,1],[3,2])`	Executes the code block body two times, with `i` first `1`, then `2`
`[@i, @j] = [1, 2, 1]`	Executes the code block body once, with `i` containing `1` and `j` containing the tuple `[2,1]` (Other assignments for `i` and `j` would be possible. CMPL selects the assignment in such a way that `rank-1` tuples are assigned from the front as far as possible.)
`[@i, 3, @j] = [1, 2, 3, 4]`	Executes the code block body once, with `i` containing the tuple `[1,2]` and `j` containing `4`
`[@i, @j] in set([1,1],[2,1],[3])`	Executes the code block body three times, first with `i=1` and `j=1`, then `i=2` and `j=1`, then `i=3` and `j=[null]`

In addition, a code block header may also consist of a stand-alone code block symbol. Such a code block symbol is not given a value and may not be used in expressions within the code block. It can only be used as a reference for `break`, `continue` or `repeat`.

2.1.6.3 Control commands in code blocks

Within a code block, special control commands can be used to set the result value of the code block and to control iterations.
Syntactically, these commands are assignments with special attributes, whereby the left-hand or right-hand side of the assignment can be missing. Generally, the left-hand side of the assignment references the code block, idendified by the first code block symbol defined there. If the left-hand side is missing, the innermost code block is affected. The right-hand side of the assignment represents the result value of the code block. If the right-hand side is missing, `null` is used as the value.

`break`	The execution of the body of the referenced code block is cancelled. Remaining statements are skipped. If the referenced code block contains iteration, the execution of the remaining iteration steps are skipped. E.g.: `{ @i of a: { a[i] = v: break i := i; } }` searches for the value `v` in the array `a` and returns the index of the first element found as result, or `null` if not found. The `i` on the left side of the break statement is necessary because the innermost code block is the comparison, but the over all `i` has to be cancelled and must therefore be named here. The `i` on the right side is the index of the found element as the result value of the code block.
`continue`	The execution of the body of the referenced code block is cancelled. Remaining statements are skipped. If the referenced code block contains iterations, remaining iteration steps are executed.

	If the referenced code block contains no iteration `continue` is equivalent to `break`. e.g.: `{ @i of a: { a[i] = v: continue i[i] := 1; } }` searches for the value `v` in the array `a` and creates an array as code block result, which contains only the indices of the values found. The first `i` in the `continue` statement references the code block, for which the result is set. The second `i` is used as normal indexation value within the code block result array.
`repeat`	The execution of the body of the referenced code block is cancelled. Remaining statements are skipped. Execution starts again with the referenced code block, but the code block result is not reinitialised. The code block headers are evaluated again. If the header conditions are not longer fulfilled, the code block body is not executed again. e.g.: `i ::= 0; p := { @r, a[++i] <> v: repeat r[i] := a[i]; };` searches for the value `v` in array `a` and returns the part of array `a` before the value found as a code block result, which is then assigned to `p`. Note that one cannot define `i` as code block symbol here, because a code block symbol cannot be be assigned or incremented. It should also be noted that the code block must define a code block symbol (`@r`) to assign a code block result.
`return`	Only allowed inside a code block used as a function definition. This command works like `break`, but refers to the innermost function definition instead of the innermost code block. Explicit referencing to a code block by specifying a left side of the assignment is not allowed. e.g.: `f := &{ a = $arg[1,], v = $arg[2]: { @i of a: { a[i] = v: return i; } } };` Defines a function `f` with two arguments. The first argument is an array, in which the value given as second argument is searched. The function value is the index of the first found element in the array, or `null` if not found.

2.1.6.4 Validity scope of symbols

Local and private symbols can be defined within a code block body. Such symbols can only be accessed within the code block body.

Unlike local symbols, private symbols nevertheless have a global lifetime. This means that when the code block body is executed again, the previous value of the symbol is accessible again. This can be used in particular to encapsulate functionality and data in the sense of object-oriented programming.

Within a directly executed code block, all symbols that are directly accessible outside the code block are also accessible inside the code block.

This applies regardless of whether the symbols are public, local or private, or whether they are code block symbols of an external code block. Within a directly executed code block, new public symbols can also be defined, which are then accessible even after the end of the code block.

In a code block used as a function, however, no external symbols are accessible at all. The only exceptions are predefined symbols, such as the data types. No new public symbols can be defined within such a code block. These restrictions apply in order to design CMPL functions as pure functions, which receive all input data via the function arguments and return all result data as function values.

Public symbols can be made accessible by writing the code block in the function definition with `&+{ ... }` instead of `&{ ... }`. Then all public symbols are accessible in the code block and new ones can be defined in it.

2.1.6.5 Validity scope of sections

An outer section continues to apply within a code block. A section started within a code block body is only valid until the end of the code block, after which the section valid before the code block becomes active again. Especially with `{:: ... }`, the defaults of the valid section can be temporarily discarded. Note the two colons, the first ends the empty code block header, the second starts a section without defaults within the code block body.

2.1.6.6 Code block as statement or expression

A directly executed code block can be used as an instruction or as an expression. Likewise, a code block body can consist of instructions or represent an expression. The following four cases can be distinguished:

code block body contains statements code block is used as statement	The statements within the code block body are executed. The result value of the code block is discarded (usually the code block has no explicit result value, which means the result value is `null`). Note that the code block itself does not need a semicolon as an end of statement, but the statements within the code block body do. e.g.: `{ @i in 1..3: a[i] := 2*i; }`
code block body contains statements code block is used as expression	The statements within the code block body are executed. Usually, one sets a result value of the code block within these instructions with `break`, `continue` or `repeat`. e.g.: `a[] := { @i in 1..3: continue 2*i; };` Note the semicolon at the end of this example. It is necessary here because the statement it ends is the assignment, not the code block itself.
code block body is expression code block is used as expression	The expression of the code block body constructs the result of the code block. e.g.: `a[] := { @i in 1..3: 2*i };` Note that within the code block body there is no semicolon, because the code block body is not a statement.

code block body is expression code block is used as statement	This case is not allowed. It would also make no sense because the code block as an statement would discard the value constructed by the expression within the code block body

A code block used as a function definition always represents an expression of the data type function. The code block body in it can be a statement or an expression:

code block used for function definition code block body contains statements	When the function is called, the statements within the code block body are executed. The code block body can construct a return value by using `return` (or `break`, `continue` or `repeat`). Otherwise the return value of the function is null. e.g.: `f := &{: return 2*$arg; };`
code block used for function definition code block body is expression	When the function is called, the expression is evaluated and forms the return value of the function. e.g.: `f := &{: 2*$arg };`

If a code block consists of several parts, then code block bodies consisting of statements and expressions may be combined with each other as desired.

2.1.6.7 Specific control structures

As already described, code blocks can be used to emulate the various control structures known from other programming languages. The most important control structures are described below.

For loop

A for loop is defined by code block with at least one iteration header. The code block body contains user-defined instructions which are repeatedly carried out. The number of repeats is based on the iteration header definition.

Examples:

`{ @i in 1(1)3 : ... }`	Loop counter `i` with a start value of 1, an increment of 1 and an end condition of 3
`{ @i in 1..3 : ... }`	Alternative definition of a loop counter; loop counter `i` with a start value of 1 and an end condition of 3. (The increment is automatically defined as 1)
`products:= set( "p1", "p2", "p3");` `hours[products]:=(20,55,10);` `{@i in products:` `  echo ("hours of product " +` `  i + " : "+ hours[i]);` `}`	For loop using the set `products` returning user messages `hours of product: p1 : 20` `hours of product: p2 : 55` `hours of product: p3 : 10`
`{@i in 1(1)2:` `  {@j in 2(2)4:` `  A[i,j] := i + j;` `  }` `}`	Defines `A[1,2] = 3`, `A[1,4] = 5`, `A[2,2] = 4` and `A[2,4] = 6`

`a := set([1,1],[1,2],[2,2],[3,2]);` `b[a] := ( 10, 20, 30 , 40);` `{ @k in a: echo (k + ":"+ b[k]); }`	`k` is iterated over the 2-tuple set `a` The following user messages are displayed: `[1, 1]:10` `[1, 2]:20` `[2, 2]:30` `[3, 2]:40`

Several loop heads can be combined. The above example can thus be abbreviated to:

`{@i in 1(1)2, @j in 2(2)4:` `  A[i,j] := i + j;` `}`	Defines `A[1,2] = 3, A[1,4] = 5, A[2,2] = 4` and `A[2,4] = 6`
`{@i in 1(1)5, @j in 1(1)i:` `  A[i,j] := i + j;` `}`	Definition of a triangular matrix

If-then clause

An if-then consists of one condition as code block header and user-defined expressions which are executed if the if condition or conditions are fulfilled. Using an alternative non-conditioned body the if-then clause can be extended to an if-then-else clause.

Examples:

`{@i in 1..5, @j in 1..5:` `  {i = j:  A[i,j] := 1; }` `  {i != j: A[i,j] := 0; }` `}` `{@i in 1..5, @j in 1..5:` `  {i = j:  A[i,j] := 1;` `  |:  A[i,j] := 0; }` `}`	Definition of the identity matrix with combined loops and two if-then clauses Same example, but with one if-then-else clause
`i:=10;` `{ i<10: echo ("i less than 10");` `  |: echo ("i greater than 9");` `}`	Example of an if-then-else clause It returns user message `i greater than 9.`
`{ i = j: 1 |: 2 }`	Conditional expression that results the value 1 if `i=j`, otherwise the value 2.

Switch clause

Using more than one alternative body the if-then clause can be extended to a switch clause.

Example:

`i:=2;` `{  i=1: echo ("i equals 1");` `  | i=2: echo ("i equals 2");` `  | i=3: echo ("i equals 3");` `  | :  echo ("any other value");` `}`	Example of a switch clause that returns user message `i equals 2.`

While loop

A while loop is defined by a code block with a condition header and using the `repeat` command within the code block body. The body contains user-defined instructions which are repeatedly carried out until the condition in the header evaluates to false.

Examples:

`i:=2;` `{i<=4:` `  A[i] := i;` `  i += 1;` `  repeat;` `}`	While loop with a global parameter that defines `A[2] = 2, A[3] = 3` and `A[4] = 4.`
`{: a ::= 1;` `  {a < 5:` `  echo (a);` `  a += 1;` `  repeat;` `  }` `}`	While loop using a local symbol defined in an outer code block that returns user messages `1` `2` `3` `4.`
`{: a ::= 1;` `  {@x:` `  echo (a);` `  a += 1;` `  {a>=4: break x;}` `  repeat;` `  }` `}`	Alternative formulation: This code block uses a reference code block symbol `x`. It is necessary, because it is needed as reference for the `break` statement in the inner code block. (Without this reference the `break` statement would refer to the condition a>=4)

Function definition

A code block can define a function. A function always has exactly one argument. Since this argument can be an array of any number of elements, this is not a restriction. The elements of the argument array can be of any data type and object type, as can the return value. They can therefore also be decision variables or constraints.

Examples:

`square := &{` `  @i of $arg: $arg[i] * $arg[i]` `};`	Defines a function that squares each element of the argument array and returns these results as a result array. e.g.: `square(3, 4, 7)` results to `(9, 16, 49)`
`square_sum := &{:` `  res ::= 0;` `  { @i of $arg:` `  res += $arg[i] * $arg[i]; }` `  return res;` `};`	Defines a function that squares each element of the argument array, sums these results, and returns the sum as the result. e.g.: `square_sum(3, 4, 7)` results to `74`

`fib := &+{` `  $arg <= 2: 1` `  \|: fib($arg-1) + fib($arg-2)` `};`	Defines a recursive function that returns the `n`th number of the Fibonacci sequence for an argument `n`. The definition with `&+{ ... }` is necessary so that the symbol `fib` defined outside the function body can be accessed inside the function body. e.g.: `fib(8)` results in `21`
`map := &{` ` @f = $arg[1], @a = $arg[2,]:` ` { @i of a: f(a[i]) }` `};`	Defines a function that receives another function as its first argument and applies that other function to the elements of the array passed as its second argument. e.g., using the fib function shown above: `map(fib, (3, 4, 7))` results in `(2, 3, 13)` e.g., using an anonymous function: `map(&{: 2*$arg}, (3, 4, 7))` results in `(6, 8, 14)`
`fixcosts := &{` `  @v = $arg[1],` `  @f = $arg[2],` `  @m = $arg[3]:` `  { f == 0: return 0; }` `  local var b := binary;` `  con v <= m * b;` `  return f*b;` `};` `obj:` `  sum{ @i of x:` `  c[i]*x[i]` `  -fixcosts(x[i], fc[i], mx)` `  } -> max;`	Defines a step-fixed cost function to be used for a decision variable. The first argument is the decision variable, the second is the step-fixed cost and the third is a large value at least as high as the upper bound of the decision variable. An additional binary variable is created in the function, as well as a constraint that sets this binary variable to 1 if the decision variable has a value greater than 0. The term representing the step-fixed costs is returned. This function can be used in an objective function. For example, in a production planning problem, let x be the vector containing the decision variables, c be the a vector of the profit contributions per unit and fc be the vector containing the associated step-fixed cost. Let also mx be a number greater than or equal to the largest upper bound of x.

2.1.6.8 Multithreading

The option -threads n can be used to determine how many threads CMPL may use. If more than one thread is allowed, then a maximum of this many threads are used for parallel execution for iterations in a code block.

If the iteration steps are to be executed sequentially in their order, the ordered attribute must be used in the definition of the code block symbol for the iteration.

Examples:

`{ @i in 1..10: echo(i); }`	When using multiple threads, the output is in unordered sequence.
`{ ordered @i in 1..10: echo(i); }`	Regardless of how many threads are allowed, the output is guaranteed to be in order from 1 to 10.

Caution: Multithreading is an experimental feature. Errors may occur when using it. Therefore, the number of threads is currently set to 1 by default.

2.1.7 Names for rows and columns

2.1.7.1 Name prefix

When defining decision variables or constraints or objective functions, the name and index tuple of the CMPL symbol are used to name the column or row in the LP problem matrix. If a constraint is not assigned to a CMPL symbol, it is automatically given a name in the LP problem matrix. This behaviour can be adjusted by a name prefix.

Usage:

`` `*nameprefix* { … } ``	Sets the name prefix effective for the execution of the code block.
`` `*nameprefix statement*; ``	Sets the name prefix effective for the execution of the statement. Especially useful if the statement is a function call.

If a name prefix is used in the definition of a decision variable or a restriction, then:

- The name for the column or row is composed of the name prefix and the CMPL symbol name. If a constraint is not assigned to a CMPL symbol, then the name prefix alone forms the name.
- If the index tuple for the name of the column or row is composed of the current values of all surrounding iterations and the index tuple of the CMPL symbol.

Name prefixes can be nested. The effective name prefix is then composed of all specified name prefixes. A command line option can be used to determine whether a separator string is placed between the parts in this composition.

Examples:

`var:` `  x[1..3], y[1..3];` `con:` ` `` `aaa { @i in 1..3: x[i] <= y[i]; } ``	The 3 constraints get the names `aaa[1]`, `aaa[2]` and `aaa[3]`
`var:` `  x[1..3], y[1..3];` `con:` ` `` `aaa {: cc: x <= y; } ``	The 3 constraints get the names `aaacc[1]`, `aaacc[2]` and `aaacc[3]`

`var:` `  x[1..2, 3..4], y[1..2, 3..4];` `con:` `  `aaa { @i in 1..2:` ` `bbb { @j in 3..4: x[i,j] <= y[i,j]; };` `  }`	The 4 constraints get the names `aaabbb[1,3]`, `aaabbb[2,3]`, `aaabbb[1,4]` and `aaabbb[2,4]`
`var:` `  `a x[1..2] := int;`	Definition of two variables with CMPL symbol name `x[1]` and `x[2]`, but column name in the LP problem matrix is `a[1]` and `a[2]`. Note the assignment operator `:=` if you use `:` then the column names would be `ax[1]` and `ax[2]`
`fct := &{: local var x[1..2]; … };` `{ @i in set("a", "b"):` `  `f1 fct();` ` `f2 fct();` `}`	Defines a function which uses two decision variables `x[1]` and `x[2]`. The function is called 4 times. Without a name prefix it would be an error because the use of the same names for different columns in the LP problem matrix. Using a name prefix the columns get the names `f1x[a,1]`, `f1x[a,2]`, `f2x[a,1]`, `f2x[a,2]`, `f1x[b,1]`, `f1x[b,2]`, `f2x[b,1]`, `f2x[b,2]`

For compatibility with the previous version of CMPL, the name prefix before a code block may also be specified without an introductory `. However, this is only possible if the name prefix does not correspond to a defined CMPL symbol, as otherwise the construct would syntactically correspond to a function call.

2.1.7.2 Explicit control of the name prefix

The currently effective name prefix can be obtained within CMPL with `$curDestName`. It is also possible to set or delete the name prefix:

`$curDestName`	Special symbol for reading and setting the currently effective name prefix. When reading, the effective name prefix is returned as a string, regardless of whether it was set with `` `nameprefix `` or was previously set with `$curDestName`. If no name prefix was set, `null` is returned. When set, the new value remains effective until the end of the innermost code block body in which the setting is executed or, if necessary, until it is set again within the code block body. The effectiveness also extends to called functions.

With regard to the index tuple that becomes effective when the name prefix is used for name generation, there is a difference between setting the name prefix with `` `nameprefix `` and setting it via `$curDestName`. If setting via `$curDestName` is relevant for the effective nameprefix, then only the current values of iterations started after setting with `$curDestName` are included in the index handle.

Other special symbols in this context are:

`$curTuple`	Gets the current tuple of the innermost iteration
`$curFullTuple`	Gets the current tuple of all iterations
`$curDestTuple`	Gets the tuple of all iterations up to the innermost iteration, in which `$curDestName` is set. If `$curDestName` is never set, then equivalent to `$curFullTuple`. This tuple prefixes the index tuple in names for new columns or lines in the LP problem matrix.

Examples:

`var:` `  x[1..3], y[1..3];` `con:` `  { @i in 1..2:` `  $curDestName ::= "a" + i;` `  x[i] <= y[i];` `  }`	The 3 constraints get the names `a1`, `a2` and `a3` Note the assignment with `::=` to `$curDestName`. This is necessary, because in an assigment with `:=` within a `con` section it would be tried to convert the right hand side of the assignment to a constraint.
`var:` `  x[1..3], y[1..3];` `con:` `  `a { @i in 1..2:` ` $curDestName += i;` ` x[i] <= y[i];` ` }`	Also the 3 constraints get the names `a1`, `a2` and `a3`
`echo($curDestName);` `{ @i in 1..1:` `  $curDestName += "a";` `  echo($curDestName);` `  { @j in 2..2:` `  $curDestName += "b";` `  `c { @k in 3..3:` ` echo($curDestName);` ` echo($curDestTuple);` ` echo($curFullTuple);` ` }` ` }` ` echo($curDestName);` `}`	prints: `null` prints: `a` prints: `abc` prints: `[3]` prints: `[1,2,3]` prints: `a`

2.1.7.3 Explicitly set the name for rows and columns

The name, including the index multiple for rows and columns in the LP problem matrix, can also be set completely independently of the name in CMPL. The following special symbols can be used for this purpose:

o.**$destName**	Gets or sets the name for the column or line in the LP problem matrix represented by *o*, which must be a scalar value of object type `var`, `con` or `obj`. e.g. if `var: x[1..5];` then `x[1].$destName` results to `"x"` and `x[1].$destName := "X#";` change its name in the LP problem matrix to `X#[1]`.
o.**$destTuple**	Gets or sets the index tuple for the column or line in the LP problem matrix represented by *o*, which must be a scalar value of object type `var`, `con` or `obj`. e.g. if `var: x[1..5];` then `x[1].$destTuple` results to `[1]` and `x[1].$destTuple := ["a", "b"];` change its name in the LP problem matrix to `x[a,b]`.
o.**$destNameTuple**	Gets or sets the name and the index tuple (both together within a tuple) for the column or line in the LP problem matrix represented by *o*, which must be a scalar value of object type `var`, `con` or `obj`. e.g. if `var: x[1..5];` then `x[1].$destNameTuple` results to `["x", 1]` and `x[1].$destNameTuple := "X#1";` change its name in the LP problem matrix to `X#1`.
o.**$destFullName**	Gets the name and the index tuple (together as a string) for the column or line in the LP problem matrix represented by *o*, which must be a scalar value of object type `var`, `con` or `obj`. e.g. if `var: x[1..5];` then `x[1].$destFullName` results to `"x[1]"`

2.1.8 Extensions of CMPL

2.1.8.1 Logical constraints

In a linear optimisation model, a constraint consists of an inequality or an equation. In CMPL, however, it is also possible to specify any logical combination of equations and inequalities as a constraint. Such a construct is then suitably transformed using automatically added binary variables.

Relevant operators and functions:

`&&`	AND operator: Both equations or inequations must be satisfied.
`and( … )`	AND function: All equations or inequations that are given as an argument array have to be satisfied.
`||`	OR operator: Only one of the two equations or inequations linked must be fulfilled.
`or( … )`	OR function: Only one equations or inequations that are given as an argument array have to be satisfied.
`!`	Negation: the equation or inequality must not be fulfilled.

`( … )`	Bracketing can be used to form arbitrarily complex logical constructs.
`<>` `<` `>`	These comparison operations are not possible in a linear optimisation model. But CMPL allows them by automatically considering them as negations. For example, CMPL processes `x < y` as `!(x >= y)`.

In addition to equations and inequations, single binary variables can also be used in logical operations as Boolean values. If `b` is defined as a binary variable, then the use of b in place of an equation or inequation in a logical operation is considered to be the equation `b = 1`.

Examples:

`var: x, y: real;` `con: x >= y || x = 0;`	If `x` is greater than 0, then it must also be greater than or equal to the value of `y`.
`var: x: real;` `con: or(x = (1, 2.4, 5.6));`	The variable `x` may only take the values `1, 2.4 or 5.6`. To do this, an array of three equations is first formed, which are then linked with a logical OR.
`var: x, y: real; b: bin;` `con: (b && x > y) || (!b && x < y);`	If the variable `b` is equal to the value 1, then `x` must be greater than `y`. If, on the other hand, `b` takes the value 0, then `x` must be less than `y`.

2.1.8.2 Products of decision variables

In CMPL, products of decision variables can be used. These can either be passed directly to the solver if the solver supports quadratic optimisation QP (Cplex, Gurobi, Scip). Or they are linearised by CMPL if in one of the operands is binary or integer.

Examples:

`var: x: real; b: bin;` `con: x*b <= 10;`	Product with a binary variable
`var: a, b: int[0..5];` `con: (a + b)^2 <= 100;`	Products of integer variables
`prod := &{:` `  p ::= 1;` `  { i of $arg: p *= $arg[i]; }` `  return p;` `};` `var: a[1..5]: int[0..3];` `con: prod(a) <= 100;`	Definition of a function that multiplies all elements of its argument array with each other. This function can be applied to use the product of the variables of an array in a restriction.

2.1.8.3 Container values and class-like constructs

In CMPL, the special data type container is available. Data objects of this type do not directly contain values themselves, but instead subordinate symbols that can contain any values, arrays, or other containers. The subordinate symbols are addressed via a point as operator .

Examples:

`myfunctions := container();`	Similar to namespace: Defines a new container object and assigns it to the symbol `myfunctions`.
`myfunctions.fct1 := &{ … };` `myfunctions.fct2 := &{ … };`	Defines functions in the container.
`myfunctions.fct1(…);`	Calls a function.
`container c[1..3];`	Similar to struct in C: Defines three new container objects and assigns them to an array `c`.
`c[1].a := (1, 2, 3);` `c[2].a := (3, 4, 5);` `c[3].a := (6, 7, 8);`	Defines a child symbol `a` in each of the containers and assigns an array to it.
`echo(c[2].a[3]);`	Accesses the second container, therein the third element of the array `a` and outputs the value 5.

CMPL also offers possibilities for the class-like use of container objects, as known from object-oriented programming. The following language elements are available:

`class.construct( … );`	Function for creating a class-like container object. A constructor-like function is to be given as an argument to this function, which defines the instance variables and instance functions of the class. If a second argument is given, this is passed on as an argument to the constructor function.
`$this`	Access to the container that contains the instance of the class. Access to instance variables and instance functions must always take place via it.
`as_string`	If a class defines an instance function with this name, it is implicitly called when the container is converted to a string. This is particularly useful to be able to simply output a suitable textual representation of the class object with `echo()`.
`as_var`	If a class defines an instance function with this name, it is implicitly called when the container is converted to decision variables.
`as_con`	If a class defines an instance function with this name, it is implicitly called when the container is converted to a constraint.
`as_obj`	If a class defines an instance function with this name, it is implicitly called when the container is converted to an objective function.

In the following, a class for the Fibonacci sequence is given as an example. The sequence calculated so far is stored in the class. If an element is queried that has not yet been calculated, the sequence is extended.

```
// constructing function for the class
fibcl := &{:
  private par:
    // stored values, initialized with first two elements
    $this._fib[1..2] := 1;
    // count of computed elements
    $this._maxind := 2;
    // name for this object, given as constructor argument
    $this._name := $arg;

    // function to compute values up to given element number
    $this.compute := &{ $arg > $this._maxind:
      { i in ($this._maxind+1) .. $arg:
            $this._fib[i] := $this._fib[i-1] + $this._fib[i-2]; }
      $this._maxind := $arg;
    };

  public par:
    // function to get value for given element number
    $this.get := &{:
      // if element is not stored yet then compute it
      { $arg > $this._maxind: $this.compute($arg); }
      return $this._fib[$arg];
    };

    // function to get info string
    $this.as_string := &{:
      "Fibonacci " + $this._name + " computed up to element " + $this._maxind
    };
};

// construct two objects of the class
fibobj1 := class.construct(fibcl, "Fib1");
fibobj2 := class.construct(fibcl, "Fib2");

// get value of element number 20, outputs 6765
echo(fibobj1.get(20));
// outputs info string for the class object
echo(fibobj1);
// outputs info string for the second class object
echo(fibobj2);
```

2.1.8.4 Special ordered sets

Classes for SOS and SOS2 are predefined in CMPL. The following functions are available for creating the objects for these:

`sos.sos1();`	Constructs an object for one new SOS1. Returns the container object representing the SOS. Decision Variables must be added subsequently to this object by member function `add`.
`sos.sos1(`*var1*`, `*var2*`, … );`	Constructs an object for one new SOS1 over the given decision variables. Returns the container object representing the SOS.
`sos.sos1(`*ds*`, `*tp*`);`	Constructs an object for a new SOS1 with new decision variables to be created. An array of decision variables is created via the definition set `ds` with the data type `tp`. Returns the container object that represents the SOS.
`sos.sos2();` `sos.sos2(`*var1*`, `*var2*`, … );` `sos.sos2(`*ds*`, `*tp*`);`	Same construction functions for a new SOS2

The objects have the following member functions:

`name`	Sets the name that is used in the linearisation of the SOS. The argument is a string. For SOS2 it can also be two strings, the second one being used in the linearisation of the sequence restriction of the SOS2. Returns the container itself.
`add`	Adds decision variables to the SOS. One or more decision variables can be passed as arguments. Returns the container itself.
`as_var`	Returns the decision variables from the SOS. This function is not called directly, but is used to be able to use the SOS object itself in variable definitions.
`ord`	Returns a consecutively assigned number of the SOS object.

Examples:

`var:` `  x[]: sos.sos1([1..5], real[0..100]);`	Creates an SOS of five new variables.
`var:` `  y[]: sos.sos2([1..10], real).name("test");`	Creates an SOS2 of ten new variables with the name `test`.
`var:` `  a, b, c: int[0..100];` `  sos.sos1(a, b, c).name("test2");`	Defines 3 variables and then create a new SOS over these variables and give a name to the SOS.
`var:` `  xm[1..5, 1..10]: real;` `par:` `  { i in 1..5:` `  s[i] := sos.sos1();` `  s[i].name("SOS_row_" + i);` `  s[i].add(xm[i,]);` `  }`	Defines a matrix of variables. For every row of the matrix an SOS is created with the `name SOS_row_1, SOS_row_2, …` . Each variable in this row are added to the SOS.

CMPL handles SOS in two ways. If the solver invoked does not support special ordered sets directly then the SOS are linearised in the form of suitable constraints. Otherwise the SOS are passed directly to the solver (e.g. Cplex, Gurobi, Cbc and Scip) via the generated Free-MPS file.

2.1.8.5 Other model reformulations

CMPL performs the following simple model transformations by default:

- Constraints without decision variable

  This case can occur if the constraint actually contains no variable or the variables are multiplied by parameters equal to zero. In addition, it is possible that logical operations in a constraint show that the satisfaction of the constraint does not depend on the specified decision variables. In these cases, the constraint is trivially always satisfied or can never be satisfied.

  Such a constraint is automatically supplemented by an additional decision variable so that it can be included in the LP problem matrix and appears in the result. If the restriction can never be satisfied, this decision variable is restricted accordingly.

  Alternatively, CMPL can remove trivially always satisfied constraints and issue an error message for a constraint that can never be satisfied.

- Constraints with only one decision variable

  A constraint with only one decision variable can be replaced by a bound for this decision variable.

  By default, this is only done for unnamed constraints. Named constraints, on the other hand, remain unchanged so that they can be included as a row in the LP problem matrix and appear in the result.

  Alternatively, either all or none of such constraints can be replaced by bounds.

- Decision variables not used in any constraint

  If a decision variable is defined but not used in any constraint, this decision variable has no meaning for the optimisation and is not given a value.

  By default, CMPL removes such decision variables. However, if the decision variable does not appear in any constraint only because such constraints were removed by the previous transformations, an additional constraint is created for the variable instead so that the variable is included as a column in the LP problem matrix and appears in the result.

  Alternatively, either all decision variables not used in constraints can be omitted or an additional constraint is created for all of these decision variables.

2.1.9 Short Language reference

Attributes

public **private** **local**	Specifies the validity scope of the symbol defined in the assignment
const	The symbol is write protected.
ref	Creates a reference to another symbol
new	Even if the assigned symbol already exists, a new symbol is defined that hides the original symbol.
ordered	Ordered execution without parallel threads. Currently only effective for iterations within a code block.
extern	Assigns values from an external source. Mainly for internal use
assert	Assert condition for symbol definition. Mainly for internal use
declare	Declaration of symbol name
initial	Performs an assignment only the first time.
break **continue** **repeat** **return**	Control commands for code blocks

Literal values

number	Value of data type `real` or `int`.
"string"	Value of data type `string`. If the string contains double quotes, they have to be escaped with `\"`.
true **false**	Literal values of data type `bin`

Special values

inf	Infinite value of data type `real`
invalid	Marker for a non existing value
null	Empty array
(omitted value)	• After unary operators * and /: Converted to the set of all index tuples with rank 1. • Within tuple construction: Converted to the full set of all possible index tuples of all ranks. • Within array construction: Marks a non existing element (equivalent to `null`). • Within interval construction: Converted to the infinite value (equivalent to `-inf` (on the left side of operator `..`) or to `inf` (on the right side of `..`)

Object types (also usable as convert functions)

var **variables**	Decision variables (columns within the linear programming model)
obj **objectives**	Objective functions (neutral rows within the linear programming model)
con **constraints**	Constraints (restricted rows within the linear programming model)
par **parameters**	Everything else …

Data types (also usable as convert functions)

real	Floating point number (uses internally C data type `double`) The literal value consists of digits, decimal point and optional exponent.
int	Integer number (uses internally C data type `long`).
integer	The literal consists only of digits.
bin	Numeric value that can only be 0 or 1 (subtype of `int`)
binary	It can also be used as boolean value. The literal values are `true` (value 1) and `false` (value 0).
numeric	Union type for `real` and `int`
formula	An expression of parameters and decision variables Note that a formula is not a constraint, but a suitable formula can be converted into a constraint.
string	Character string Th literal value is enclosed in double quotes.
indexpart	Union type for `int` and `string`
interval	Interval between two numeric values If one or both bounds are omitted, the interval on this side is unbounded.
tuple	Tuple of an arbitrary number of elements (also no element) with any data type A special kind of tuple is an index tuple, which consists only of elements of the data type `indexpart`.
set	Set of an arbitrary number of elements (also no element). All elements must be index tuples.
function	Function object, constructed by `&{ … }`
container	A value that contains other symbols (similar to struct or class in C)
type	Data type
objecttype	Object type

Assignment operators (assignments can only be used as statements, but not as part of other expressions)

u **:=** *v*;	Declares symbol *u*, if not already declared. Assigns value *v* (converted according to the attributes) to the symbol *u*.
u **:** *v*;	Declares symbol *u*, if not already declared. Assigns value *v* (converted according to the attributes) to the symbol *u*. The object type of converted value *v* must be `var`, `obj` or `con`. The row or column in the resulting LP problem matrix is named according symbol *u*.

	(Besides being used as an assignment operator, the colon is also used as a separator).
u **::=** *v*;	Declares symbol *u* with `local` validity scope, if not already declared. Assigns value *v* (without conversion according to the attributes) to the symbol *u*.
u **+=** *v*; *u* **-=** *v*; *u* ***=** *v*; *u* **/=** *v*;	Performs given operation (+, –, * or /) on values *u* and *v* (without conversion according to the attributes). Assigns the result to symbol *u*.
u;	Assignment without given right hand side. Declares symbol *u*, if not already declared. A default value of an object type is assigned to the symbol *u*. If not specified by attribute the default value for `par` is `invalid` and the default value for `var` is `real`. Other object types do not have a default value. For object type `var` the column in the resulting LP problem matrix is named according symbol *u*.

Increment and decrement operators

++*u* **--***u*	Increments or decrements the value of symbol *u* and then gives the resulting value. The symbol *u* must be a symbol with a scalar `int` value. This operation is guaranteed to be atomic when used with multi-threading, while an assignment such as `u += 1;` is not guaranteed to be atomic.
*u***++** *u***--**	Gives the current value of symbol *u* and then increments or decrements the value of the symbol. *u* have to be a symbol with a scalar `int` value. This operation is guaranteed to be atomic when used with multi-threading, while an assignment such as `u += 1;` is not guaranteed to be atomic.

Computational operators

u **+** *v*	Adds both operands. Operands can be: • `numeric` or `formula`: numerical addition • `string`: string concatenation • `set`: set union • `null`: the other operand is the result If the operands are arrays, the operation is performed for each element.
+*u*	Positive sign for the operand. Operand can be: • `numeric` or `formula`: numerical sign • `null`: the result is `null` If the operand is an array, the operation is performed for each element.
u **-** *v*	Subtract the second operand from the first. Operands can be: • `numeric` or `formula`: numerical addition • `set`: set of all elements of *u* which are not contained in *v* • `null`: the other operand is the result If the operands are arrays, the operation is performed for each element.
-*u*	Negative sign for the operand. Operand can be:

	• `numeric` or `formula`: numerical sign • `null`: the result is `null` If the operand is an array, the operation is performed for each element.
u ***** *v*	Multiplication of both operands. Operands can be: • `numeric` or `formula`: numerical multiplication • `set`: set intersection
*****u	Converts value `u` to a set. Value must be `indexpart` or an index tuple (or already a `set`). e.g. `*1` results to `set(1)` If it is used before a bracket, then the constructed array receives a definition set `[1..]`, even if it contains only one element. E.g. `*(7)` results in an array with one element and definition set `*1`.
u **/** *v*	Division. Both operands must be `numeric`. The data type of the result value is always `real`, even if both operands are `int`.
/*u*	Operand must be a `set` (or already an `indexpart`). If the value is a `set` with only one element, it is converted to that element. If the value is another `set`, then it is marked, so that in a match operation or an indexation operation with this set the corresponding part of the index is removed from the result. e.g. `set([1,1], [2,3], [4,2]) *> [*, /set(1,2)]` results to `set(1, 4)` If it is used before a bracket, then the constructed array gets a definition set `[*null]`. e.g. `/(7,)` results to an array with one element and definition set `[*null]`.
u **^** *v*	To the power of. The second operand must be `numeric`. The first operand can be `numeric` or `formula`. If first operand is a `formula`, then second operand must be a non-negative `int`.
*a***^T**	Transpose an operand array. Only for use in matrix multiplication. If `T` follows directly after a closing square bracket, then `^` can be omitted (e.g. `a[]T` is equivalent to `a[]^T`).

Comparison operators

u **=** *v*	Equal to
u **>=** *v*	Greater than or equal to
u **<=** *v*	Less than or equal to
u **<>** *v*	unequal
u **>** *v*	Greater than
u **<** *v*	Less than
u **==** *v*	Total equality. The full operands are checked (not the elements of arrays), result is scalar `bin`.
u **!=** *v*	Negated total equality. The full operands are checked (not the elements of arrays), result is scalar `bin`.

Logical operators

u **&&** *v*	Combines both operands by logical And. Operands must be convertible to `bin`, or be a `formula` with a boolean value.
u **||** *v*	Combines both operands by logical Or. Operands must be convertible to `bin`, or be a `formula` with a boolean value.
!*u*	Logical negation of the operand. Operand must be convertible to `bin`, or be a `formula` with a boolean value.

Construction operators

(*u*, *v*, … **)**	Array construction from elements. There can be any number of elements, including zero.
[*u*, *v*, … **]**	Tuple construction from elements. There can be any number of elements, including zero.
u`..`*v*	Interval construction between lower and upper bound. Bounds must be `numeric` (or `inf`). One or both bounds can be omitted.
*u***(***v***)***w*	Constructs a set with elements from *u* to *w* with increment/decrement *v*. All operands must be `int`. e.g. `1(3)10` results to `set(1, 4, 7, 10)`

Set and array operators

s1 ***>** *s2*	Matching operation. Performs the intersection between both sets, and removes then such parts from the tuples of the result set, which correspond to parts of *s2*, which are no set or are marked with the unary `/` operator.
t **in** *s*	Checks whether an index tuple *t* is element of the set *s*, result is `bin`.
@t **in** *s*	For all tuples *t* in set *s*. Only usable as a code block header.
t **of** *a*	Checks whether an index tuple *t* is element of `validset(`*a*`)`, result is `bin`.
@t **of** *a*	For all tuples *t* in `validset(`*a*`)`. Only usable as a code block header.

Optimisation sense operator

f **->** *d*	Specify the optimisation sense for the formula *f*. Only the values `min` and `max` are permitted. The result is a formula, which can be used as objective function.

Empty operator (two expressions directly adjacent, the operation is chosen by the token type)

n s	*n* is a literal number, and *s* a symbol: equivalent to *n*s*
f **(** … **)**	Function call: The second expression constructs an array, then the function *f* is called with that array as argument.
f **{** … **}**	Function call: The code block given as second expression is evaluated, then

	function *f* is called with the code block result as argument.
a **[…]**	Indexing: The second expression constructs an index tuple or a tuple set, then array *a* is indexed with that tuple or set. But if the value of the expression *a* is a scalar data type, then the tuple is used for data type restrictions, instead of indexing. In this case, the tuple doesn't need to be an index tuple, but must be suitable to restrict the data type.
[…] *a*	Array cast: The first expression constructs an index tuple or a tuple set, then the definition set of array *a* is changed to this tuple or set.

Other syntactic elements: code blocks

{ … }	Includes a code block.
|	Separates the parts of a code block
&	Only permitted directly before a code block: The code block is not evaluated directly, but a function pointer to the code block is given.
&+	Like **&**, but the code block gets access to public symbols.

Other syntactic elements: separators

;	Completes a statement
,	Separates elements of lists: • elements in array construction • elements in tuple construction • multiple left sides in an assignment • multiple code block headers
:	Ends a header: • section header • code block header
.	Separates container value from contained part. To the left of this must be a value that contains parts. Such a value is either of data type `container` or of object type `var`, `obj` or `con`. To the right of it must be the name of the contained part.

Other syntactic elements: symbol markers

@*s*	Only usable in code block header: Marks symbol as a new defined code block symbol. Needed only if `s` is already a defined symbol.
s	Can only be used with code block control commands: Marks that the symbol is to be used as a reference for the code block (instead of using it as an expression). Only needed in rare cases when without it both interpretations would be syntactically correct.
`*s*	Can only be used directly before a code block. It marks that `s` is to be used

	as name prefix for naming of rows and columns in the LP problem matrix (instead of using it as function name). Only needed if `s` is already a defined symbol.

Comments

// **#**	comment up to end of line
/* … */	comment between /* and */

Special symbols

$arg	Returns argument array within a function
$this	Returns container object in which a member function is called
$curTuple	Returns the current tuple of the innermost iteration
$curFullTuple	Returns the current tuple of all iterations
$curDestName	Returns or sets the current name prefix for new columns or lines in the LP problem matrix. A setting is only effective up to the end of the current innermost code block, then the previous value is restored.
$curDestTuple	Returns the tuple of all iterations up to the innermost iteration, in which `$curDestName` is set. This tuple prefixes the index tuple in names for new columns or lines in the LP problem matrix.
o.**$destName**	Returns or sets the name for the column or line in the LP problem matrix represented by *o*, which must be a scalar value of object type `var`, `con` or `obj`. e.g. if `var: x[1..5];` then `x[1].$destName` results to `"x"` and `x[1].$destName := "X#";` changes its name in the LP problem matrix to `X#[1]`.
o.**$destTuple**	Gets or sets the index tuple for the column or line in the LP problem matrix represented by *o*, which must be a scalar value of object type `var`, `con` or `obj`. e.g. if `var: x[1..5];` then `x[1].$destTuple` results to `[1]` and `x[1].$destTuple := ["a", "b"];` changes its name in the LP problem matrix to `x[a,b]`.
o.**$destNameTuple**	Gets or sets the name and the index tuple (both together within a tuple) for the column or line in the LP problem matrix represented by *o*, which must be a scalar value of object type `var`, `con` or `obj`. e.g. if `var: x[1..5];` then `x[1].$destNameTuple` results to `["x", 1]` and `x[1].$destNameTuple := "X#1";` changes its name in the LP problem matrix to `X#1`.
o.**$destFullName**	Gets the name and the index tuple (together as a string) for the column or

	line in the LP problem matrix represented by *o*, which must be a scalar value of object type `var`, `con` or `obj`. e.g. if `var: x[1..5];` then `x[1].$destFullName` results to `"x[1]"`
o.**$objectType**	Returns the object type of symbol or expression *o*.
o.**$dataType**	Returns the data type of symbol or expression *o*.
o.**$typeBase**	Returns the data type without type parameter of symbol or expression *o*.
o.**$typePar**	Returns the type parameter tuple from the data type of symbol or expression *o*. If the data type has no type parameter, it results to `null`.

Built-in functions for aggregating values

sum(*a***)**	Calculates a sum over all values of the argument array. An operation is performed by the operator +, depending on the data type of the operands.
max(*a***)** **min(***a***)**	Gives the maximum or the minimum value of the values in the argument array. All values in the array must be numeric or interval. If a value is interval, its upper (max) or lower (min) bound is used.
max **min**	These function names are also used for the objective sense at the end of an objective function.
and(*a***)** **or(***a***)**	All values of the argument array are combined by logical and or logical or. All values must be either convertible to `bin`, or be a formula with boolean value.

Built-in functions for output

echo(*a***)**	Console output of the argument value. If the argument is an array, the values are separated by space.
error(*s***)**	Outputs an error message with the argument string and ends execution.
format(*f,u,…* **)**	Creates a formatted string from the values of the arguments *u,…*, using the format string *f*. This is done by the C function sprintf, see its documentation for the format string.

Built-in functions for sets and arrays

len(*s***)**	Argument can be a set, a string, or an array of set or string values. Gives the count of elements in the set (`inf` for an infinite set) or the count of characters in the string. If the argument is an array, it is done for every element of the array and gives the results also as an array.
rank(*s***)**	Returns the rank of the argument value. For a set of tuples with different ranks the result is an interval. For values other than tuple or set the result is ever 1. If the argument is an array, it is done for every element of the array and gives the results also as an array.
defset(*a***)**	Returns the definition set of array *a*
validset(*a***)**	Returns the set of all index tuples of array *a*, for which a value exists in the array (i.e. for which the value is not `invalid`)

	If the array contains only valid values, then `validset` results the same as `defset`.
valid(*a***)**	Checks if all elements of the array are valid. Equivalent to `validset(a)==defset(a)`
def(*a***)**	Counts the elements of the array. Equivalent to `len(defset(a))`
count(*a***)**	Counts the valid elements of the array. Equivalent to `len(validset(a))`
array(*s***)**	Argument can be a set or a tuple. Returns an array of the elements of the set or the parts of the tuple. e.g. with tuple: `array([1..2, 1])` returns `(1..2, 1)` e.g. with set: `array(set([1..2, 1]))` returns `([1,1], [2,1])`

Built-in mathematical functions

dim(*a***)**	Gives the first part of the last tuple of the definition set of the argument array. For instance if `a` has the definition set `[1..3, 1..5]`, then `dim(a)` returns 3, and `dim(a[1,])` returns 5.
div(*c***,***d***)** **mod(***c***,***d***)**	Integer division or remainder of integer division. Both arguments must be scalar integer numbers.
srand(*x***)**	Initialisation of a pseudo-random number generator using the argument *x*. The argument must be a scalar number and is converted to int. Returns the value of the argument x.
rand(*x***)**	Returns an integer random number in the range 0<= rand < x. The argument must be a scalar number and is converted to int.
sqrt(*x***)**	Square root function: The argument must be a scalar number and is converted to real.
exp(*x***)**	Exp function: The argument must be a scalar number and is converted to real.
ln(*x***)**	Natural logarithm: The argument must be a scalar number and is converted to real.
lg(*x***)**	Common logarithm: The argument must be a scalar number and is converted to real.
ld(*x***)**	Logarithm to the basis 2: The argument must be a scalar number and is converted to real.
sin(*x***)**	Sine function measured in radians: The argument must be a scalar number and is converted to real.
cos(*x***)**	Cosine function measured in radians: The argument must be a scalar number and is converted to real.
tan(*x***)**	Tangent function measured in radians: The argument must be a scalar number and is converted to real.
acos(*x***)**	Arc cosine function measured in radians: The argument must be a scalar number and is converted to real.
asin(*x***)**	Arc sine function measured in radians: The argument must be a scalar number and is converted to real.
atan(*x***)**	Arc tangent function measured in radians: The argument must be a scalar number and is converted to real.
sinh(*x***)**	Hyperbolic sine function: The argument must be a scalar number and is converted to real.

cosh(*x***)**	Hyperbolic cosine function: The argument must be a scalar number and is converted to real.
tanh(*x***)**	Hyperbolic tangent function: The argument must be a scalar number and is converted to real.
abs(*x***)**	Absolute value: The argument must be a scalar number.
ceil(*x***)**	Smallest integer value greater than or equal to a given value. The argument must be a scalar number.
floor(*x***)**	Largest integer value less than or equal to a given value. The argument must be a scalar number.
round(*x***)**	Simple round: The argument must be a scalar number.

Class support

class	Namespace for related functions
class.construct(*f,a***)**	Constructs a new object of the class, using the class constructor function *f*, which is given *a* as argument.
class.runat(*c,f,a***)**	Calls the function *f* with argument *a*, with $this within the function set to the container *c*.
class.copy(*c***)**	Creates a copy of the container *c*, by assignments all of its elements to the new container.
class.refcopy(*c***)**	Like `class.copy`, but all assignments of the elements are made using `ref`.
class.finalize(*c***)**	Marks the container *c* as final, which prohibits the creation of new elements in the container.
*c***.as_string**	If an element `as_string` is defined in the container *c*, then it is called as function when the container is converted to a string, for instance in `echo(c);`
*c***.as_var**	If an element `as_var` is defined in the container *c*, then it is called as function when the container is converted to decision variables, for instance by using it within a `var` section.
*c***.as_obj**	If an element `as_obj` is defined in the container *c*, then it is called as function when the container is converted to an objective function, for instance by using it within a `obj` section.
*c***.as_con**	If an element `as_con` is defined in the container *c*, then it is called as function when the container is converted to constraints, for instance by using it within a `con` section.

SOS support

sos	Namespace for related functions
sos.sos1()	Constructs an object for one new SOS. Returns the container object representing the SOS. Decision Variables must be added subsequently to this object by member function `add`.
sos.sos1(*v1,v2,…* **)**	Constructs an object for one new SOS over the given decision variables. Returns the container object representing the SOS.
sos.sos1(*ds,tp***)**	Construct an object for one new SOS with new created decision variables. An

	array of decision variables with definition set *ds* is created, all having the data type *tp*. Returns the container object representing the SOS.
sos.sos2() **sos.sos2(***v1*,*v2*,… **)** **sos.sos2(***ds*,*tp***)**	Same constructing functions for SOS2
*c***.name(***s***)**	Assigns the name `s` to the SOS object `c`.
*c***.add(***v1*,*v2*,… **)**	Adds the given decision variables to the SOS object `c`.
*c***.ord()**	Gives the internal number of the SOS object `c`.
*c***.as_var()**	Gives the decision variables belonging to SOS object `c`. Implicitly called if the SOS object is used within a `var` section.

2.2 CMPL Header

2.2.1 CMPL Header elements

A CMPL header is intended to define CMPL options, solver options and display options for the specific CMPL model. An additional intention of the CMPL header is to specify external data files which are to be connected to the CMPL model. The elements of the CMPL header are not part of the CMPL model and are processed before the CMPL model is interpreted.

The elements of CMPL header correspond to the command line options that can be used in the call to CMPL. Exceptions are only those command line options that must already be evaluated before the CMPL file is read and therefore cannot be used in CMPL header.

Each line for CMPL header starts with % as the first non-whitespace character. This is followed by the name of the command line option (without the -, which introduces a command line option in the command line). This is followed by the arguments of the command line option, separated by whitespace.

Alternatively, the line can begin with %arg. In this case, command line options and their arguments can be specified as on the command line itself (i.e. with - in front of the name of the command line option). Several command line options can then also be on one line.

There are the following minor differences in syntax between specifying options directly on the command line or in the CMPL header:

- CMPL comments can also be used in CMPL header lines as desired.
- In the CMPL header, only double quotes can be used to enclose arguments that contain whitespaces. Double quotes contained therein must be escaped with `\`. On the command line, however, the operating system-specific rules apply.
- Except for `%arg`, a value is also considered an argument if it begins with `-`. With `%arg`, on the other hand, a new command line option is started with it, so that several command line options can be in one CMPL header line. If the value is enclosed in double quotes, it is also considered an argument with `%arg`.
-

Important uses of CMPL headers include specifying options for the solver and for the result display:

%solver *solverName*	Specifies the solver
%opt *solverName solverOpt* [*solverOptVal*]	Specifies an option for a solver
%display var|con=*name*[*****][,*name1*[*****]]	Sets variable name(s) or constraint name(s) that are to be displayed in one of the solution reports. Different names are to be separated by spaces. If `name` is combined with the asterix `*` then all variables or constraints with names that start with `name` are selected.
%display nonZeros	Only variables and constraints with nonzero activities are shown in the solution report.
%display ignoreCons	Ignores constraints in the solution report. Only variables are shown in the solution report.
%display ignoreVars	Ignores variables in the solution report. Only constraints are shown in the solution report.
%display solutionPool	Gurobi and Cplex are able to generate and store multiple solutions to a mixed integer programming (MIP) problem. With the display option `solutionPool` feasible integer solutions found during a MIP optimisation can be shown in the solution report. It is recommended to control the behaviour of the solution pool by setting the particular Gurobi or Cplex solver options.

Examples:

`%solver glpk`	GLPK is used as the solver.
`%solutionAscii`	CMPL writes the optimisation results in an ASCII file.
`%arg -solver cbc ↵` `  -url http://194.95.44.187:8008`	CBC is to be executed on a CMPLServer located at 194.95.44.187.
`%opt cbc ratio=0.1`	If CBS is the invokes solver then a MipGap of 10% is used.
`%opt glpk nopresol`	If GLPK is used then the pre-solver is switched off.
`%display var=x`	Only the variable `x` is to be displayed in the solution report.
`%display con=x*,y*`	All constraints with names that start with `x` or `y` are shown in the solution report.

2.2.2 Include

The command line option `include` can be used to specify a CMPL file to be included. It is particularly useful to use this as a CMPL header, as the specified CMPL file is inserted at this point.

%include *fileName*	Includes the specified CMPL file (relative to the directory in which the current CMPL file is located).

If the file name contains spaces, then it must be enclosed in double quotes.

If the file name contains a directory specification, then `/` has to be used as separator (independently of the operation system).

The following CMPL file `parameters.cmpl` is used for the definition of a couple of parameters:

`c := ( 1, 2, 3 );` `b := ( 15, 20 );` `A := ( ( 5.6, 7.7, 10.5 ),` `( 9.8, 4.2, 11.1 ) );`	`parameters.cmpl`
`par:` `%include parameters.cmpl` `var:` `x[defset(c)]: real[0..];` `obj:` `c^T * x -> max;` `con:` `A * x <= b;`	Using `include` CMPL generates the following model: $1 \cdot x_1 + 2 \cdot x_2 + 3 \cdot x_3 \rightarrow max!$ $s.t.$ $5.6 \cdot x_1 + 7.7 \cdot x_2 + 10.5 \cdot x_3 \leq 15$ $9.8 \cdot x_1 + 4.2 \cdot x_2 + 11.1 \cdot x_3 \leq 20$ $x_j \geq 0; j = 1(1)3$

2.2.3 CmplData

2.2.3.1 CmplData in CMPL Header

CmplData is used to separate model and data in CMPL. The command line option `data` is used for this purpose. The arguments of this command line option define parameters and sets for CMPL, whose concrete values are read from a CmplData file. It makes sense to use this command line option in the CMPL header, but of course it can also be used on the command line itself.

Usage CMPL header for defining external data:

```
%data [filename] : [set1 set[[rank]]] [, set2 set[[rank]] , … ]
%data [filename] : [param1] [, param2 , … ]
%data [filename] : [paramarray1[set]] [, paramarray2[set] , … ]
```

filename	File name of the CmplData file If the file name contains white spaces the name must be enclosed in double quotes. If *filename* is not specified the generic name *modelname*.cdat will be used, where *modelname*.cmpl is the name of the cmpl file.
[set1 **set***[[rank]]][,set2* **set***[[rank]],* *…]*	Specifies a set with the name *set1* and the rank *rank*

	Specification of the rank is optional. If specified, then it must match the rank within the CmplData file. For more than one set the sets are to be separated by commas.
[param1] [, param2 , …]	Specifies a scalar parameter If more than one parameters are to be specified then the parameters are to be separated by commas.
[paramarray1`[`*set*`]][,paramarray2`[`*set*`],…]`	Specifies a parameter array and the set over which the array is defined For more than one parameter array the entries are to be separated by commas.

The easiest form to specify an external data is `%data`. In this case a generic filename *`modelname`*`.cdat` will be used and all sets and parameters that are defined in this file will be read.

If parameters and sets are specified in `%data`, then all definitions of the parameters and sets can be mixed with another. But a set must be specified before it is used in a definition of a parameter array.

Any number of CMPL header lines can be specified, both for the same data file and for any number of other data files.

Examples:

`%data myProblem.cdat: n set, a[n]`	Reads the set `n` and the vector `a` which is defined over the set `n` from the file `myProblem.cdat`
`%data myProblem.cdat`	Reads all parameters and sets that are defined in the file `myProblem.cdat`
`%data : n set[1], a[n]`	Reads (assuming a CMPL model name `myproblem2.cmpl)` the 1-tuple set `n` and the vector `a` which is defined over `n` from `myProblem2.cdat.`
`%data`	Assuming a CMPL model name `myproblem2.cmpl` all sets and parameters are to be read from `myProblem2.cdat.`
`%data : routes set[2],costs[routes]`	Assuming a CMPL model name `myproblem.cmpl` the 2-tuple set `routes` and the matrix `costs` defined over `routes` are to be read from `myProblem.cdat.`

If `data` is used as a command line option directly on the command line, the corresponding definitions including the file name can be specified altogether as a single argument string in double quotes. Even then, this command line option can be used as often as desired.

2.2.3.2 CmplData file format

A CmplData file is a plain text file that contains the definition of parameters and sets with their values in a specific syntax. The parameters and sets can be read into a CMPL model by using the command line option `data`, for instance by using it in CMPL header.

Usage:

```
%name < numberOrString >                  # scalar parameter

%name set[[rank]] < setExpression >       # set definition

%name [set] [= default] [indices] < listOfNumbersOrStrings >
                                          # parameter array
#text                                     # comments
```

Excluding comments each CmplData definition starts with `%`.

`%name < numberOrString >`	A scalar parameter `name` is assigned a single string or number.
`%name set[[rank]] < setExpression >`	Definition of an *n*-tuple set A set definition starts with the *`name`* followed by the keyword `set`. For *n*-tuple sets with *n*>1 the rank of the set is to be specified enclosed by square brackets. For enumeration sets the entries of the sets are separated by white spaces and imbedded in angle brackets. It is also possible to define algorithmic sets in normal CMPL syntax.
`%name [set] [= default] [indices] < listOfNumbersOrStrings >`	Definition of a parameter array The specification of a parameter array starts with the `name` followed by one or more sets, over which the array is defined. If more than one set is used then the sets have to be separated by commas. The set or sets have to be defined before the parameter definition. If the data entries are specified by their indices (keyword *`indices`*) then a default value can be defined. The data entries can be strings or numbers and have to be separated by white spaces and imbedded in angle brackets.

If the data entries are specified by their indices then each data entry has to start with the indices followed by the value and separated by white spaces. A thousand separator is not to be used.

For real numbers, the decimal separator is always a dot, regardless of the language used by the operating system.

If not so then the order of the elements are given by the natural order of the set or sets.

Examples:

`%a < 10 >`	Defines a scalar parameter `a` and assigns the number 10.
`%s set < 0..6 >`	`s` is assigned $s \in (0,1,\ldots,6)$
`%s set < 10(-2)4 >`	`s` is assigned $s \in (10,8,6,4)$
`%prod set < bike1 bike2 >` `%prod set < "bike 1"  "bike 2" >`	1-tuple enumeration set of strings
`%a set< 1 a  3  b 5 c >` `%x[a] < 10 20 30 40 50 60 >`	1-tuple enumeration set of strings and integers vector `x` identified by the set `a` is assigned an integer vector
`%data : a set, x[a]` `echo( x[1] );` `echo( x["a"] );` `{ ordered @i in a: echo( x[i] ); }`	reads the set `a` and the vector `x` into a CMPL model The following user messages are displayed: `10` `20` `10 20 30 40 50 60`
`%n set < 1..3 >` `%a[n,n] = 0 indices <  1 1 1` `2 2 1` `3 3 1 >` `%a[n,n] < 1 0 0` `0 1 0` `0 0 1 >`	Defines a 3x3 identity matrix Alternative formulation
`%x set < 1..2 >` `%y set < 1..2 >` `%z set < 1..2 >`	Definition of a data cube with the dimension `x,y,z`

`%cube[x,y,z] < 1 2 3 4 5 6 7 8 >`	*x y z value* 1 1 1 1 1 1 2 2 1 2 1 3 1 2 2 4 2 1 1 5 2 1 2 6 2 2 1 7 2 2 2 8
`%data : x set, y set, z set, cube[x,y,z]` `{ @i in x, @j in y, @k in z:` `  echo( i+","+j+","+k+":"+cube[i,j,k] );` `}`	Reads the sets `x,y,z` and the `cube` into a CMPL model The following user messages are displayed: `1,1,1:1` `1,1,2:2` `1,2,1:3` `1,2,2:4` `2,1,1:5` `2,1,2:6` `2,2,1:7` `2,2,2:8`
`%cube[x,y,z] = 0 indices <` `  1 1 1 1` `  2 2 2 1` `>`	Defines the following data cube: *x y z value* 1 1 1 1 1 1 2 0 1 2 1 0 1 2 2 0 2 1 1 0 2 1 2 0 2 2 1 0 2 2 2 1
`%x set[3] <` `  1 1 1` `  1 1 2` `  1 2 1` `  1 2 2` `  2 1 1` `  2 1 2` `  2 2 1` `  2 2 2` `>` `%cube[x] < 1 2 3 4 5 6 7 8 >`	A cube defined over a 3-tuple set: *x y z value* 1 1 1 1 1 1 2 2 1 2 1 3 1 2 2 4 2 1 1 5 2 1 2 6 2 2 1 7 2 2 2 8

`%data : x set[3], cube[x]` `{ @i in x:  echo( i +":"+cube[i] ); }`	Reads the 3-tuple set `x` and the parameter array `cube` The following user messages are displayed: `[1, 1, 1]:1` `[1, 1, 2]:2` `[1, 2, 1]:3` `[1, 2, 2]:4` `[2, 1, 1]:5` `[2, 1, 2]:6` `[2, 2, 1]:7` `[2, 2, 2]:8`
`%x set[3] < 1 1 1  1 1 2  1 2 1  1 2 2` `2 1 1  2 1 2  2 2 1  2 2 2 >` `%cube[x] = 0 indices <  1 1 1 1` `2 2 2 1 >`	Data `cube` defined over `x` *x y z value* 111 1 112 0 121 0 122 0 211 0 212 0 221 0 222 1
`%routes set[2] <  p1 c1` `p1 c2` `p1 c4` `p2 c2` `p2 c3` `p2 c4` `p3 c1` `p3 c3  >` `%c[routes]  < 3  2  6  5  2  3  2  4 >`	Defines a 2-tuple set `routes` and a matrix `c` that is defined over `routes`
`%data : routes set[2], c[routes]` `{ @i in routes:` `  echo( i + " : "+ c[i] );` `}`	Reads the 2-tuple set `routes` and the matrix `c` into a CMPL model The following user messages are displayed: `["p1", "c1"] : 3` `["p1", "c2"] : 2` `["p1", "c4"] : 6` `["p2", "c2"] : 5` `["p2", "c3"] : 2` `["p2", "c4"] : 3` `["p3", "c1"] : 2` `["p3", "c3"] : 4`

2.2.4 CmplXlsData

2.2.4.1 CmplXlsData in CMPL Header

CmplXlsData is CMPL's interface for reading sets and parameters from an Excel file and for writing optimisation results to an open Excel file. If the Excel file is not open, CMPL will open it automatically and the results of the optimisation can be seen immediately. Please note, this feature is only available on Windows and macOS if Microsoft Excel is installed on this system. CmplXlsData is implemented with Python3 using the Python for Excel (open-source) library by xlwings (www.xlwings.org).

The command line option `xlsdata` is used for this purpose. The arguments of this command line option define parameters and sets for CMPL, whose source Excel file and the corresponding cell ranges are specified in a CmplXlsData file. It makes sense to use this command line option in the CMPL header, but of course it can also be used on the command line itself.

Usage CMPL header for defining external data:

```
%xlsdata [filename] : [set1 set[[rank]]] [, set2 set[[rank]] , … ]
%xlsdata [filename] : [param1] [, param2 , … ]
%xlsdata [filename] : [paramarray1[set]] [, paramarray2[set] , … ]
```

filename	File name of the CmplXlsData file If the file name contains white spaces the name must be enclosed in double quotes. If *filename* is not specified the generic name *modelname*.xdat will be used, where *modelname*.cmpl is the name of the cmpl file.
[set1 **set[[***rank***]]][,***set2* **set[[***rank***]],** *…]*	Specifies a set with the name *set1* and the rank *rank* Specification of the rank is optional. If specified, then it must match the rank within the CmplXlsData file. For more than one set the sets are to be separated by commas.
[param1] [, param2 , …]	Specifies a scalar parameter If more than one parameters are to be specified then the parameters are to be separated by commas.
*[paramarray1***[***set***]***][,paramarray2***[***set***]***,…]*	Specifies a parameter array and the set over which the array is defined For more than one parameter array the entries are to be separated by commas.

The easiest form to specify an external data is `%xlsdata`. In this case a generic filename *model-name*`.xdat` will be used and all sets and parameters that are defined in this file will be read.

If parameters and sets are specified in `%data`, then all definitions of the parameters and sets can be mixed with another. But a set must be specified before it is used in a definition of a parameter array.

Any number of CMPL header lines can be specified, both for the same data file and for any number of other data files.

2.2.4.2 CmplXlsData file format

A CmplXlsData file is a plain text file that contains the definition of parameters and sets with the cell addresses of their values in the specified Excel file in a specific syntax. Additionally, the optimisation results to be written to Excel with their cell addresses can be specified in this file. The parameters and sets can be read into a CMPL model by using the command line option `xlsdata`, for instance by using it in CMPL header.

A CmplXlsData file contains usually the three sections `@source`, `@input` and `@output`. The `@meta` section is intended to specify the Excel file and optionally the sheet to be used to read sets and parameters and to write the optimisation results. If the Excel file is not already open, it will be opened automatically when the CmplXlsData file is accessed. In the `@input` section, the sets and parameters to be read into the Cmpl model have to be specified with their cell ranges. In contrast to a CmplData file, such a specification can only be specified in one line. The `@output` section specifies the optimisation result elements to written to the Excel file. This results are displayed directly in the Excel file.

Usage:

```
#text                                       # comments

@source                                     # section for specifying the Excel
                                              file and the default sheet

%file < excelFileName >                     # name of the Excel file
[%sheet < activeSheetName > ]               # optional - name of the active
                                              sheet

@input                                      # section for specifying sets and
                                              parameters to be read into Cmpl

%name < cellReference >                     # scalar parameter

%name set[[rank]] < cellRangeReference >    # set definition

                                            # parameter array
%name [set[,set1, ...]] < cell_range_reference >
```

```
@output                                    # section for specifying the
                                             optimisation results to be written
                                             to Excel

                                           # singleton variable or constraint
%name.activity  < cell_reference >
%name.type < cell_reference >
%name.lowerBound < cell_reference >
%name.upperBound < cell_reference >
%name.marginal < cell_reference >
                                         # array of variables or constraints

%name[set[,set1, ...]].activity < cell_range_reference >
%name[set[,set1, ...]].type < cell_range_reference >
%name[set[,set1, ...]].lowerBound < cell_range_reference >
%name[set[,set1, ...]].upperBound < cell_range_reference >
%name[set[,set1, ...]].marginal < cell_range_reference >

                                        # general solution information
%objName < cell_reference >
%objSense < cell_reference >
%objValue < cell_reference >
%objStatus < cell_reference >
%nrOfVars < cell_reference >
%nrOfCons < cell_reference >
%solverName < cell_reference >
%solverMsg < cell_reference >
```

@source	Section for specifying the Excel file and the default sheet
%file < excelFileName **>**	Name of the Excel file The name can contain spaces, but it is not allowed to enclose the file name with double quotes.
*[***%sheet <** activeSheetName **>** *]*	Optional argument to specify the name of the active sheet In each entry for the inputs and the outputs, the sheet can be specified directly.
@input	Section for specifying sets and parameters to be read into Cmpl
%name **<** *cellReference* **>**	A scalar parameter `name` is assigned a single string or number available in Excel at the specified cell.

%name **set***[***[***rank***]***]* **<** *cellRangeReference* **>**

Definition of an *n*-tuple set

A set definition starts with the name followed by the keyword `set`. For *n*-tuple sets with *n*>1 the rank of the set is to be specified enclosed by square brackets.

The set is assigned the entries available in Excel in the cells specified in the cell range reference.

Please note that whitespaces are not allowed as part of an index.

%name **[***set[,set1, ...]***]**
<*cellRangeReference* **>**

Definition of a parameter array

The specification of a parameter array starts with the name followed by one or more sets, over which the array is defined. If more than one set is used then the sets have to be separated by commas.

The set or sets have to be defined before the parameter definition.

The data entries can be strings or numbers and have to be found at the specified cell range reference in Excel.

@output

Section for specifying the optimisation results to be written to Excel

%name.**activity<** *cell_reference* **>**
%name.**type <** *cell_reference* **>**
%name.**lowerBound <** *cell_reference* **>**
%name.**upperBound <** *cell_reference* **>**
%name.**marginal <** *cell_reference* **>**

Singleton variable or constraint

For a singleton variable or constraint named `name`, the activity, type, limits and dual values can be written to Excel in the cell specified by *cell_reference*.

The name is followed by a dot and one of the keyword (`activity`, `type`, `lowerbound`, `upperbound`, `marginal`) for the information to be written to Excel.

*%name***[***set[,set1, ...]***]**.**activity <**
cell_range_reference **>**
*%name***[***set[,set1, ...]***]**.**type <**
cell_range_reference **>**
*%name***[***set[,set1, ...]***]**.**lowerBound <**
cell_range_reference **>**
*%name***[***set[,set1, ...]***]**.**upperBound <**
cell_range_reference **>**
*%name***[***set[,set1, ...]***]**.**marginal <**
cell_range_reference **>**

Arrays of variables or constraints

A complete array of variables or constraints named `name`, the activity, type, limits and dual values can be written to Excel in the specified cell range.

The specification of an array of variables or constraints starts with the name followed by one or more sets, over which the array is defined. If more than one set is used then the sets have to be separated by commas. This is followed by a dot and one of the keywords for the attributes `activity`, `type`, `lowerbound`, `upperbound`, `marginal` of the result information to be written to Excel.

A single element of the array cannot be accessed.

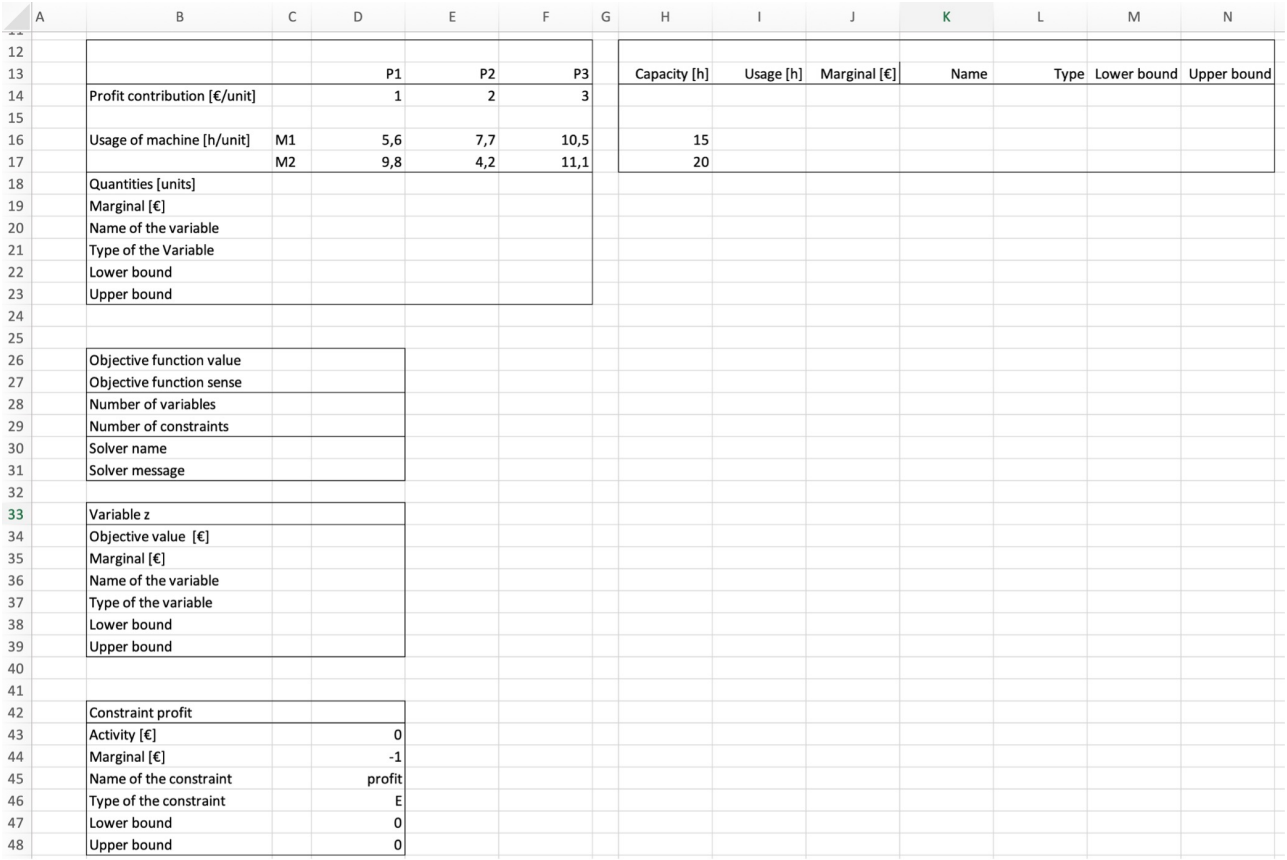
%objName < *cell_reference* **>**	Writes the name of the objective function to Excel in the specified cell
%objSense < *cell_reference* **>**	Writes the objective sense to Excel in the specified cell
%objValue < *cell_reference* **>**	Writes the objective function value to Excel in the specified cell
%objStatus < *cell_reference* **>**	Writes the status of the objective function to Excel in the specified cell
%nrOfVars < *cell_reference* **>**	Writes the number of the variables to Excel in the specified cell
%nrOfCons < *cell_reference* **>**	Writes the number of the constraints to Excel in the specified cell
%solverName < *cell_reference* **>**	Writes the name of the solver invoked to Excel in the specified cell
%solverMsg < *cell_reference* **>**	Writes a message of the solver invoked to Excel in the specified cell

The following example illustrates the functionality of CmplXlsData for the simple production mix problem below:

$$1 \cdot x_1 + 2 \cdot x_2 + 3 \cdot x_3 \rightarrow max!$$
$$s.t.$$
$$5.6 \cdot x_1 + 7.7 \cdot x_2 + 10.5 \cdot x_3 \leq 15$$
$$9.8 \cdot x_1 + 4.2 \cdot x_2 + 11.1 \cdot x_3 \leq 20$$
$$x_j \geq 0;\ j = 1(1)3$$

This model seeks the quantities of three products depending on the capacities of two machines in order to maximise the total profit contribution margin. The coefficients in the objective function are the unit contribution margins. There are two machines whose capacities of 15 hours for the first machine and 20 hours for the second machine cannot be exceeded. The utilisation of the machines as a function of the production quantities is given in the left-hand sides of the two constraints. The coefficients are the utilisation of a machine per unit of product.

The sets and parameters are organised in an Excel file named `ExampleData.xlsx` in the `ProdProg` sheet as shown in the following screenshot. Afterwards, the results of the optimisation are also written to this sheet. In contrast to CMPLData, the notation of the operating system language can be used for real numbers that are to be read into the Cmpl model. For example, the Excel example uses a German notation for the data of the `A` matrix in the cell range `D16:F17`.



			P1	P2	P3	Capacity [h]	Usage [h]	Marginal [€]	Name	Type	Lower bound	Upper bound
Profit contribution [€/unit]			1	2	3							
Usage of machine [h/unit]		M1	5,6	7,7	10,5	15						
		M2	9,8	4,2	11,1	20						
Quantities [units]												
Marginal [€]												
Name of the variable												
Type of the Variable												
Lower bound												
Upper bound												

Objective function value	
Objective function sense	
Number of variables	
Number of constraints	
Solver name	
Solver message	

Variable z	
Objective value [€]	
Marginal [€]	
Name of the variable	
Type of the variable	
Lower bound	
Upper bound	

Constraint profit	
Activity [€]	0
Marginal [€]	-1
Name of the constraint	profit
Type of the constraint	E
Lower bound	0
Upper bound	0

The corresponding Cmpl file `prodProg.cmpl` starts in the first line with the header entry for processing the CmplXlsData file `prodProg.xdat`. This line specifies the sets and the parameter arrays to be read into the Cmpl model, which are then used in the following sections of the model.

```
%xlsdata prodProg.xdat : P set, M set, c[P], b[M], A[M,P]

var:
   x[P]: real;
   z: real;

obj:
   z -> max;

con:
   profit: c^T * x = z;
   machine: A * x <= b;
```

In addition to the array of variables `x` for the production quantities, a real variable `z` is defined to store the objective function value. Therefore, the value of the variable `z` is to be maximised. The original objective function is defined as a constraint `profit` and its value must be equal to the variable `z`. The constraints for the two machines are defined in the last line.

The CmplXlsData file `prodProg.xdat` starts with a `source` section with the entries for the Excel file `ExampleData.xlsx` and the sheet `ProdProg`.

```
@source
%file < ExampleData.xlsx >
%sheet < ProdProg >
```

The following input section usually starts with the definition of index sets that will later be used for parameter arrays. The line `%P set < D13:F13 >` defines a set `P` and assigns the data given in the Excel sheet in the cell range `D13:F13`. This set is the set of the products and is assigned the values `"P1","P2"` and `"P3"`. The following line defines the set of the machines named `M`, which is assigned the values `"M1"` and `"M2"` (from of the cell range `C16:C17)`.

```
@input
%P set < D13:F13 >
%M set < C16:C17 >

%c[P] < D14:F14 >
%A[M,P] < D16:F17 >
%b[M] < G16:G17 >
```

The set `P` is used for the definition of the vector of the objective function coefficients `c`. This vector is assigned the values in the cell range `D14:F14`. For the usage of the capacities of the machines per unit of the products, a matrix `A` is defined in the next row. For this purpose, the set `M` is used for the rows and the set `P` for the columns. The data given in the cell range `D16:F17` are assigned to the matrix. The last parameter array to be defined is the vector `b` of the capacities of the machines using the set `M`. The both capacities are given in `G16:G17`.

The output section is intended to write all requested results of the optimisation into the specified Excel sheet. This can be general information about the solution, the model and the solver invoked as shown in the following listing:

```
@output
%objValue < D26 >
%objSense < D27 >
%nrOfVars < D28 >
%nrOfCons < D29 >
%solver < D30 >
%solverMsg < D31 >
```

The objective function value of `4.28571` is to be written in the cell `D26`. Please note, the format of the results written into the Excel sheet follows notation of the operating system language (here German). The information about the objective sense, the number of the variables and constraints are written into the cells `D27`, `D28` and `D29`. The name of the solver and a general message of this solver shall be shown after the optimisation in the cells `D30` and `D31`.

	B	C	D
25			
26	Objective function value		4,28571
27	Objective function sense		max
28	Number of variables		4
29	Number of constraints		3
30	Solver name		CBC
31	Solver message		normal

All result information of an array of variables can be written using the attributes `activity`, `type`, `lowerbound`, `upperbound`, `marginal` into the specified cell ranges. All numeric values are written as real numbers. If a value is equal to infinity (or negative infinity) the string `"inf"` (`"-inf"`) is written. The marginals of the variables can be either reduced costs (activity equal to zero, negative marginal value) or shadow prices if the activity of the variable is equal to its upper bound. The type of a variable can be `"C"` for a real variable, `"I"` for integer and `"B"` vor binary.

```
%x[P].activity < D18:F18 >
%x[P].marginal < D19:F19 >
%x[P].name < D20:F20 >
%x[P].type < D21:F21 >
%x[P].lowerBound < D22:F22 >
%x[P].upperBound < D23:F23 >
```

	B	C	D	E	F
11					
12					
13			P1	P2	P3
14	Profit contribution [€/unit]		1	2	3
15					
16	Usage of machine [h/unit]	M1	5,6	7,7	10,5
17		M2	9,8	4,2	11,1
18	Quantities [units]		0	0	1,42857
19	Marginal [€]		-0,6	-0,2	0
20	Name of the variable		x[P1]	x[P2]	x[P3]
21	Type of the Variable		C	C	C
22	Lower bound		0	0	0
23	Upper bound		inf	inf	inf

If a variable is a singleton variable the same attributes can used to write the result information. The following listing shows the access to the results information of the variable `z` as substitute of the objective function.

```
%z.activity < D34 >
%z.marginal < D35 >
%z.name < D36 >
%z.type < D37 >
%z.lowerBound < D38 >
%z.upperBound < D39 >
```

	B	C	D
33	Variable z		
34	Objective value [€]		4,28571
35	Marginal [€]		0
36	Name of the variable		z
37	Type of the variable		C
38	Lower bound		0
39	Upper bound		inf

The result information of a singleton constraint or an array of constraints can be accessed in the same way as the variables with the attributes `activity`, `type`, `lowerbound`, `upperbound`, `marginal` as shown below. All numeric values are written as real numbers. If a value is equal to infinity (or negative infinity) the string `"inf"` (`"-inf"`) is written. The marginals of the constraints are shadow prices if a constraint is a bottleneck. The type of a constraint can be `"L"` if the left-hand side of the constraint is `less than or equal` to the right-hand side, `"G"` for `greater than or equal` and `"E"` for `Equality`.

```
%machine[M].activity < I16:I17 >
%machine[M].marginal < J16:J17 >
%machine[M].name < K16:K17 >
%machine[M].type < L16:L17 >
%machine[M].lowerBound < M16:M17 >
%machine[M].upperBound < N16:N17 >

%profit.activity < D43 >
%profit.marginal < D44 >
%profit.name < D45 >
%profit.type < D46 >
%profit.lowerBound < D47 >
%profit.upperBound < D48 >
```

	G	H	I	J	K	L	M	N
12								
13		Capacity [h]	Usage [h]	Marginal [€]	Name	Type	Lower bound	Upper bound
14								
15								
16		15	15	0,285714	machine[M1]	L	-inf	15
17		20	15,8571	0	machine[M2]	L	-inf	20

	B	C	D
41			
42	Constraint profit		
43	Activity [€]		0
44	Marginal [€]		-1
45	Name of the constraint		profit
46	Type of the constraint		E
47	Lower bound		0
48	Upper bound		0

2.3 Incompatibilities with Cmpl 1.12

Since CMPL has been fundamentally redeveloped, there is no complete backwards compatibility. Known incompatibilities with the previous version are:

- **`echo` and `error`**
  These are now functions whose argument must be enclosed in parentheses. In the previous version, no brackets were required here.

- **Use of `:=` within a code block header**
  A code block header `{ i := s: ... }` with a set `s` now leads to an assignment of the set to the code block symbol `i`. In the previous version, it led to iteration over the set instead. Now an iteration can only be expressed by `{ i in s: ... }`.

- **Write protection for code block symbols**
  In contrast to the previous version, code block symbols can no longer be assigned in the code block body, but only receive their value initially in the code block header. A symbol used as a separate counter, as is typical for loops formulated with repeat, can therefore no longer be a code block symbol.

- **Referencing a code block in `break`, `continue` or `repeat`**
  Only the first code block symbol defined in a code block can be used as a reference to it. In the previous version, referencing was also possible via other code block symbols defined in it and via a name preceding the code block.

- **Code block with more than one header**
  Even if a code block has more than one comma-separated headers, alternatives in the code block as well as `break`, `continue` and `repeat` refer to the entire code block itself. In the previous version, it referred instead only to the last of the specified headers

- **Execution order in iterations without the keyword `ordered`**
  The execution of an iteration in a code block is done in the default order of the set being iterated over when only one thread is used. If more than one thread is used, there is no fixed execution order at all, but it is executed in parallel as far as the maximum number of threads allows. In the previous version, iteration was done in user order of the set instead. For this behaviour, the use of the keyword `ordered` is now necessary.

- **Use of `default` for alternatives in a code block header**

  A keyword `default`, as used in the previous version for an unconditional alternative in a code block, no longer exists. For such an unconditional alternative, simply use an empty header instead. Nevertheless, the old use of `default` continues to work. This is because it now corresponds to the definition of a new referencing code block symbol with the name `default`.

- **Assignment with =**

  The assignment with `=`, which was still possible in the previous version, has been dropped (it was already obsolete in the previous version). The operator `=` now only serves as a comparison operator. For an assignment, `:=` must be used instead.

- **Operators << and `element`**

  The operators `<<` and `element` implemented in the previous version have been dropped. Instead, the operator `in` must be used.

- **Operators `div` and `mod`**

  The operators `div` and `mod` implemented in the previous version have been dropped. Instead, the functions `div` and `mod` must be used.

- **Semantic of `[]`**

  In the previous version, `[]` stood for the infinite set of all 1-tuples. The infinite set of all 2-tuples was then expressed as `[,]`, etc. Now `[]` stands for the infinite set of all tuples with any rank. A tuple construction of the kind `[,]` is arbitrarily possible, and also gives the infinite set of all tuples with any rank. The infinite set of all 1-tuples can be represented with `[*]`.

- **String literals**

  A literal string must be enclosed in double quotes. In the previous version, it was alternatively possible to use single quotes.

- **Assignment with tuple disaggregation**

  In the previous version, it was possible to use an assignment of the type `[a,b] := t;` when `t` is a 2-tuple. Such an assignment is no longer possible because a tuple construction expression is not allowed on the left-hand side of an assignment. As a substitute, tuple matching can be used in a code block header, for example `{ [@a, @b] = t: ... }`.

- **Associativity for set expressions of the form `1(1)n`**

  In the previous version, the associativity of such an expression was lower than that of numerical operations, so that, for example, `1(1)n+1` was semantically equivalent to `1(1)(n+1)`. Now the associativity of such an expression is higher than that of numerical operations, and `1(1)n+1` is considered as `(1(1)n)+1`, which leads to an error message due to incompatible operands in the execution of the addition. For the former semantics, the example must now be written `1(1)(n+1)`.

- **Line names with `$` substitutions**

  The possibility in the previous version to use `$...$` substitutions in row names has been completely dropped. The names of matrix lines are now assigned via regular assignments to CMPL symbols with the assignment operator `:`. Index specifications can be used as usual in CMPL.

- **Changed semantics for function `defset`**

  This function returns the tupleset of all indices of an array. In contrast, the previous version returned a set of 1-tuples only from the first part of the indices. So the semantics is only unchanged in the case when the argument array contains only 1-tuples as indices.

- **Non implemented functions**

  The functions `readstdin` and `readcsv`, which existed in the previous version, are currently not implemented. It has not been decided whether and with which semantics they will be implemented again.

- **Not implemented operations for sets**

  The operations `*` and `-` are currently not implemented for sets. Implementation is planned.

- **Keywords `.integer.` and `.string.` are dropped**

  In the previous version, these keywords referred to infinite sets consisting of all integer values and all strings. They are dropped without replacement.

- **Function `type` is dropped**

  In the previous version, the data type of an argument could be analysed with this function. Now `type` designates the data type of a data type instead. For getting the data type of an expression or a decision variable `x` now `x.$type` can be used.

- **Type definition for parameters**

  In the previous version, the data type for a parameter symbol could be specified by writing it with a colon after the symbol in the definition. This syntax has been dropped. Data type constraints for parameter symbols must now be specified via attributes instead.

- **Syntax for `%include`**

  In the previous version, `include` was not part of CMPL header, but a special command in its own right. It was therefore written without `%`. Since the include functionality is now provided by CMPL header, it must be written `%include`.

- **Syntax for `%data`**

  If multiple symbols are specified, they must be separated by commas. If no filename is specified, there must be at least one whitespace directly after the colon after `%data`. In the previous version, there were other syntactical variants that are no longer supported.

- **Syntax for CmplServer**

  In the previous version, the url of a CmplServer was defined by the command line option `-cmplUrl`. Now the option `-url` is used.

2.4 Examples

2.4.1 Selected decision problems

2.4.1.1 The diet problem

The goal of the diet problem is to find the cheapest combination of foods that will satisfy all the daily nutritional requirements of a person for a week.

The following data is given (example cf. Fourer/Gay/Kernigham 2003, p. 27ff.) :

food	cost per package	provision of daily vitamin requirements in percentages			
		A	B1	B2	C
BEEF	3.19	60	20	10	15
CHK	2.59	8	2	20	520
FISH	2.29	8	10	15	10
HAM	2.89	40	40	35	10
MCH	1.89	15	35	15	15
MTL	1.99	70	30	15	15
SPG	1.99	25	50	25	15
TUR	2.49	60	20	15	10

The decision is to be made for one week. Therefore the combination of foods has to provide at least 700% of daily vitamin requirements. To promote variety, the weekly food plan must contain between 2 and 10 packages of each food.

The mathematical model can be formulated as follows:

$$3.19 \cdot x_{BEEF} + 2.59 \cdot x_{CHK} + 2.29 \cdot x_{FISH} + 2.89 \cdot x_{HAM} + 1.89 \cdot x_{MCH} + 1.99 \cdot x_{MTL} + 1.99\, x_{SPG} + 2.49 \cdot x_{TUR} \rightarrow min!$$

$$s.t.$$

$$60 \cdot x_{BEEF} + 8 \cdot x_{CHK} + 8 \cdot x_{FISH} + 40 \cdot x_{HAM} + 15 \cdot x_{MCH} + 70 \cdot x_{MTL} + 25\, x_{SPG} + 60 \cdot x_{TUR} \leq 700$$

$$20 \cdot x_{BEEF} + 0 \cdot x_{CHK} + 10 \cdot x_{FISH} + 40 \cdot x_{HAM} + 35 \cdot x_{MCH} + 30 \cdot x_{MTL} + 50\, x_{SPG} + 20 \cdot x_{TUR} \leq 700$$

$$10 \cdot x_{BEEF} + 20 \cdot x_{CHK} + 15 \cdot x_{FISH} + 35 \cdot x_{HAM} + 15 \cdot x_{MCH} + 15 \cdot x_{MTL} + 25\, x_{SPG} + 15 \cdot x_{TUR} \leq 700$$

$$15 \cdot x_{BEEF} + 20 \cdot x_{CHK} + 10 \cdot x_{FISH} + 10 \cdot x_{HAM} + 15 \cdot x_{MCH} + 15 \cdot x_{MTL} + 15\, x_{SPG} + 10 \cdot x_{TUR} \leq 700$$

$$x_j \in \{2, 3, \ldots, 10\} \quad ; j \in \{BEEF, CHK, DISH, HAM, MCH, MTL, SPG, TUR\}$$

The CMPL model `diet.cmpl` can be formulated as follows:

```
par:
      NUTR := set("A","B1", "B2", "C");
      FOOD := set("BEEF", "CHK", "FISH", "HAM", "MCH", "MTL", "SPG", "TUR");

      #cost per package
      costs[FOOD] := ( 3.19, 2.59, 2.29, 2.89, 1.89, 1.99, 1.99, 2.49 );
```

```
      #provision of the daily requirements for vitamins in percentages
      vitamin[NUTR, FOOD] := ( (60, 8, 8, 40, 15, 70, 25, 60) ,
                               (20, 0, 10, 40, 35, 30, 50, 20) ,
                               (10, 20, 15, 35, 15, 15, 25, 15),
                               (15, 20, 10, 10, 15, 15, 15, 10)
                              );

      #weekly vitamin requirements
      vitMin[NUTR]:= (700,700,700,700);
var:
      x[FOOD]: integer[2..10];

obj:
      cost: costs^T * x -> min;

con:
      # minimum vitamin restriction
      vit: vitamin * x >= vitMin;
```

An alternative formulation is based on the cmplData file `diet-data.cdat` that is formulated as follows:

```
%NUTR set < A B1  B2  C >
%FOOD set < BEEF CHK FISH HAM MCH MTL SPG TUR   >

#cost per package
%costs[FOOD] < 3.19 2.59 2.29 2.89 1.89 1.99 1.99 2.49 >

#provision of the daily requirements for vitamins in percentages
%vitamin[NUTR,FOOD] <   60  8  8  40  15  70  25  60
                        20  0  10  40  35  30  50  20
                        10  20  15  35  15  15  25  15
                        15  20  10  10  15  15  15  10 >

#weekly vitamin requirements
%vitMin[NUTR]  < 700 700 700 700 >
```

Assuming that the corresponding CMPL file `diet-data.cmpl` is in the same working directory the model can be formulated as follows:

```
%data diet-data.cdat: FOOD set, NUTR set, costs[FOOD], vitamin[NUTR,FOOD], vit-
Min[NUTR]

var:
      x[FOOD]: integer[2..10];

obj:
      cost: costs^T * x -> min;

con:
      # minimum vitamin restriction
      vit: vitamin * x >= vitMin;
```

Solving this CMPL model through using the command:

```
cmpl diet-data.cmpl
```

leads to the same solution as for the first formulation:

```
------------------------------------------------------------------------------------------------------------
Problem             diet-data.cmpl
Nr. of variables    8
Nr. of constraints  4
Objective name      cost
Solver name         CBC
Display variables   (all)
Display constraints (all)
------------------------------------------------------------------------------------------------------------

Objective status    optimal
Objective value     101.14 (min!)

Variables
Name                Type           Activity        Lower bound        Upper bound           Marginal
------------------------------------------------------------------------------------------------------------
x[BEEF]                I                  2                  2                 10                  -
x[CHK]                 I                  8                  2                 10                  -
x[FISH]                I                  2                  2                 10                  -
x[HAM]                 I                  2                  2                 10                  -
x[MCH]                 I                 10                  2                 10                  -
x[MTL]                 I                 10                  2                 10                  -
x[SPG]                 I                 10                  2                 10                  -
x[TUR]                 I                  2                  2                 10                  -
------------------------------------------------------------------------------------------------------------

Constraints
Name                Type           Activity        Lower bound        Upper bound           Marginal
------------------------------------------------------------------------------------------------------------
vit[A]                 G               1500                700                inf                  -
vit[B1]                G               1330                700                inf                  -
vit[B2]                G                860                700                inf                  -
vit[C]                 G                700                700                inf                  -
------------------------------------------------------------------------------------------------------------
```

2.4.1.2 Production mix

This model calculates the production mix that maximizes profit subject to available resources. It will identify the mix (number) of each product to produce and any remaining resource.

The example involves three products which are to be produced with two machines. The following data is given:

		P1	P2	P3	upper bounds [h]
upper bound of a product	[units]	250	240	250	
selling price per unit	[€/unit]	500	600	450	
direct costs per unit	[€/unit]	425	520	400	
profit contribution per unit	[€/unit]	75	80	50	
machine hours required per unit					
machine 1	[h/unit]	8	15	12	1,000
machine 2	[h/unit]	15	10	8	1,000

The mathematical model can be formulated as follows:

$$75 \cdot x_1 + 80 \cdot x_2 + 50 \cdot x_3 \rightarrow max\ !$$
$$s.t.$$
$$8 \cdot x_1 + 15 \cdot x_2 + 12 \cdot x_3 \leq 1{,}000$$
$$15 \cdot x_1 + 10 \cdot x_2 + 8 \cdot x_3 \leq 1{,}000$$

$$x_1 \in \{0,1,\ldots,250\}$$
$$x_2 \in \{0,1,\ldots,240\}$$
$$x_3 \in \{0,1,\ldots,250\}$$

The CMPL model `production-mix.cmpl` is formulated as follows:

```
par:
   #vectors for the prices and costs per unit of the three products
   price := (500, 600, 450 );
   costs := (425, 520, 400);

   #upper bound of the products
   xMax := (250, 240, 250 );

   #calculation the vector of the profit contribution per unit
   c := price - costs;

   #machine hours required per unit
   a := ((8, 15, 12), (15, 10, 8));

   #upper bounds of the machines
   b := (1000, 1000);

var:
   x[defset(price)]: integer;

obj:
   profit: c^T  * x ->max;

con:
   res: a * x <= b;
   x <=xMax;
```

The model can be formulated alternatively by using the cmplData `production-mix-data.cdat` file.

```
%products set < 1..3 >
%machines set < 1..2 >

%price[products] <500 600 450 >
%costs[products] <425 520 400 >

#machine hours required per unit
%a[machines,products] < 8  15  12  15 10 8 >

#upper bounds of the machines
%b[machines] < 1000 1000 >
```

```
#lower and upper bound of the products
%xMax[products] < 250  240  250>
%xMin[products] < 45 45 45 >

#fixed setup costs
%FC[products] < 500 400 500>
```

The parameter arrays `xMin` and `FC` are not necessary for the given problem and therefore not specified within the `%data` options in the following CMPL file `production-mix-data.cdat`:

```
%data : products set, machines set, price[products], costs[products], a[ma-
chines,products], b[machines], xMax[products]

par:
  c := price-costs;

var:
  x[defset(price)]: integer;

obj:
  profit: c^T  * x ->max;

con:
  res: a * x <= b;
  x <=xMax;
```

The CMPL command

```
cmpl production-mix-data.cmpl
```

leads to the following Solution:

```
------------------------------------------------------------------------------------------------------------
Problem              production-mix-data.cmpl
Nr. of variables     3
Nr. of constraints   2
Objective name       profit
Solver name          CBC
Display variables    (all)
Display constraints  (all)
------------------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      6395 (max!)

Variables
Name                 Type            Activity         Lower bound         Upper bound            Marginal
------------------------------------------------------------------------------------------------------------
x[1]                    I                  33                   0                 250                   -
x[2]                    I                  49                   0                 240                   -
x[3]                    I                   0                   0                 250                   -
------------------------------------------------------------------------------------------------------------

Constraints
Name                 Type            Activity         Lower bound         Upper bound            Marginal
------------------------------------------------------------------------------------------------------------
res[1]                  L                 999                -inf                1000                   -
res[2]                  L                 985                -inf                1000                   -
------------------------------------------------------------------------------------------------------------
```

2.4.1.3 Production mix including thresholds and step-fixed costs

This model seeks the production mix that maximises profit subject to available resources. When a product is produced, there are fixed set-up costs. There is also a threshold for each product. The quantity of a product is zero or greater than the threshold. This is the behaviour of a semi-continuous(integer) variable.

The example involves three products which are to be produced with two machines. The following data is given:

		P1	P2	P3	upper bounds [h]
production minimum of a product	[units]	45	45	45	
upper bound of a product	[units]	250	240	250	
selling price per unit	[€/unit]	500	600	450	
direct costs per unit	[€/unit]	425	520	400	
profit contribution per unit	[€/unit]	75	80	50	
set-up costs	[€]	500	400	500	
machine hours required per unit					
machine 1	[h/unit]	8	15	12	1,000
machine 2	[h/unit]	15	10	8	1,000

The mathematical model can be formulated as follows:

$$75 \cdot x_1 + 80 \cdot x_2 + 50 \cdot x_3 - 500 \cdot y_1 - 400 \cdot y_2 - 500 \cdot y_3 \rightarrow max!$$

$$s.t.$$

$$8 \cdot x_1 + 15 \cdot x_2 + 12 \cdot x_3 \leq 1{,}000$$

$$15 \cdot x_1 + 10 \cdot x_2 + 8 \cdot x_3 \leq 1{,}000$$

$$45 \cdot y_1 \leq x_1 \leq 250 \cdot y_1$$

$$45 \cdot y_2 \leq x_2 \leq 240 \cdot y_2$$

$$45 \cdot y_3 \leq x_3 \leq 250 \cdot y_3$$

$$x_1 \in \{0,1,\ldots,250\}$$

$$x_2 \in \{0,1,\ldots,240\}$$

$$x_3 \in \{0,1,\ldots,250\}$$

$$y_j \in \{0,1\} \quad ; j = 1(1)3$$

Using the CmplData file `production-mix-data.cdat` the CMPL model `production-mix-fc.cmpl` is formulated as follows:

```
%data production-mix-data.cdat
par:
  c := price-costs;
var:
  {j in products : x[j]: integer[0..xMax[j]]; }
  y[products] : binary;
```

```
obj:
  profit: c^T  * x - FC^T * y ->max;

con:
  res:  a * x <= b;
  bounds {j in products: xMin[j] * y[j] <= x[j] <= xMax[j] * y[j]; }
```

CMPL command:

```
cmpl production-mix-fc.cmpl
```

Solution:

```
-------------------------------------------------------------------------------------------------------
Problem              production-mix-fc.cmpl
Nr. of variables     6
Nr. of constraints   8
Objective name       profit
Solver name          CBC
Display variables    (all)
Display constraints  (all)
-------------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      4880 (max!)

Variables
Name                 Type            Activity         Lower bound         Upper bound            Marginal
-------------------------------------------------------------------------------------------------------
x[1]                    I                   0                   0                 250                   -
x[2]                    I                  66                   0                 240                   -
x[3]                    I                   0                   0                 250                   -
y[1]                    B                   0                   0                   1                   -
y[2]                    B                   1                   0                   1                   -
y[3]                    B                   0                   0                   1                   -
-------------------------------------------------------------------------------------------------------

Constraints
Name                 Type            Activity         Lower bound         Upper bound            Marginal
-------------------------------------------------------------------------------------------------------
res[1]                  L                 990                -inf                1000                   -
res[2]                  L                 660                -inf                1000                   -
bounds[1,1]             L                   0                -inf                   0                   -
bounds[1,2]             G                   0                   0                 inf                   -
bounds[2,1]             L                 -21                -inf                   0                   -
bounds[2,2]             G                 174                   0                 inf                   -
bounds[3,1]             L                   0                -inf                   0                   -
bounds[3,2]             G                   0                   0                 inf                   -
-------------------------------------------------------------------------------------------------------
```

2.4.1.4 Production mix with user-defined functions for thresholds and step-fixed costs

Since the formulations for step-fixed costs and semi-continuous variables can be used in several models, it makes sense to specify the formulations in user-defined functions which can be included in a Cmpl model.

The following listings shows the formulation of step-fixed costs:

```
fixcosts := &{ @v = $arg[1],   @f = $arg[2],@m = $arg[3]:
   // variable, fixed costs, big M
   { f == 0: return 0; }

   local var b := binary;
   con v <= m * b;
   return f*b;
};
```

With the expression `fixcosts := &{ … }`, the code block `&{ … }` is assigned to the symbol `fixcosts`. The user-defined function can be called under this name. Three code block symbols are assigned the arguments of the function. The symbol `v` is assigned the variable for which step-fixed costs are incurred if it is greater than zero. The second argument is the stored step-fixed cost that will be stored in `f`. The last symbol `m` stands for a `big-M` value. If the step-fixed cost equal to zero then the function return the value zero (`{ f == 0: return 0; }`). A local binary variable `b` is then defined. This variable is used for the following constraint `v <= m * b;`. In the last step, the function returns `f*b`. Since this function is to be be called within an objective function, the result of this function extends the corresponding objective function. Although the variable and the constraint have a local scope, they are contained in the entire matrix of the LP.

A function for semi-continuous(integer) variables can be formulated as follows:

```
semicont := &{ @tp = $arg[1], @lb = $arg[2], @ub = $arg[3]:
   // data type, threshold value, upper bound

   local var v := tp[0..ub];
  con res :=  v = 0 || v >= lb;
   return v;
};
```

The symbol `semicont` is assigned the user-defined function which can be called under this name. The arguments are the type of the variable, the threshold for the variable and the upper bound. The local symbol `tp` is used for the type, `lb` for the threshold and `ub` for the upper limit. A local variable v is defined with the type tp, a lower bound of zero and the upper bound ub (`local var v := tp[0..ub];`). In the last step, the function returns this variable (`return v;`). The behaviour that the variable `v` is either equal to zero or lies in the interval between the threshold value `lb` and the upper bound `ub` is formulated as an alternative constraint `v = 0 || v >= lb`.

Both functions mare saved in a file `production-mix-lib.cmpl` which is included in the Cmpl model `production-mix-fc-func.cmpl`. In the first line, the CmplData file from the previous example was read into the model. In the second line, the `production-mix-lib.cmpl` file is included, which contains the two functions.

```
%data production-mix-data.cdat
%include production-mix-lib.cmpl
par:
   c := price-costs;
   bigM := max(xMax);

var:
   {j in products :
     x[j] : semicont( integer, xMin[j], xMax[j]) ;
   }
obj:
   profit: sum{ j in products : c[j]  * x[j] - fixcosts( x[j], FC[j], bigM)  }->max;
con:
   res:  a * x <= b;
```

The parameter section contains the calculation of the cost vector as result of a vector subtraction (`c := price-costs;`) and the parameter bigM which is equal to the maximum of all values in the vector xMax. All elements of the vector of variables x are defined as semi-integer variables using the function *`semi-cont()`*. The objective function is also defined using a sum-loop over all products, whereby the step-fixed costs are included by calling the user-defined function *`fixcosts()`*.

Running the problem using CMPL command:

```
cmpl  production-mix-fc-func.cmpl
```

leads to the solution which is equal to the solution of the previous example:

```
-------------------------------------------------------------------------------------------------------------
Problem              production-mix-fc-func.cmpl
Nr. of variables     12
Nr. of constraints   17
Objective name       profit
Solver name          CBC
Display variables    (all)
Display constraints  (all)
-------------------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      4880 (max!)

Variables
Name                 Type            Activity         Lower bound         Upper bound            Marginal
-------------------------------------------------------------------------------------------------------------
x[1]                    I                   0                   0                 250                   -
x[2]                    I                  66                   0                 240                   -
x[3]                    I                   0                   0                 250                   -
_c1                     B                   0                   0                   1                   -
_c2                     B                   1                   0                   1                   -
_c3                     B                   0                   0                   1                   -
-------------------------------------------------------------------------------------------------------------

Constraints
Name                 Type            Activity         Lower bound         Upper bound            Marginal
-------------------------------------------------------------------------------------------------------------
line_4                  G                   0                   0                 inf                   -
line_5                  G                 184                   0                 inf                   -
line_6                  G                   0                   0                 inf                   -
res[1]                  L                 990                -inf                1000                   -
res[2]                  L                 660                -inf                1000                   -
line_10                 G                   0                   0                 inf                   -
```

```
line_11                L                   0                -inf                   0                   -
line_12                G               1e+10                  45                 inf                   -
line_14                G               1e+10                   0                 inf                   -
line_15                L              -1e+10                -inf                   0                   -
line_16                G                  66                  45                 inf                   -
line_18                G                   0                   0                 inf                   -
line_19                L                   0                -inf                   0                   -
line_20                G               1e+10                  45                 inf                   -
-------------------------------------------------------------------------------------------------------------
```

Unfortunately, the name of the variables and constraints constructed with the two user-defined functions are generated by using default names (e.g. `c1` and `line_4`). Therefore, it makes sense to generated more meaningful names within the functions. Otherwise, a user may not be interested in seeing the generated auxiliary variables and constraints. Hiding these elements can be done by defining names that start with two underscores and are not shown in the solution by default.

The function for the step-fixed costs can be extended as follows:

```
fixcosts := &{ @v = $arg[1],   @f = $arg[2],@m = $arg[3] :
   // variable, fixed costs, big M
   { f == 0: return 0; }

   local var b := binary;
   b.$destNameTuple ::= ["__b", v.$destTuple];

   local con res := v <= m * b;
   res.$destNameTuple ::= ["__b_ub", v.$destTuple];

   return f*b;
};
```

The definition of the variable `b` is followed by the definition of its name and the index in the matrix of the entire LP. The attribute `$destNameTuple` of this variable is assigned the name `"__b"` and as index the index of the variable `v` using its attribute `$destTuple`. A similar approach is used for the name and the index of the constraint `res`.

The extended function for the semi-continuous(integer) needs a fourth argument for the index of the variable to be generated:

```
semicont := &{ @tp = $arg[1], @lb = $arg[2], @ub = $arg[3], @idx = $arg[4]:
   // data type, threshold value, upper bound

   local var v := tp[0..ub];
   local var y := binary;

   y.$destNameTuple ::= ["__y", idx];

   local con res1:= y * lb <= v ;
   local con res2:= v <= y * ub;

   res1.$destNameTuple ::= ["__y_lb", idx];
   res2.$destNameTuple ::= ["__y_ub", idx];

   return v;
};
```

This argument is used for the definition of the name and index of the auxiliary variable `y` in the matrix of the LP (`y.$destNameTuple ::= ["__y", idx];`). The the auxiliary constraints follow an alternative way, where for both constraints the names and indices are defined in the same way as the constraints of the set-fixed costs. Since the names of all auxiliary variables and constraints start with two underscores they are not shown in the solution.

```
---------------------------------------------------------------------------------------------------------
Problem              production-mix-fc-func1.cmpl
Nr. of variables     9
Nr. of constraints   11
Objective name       profit
Solver name          CBC
Display variables    (all)
Display constraints  (all)
---------------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      4880 (max!)

Variables
Name                 Type            Activity         Lower bound         Upper bound            Marginal
---------------------------------------------------------------------------------------------------------
x[1]                    I                   0                   0                 250                   -
x[2]                    I                  66                   0                 240                   -
x[3]                    I                   0                   0                 250                   -
---------------------------------------------------------------------------------------------------------

Constraints
Name                 Type            Activity         Lower bound         Upper bound            Marginal
---------------------------------------------------------------------------------------------------------
res[1]                  L                 990                -inf                1000                   -
res[2]                  L                 660                -inf                1000                   -
---------------------------------------------------------------------------------------------------------
```

If these variables and constraints shall be displayed, the header entry `%display generatedElements` is to be used.

```
---------------------------------------------------------------------------------------------------------
Problem              production-mix-fc-func1.cmpl
Nr. of variables     9
Nr. of constraints   11
Objective name       profit
Solver name          CBC
Display variables    (all,generatedElements)
Display constraints  (all,generatedElements)
---------------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      4880 (max!)

Variables
Name                 Type            Activity         Lower bound         Upper bound            Marginal
---------------------------------------------------------------------------------------------------------
x[1]                    I                   0                   0                 250                   -
__y[1]                  B                   0                   0                   1                   -
x[2]                    I                  66                   0                 240                   -
__y[2]                  B                   1                   0                   1                   -
x[3]                    I                   0                   0                 250                   -
__y[3]                  B                   0                   0                   1                   -
__b[1]                  B                   0                   0                   1                   -
__b[2]                  B                   1                   0                   1                   -
__b[3]                  B                   0                   0                   1                   -
---------------------------------------------------------------------------------------------------------
```

```
Constraints
Name                 Type            Activity        Lower bound        Upper bound            Marginal
------------------------------------------------------------------------------------------------------
__y_lb[1]               L                   0               -inf                  0                   -
__y_ub[1]               G                   0                  0                inf                   -
__y_lb[2]               L                 -21               -inf                  0                   -
__y_ub[2]               G                 174                  0                inf                   -
__y_lb[3]               L                   0               -inf                  0                   -
__y_ub[3]               G                   0                  0                inf                   -
__b_ub[1]               G                   0                  0                inf                   -
__b_ub[2]               G                 184                  0                inf                   -
__b_ub[3]               G                   0                  0                inf                   -
res[1]                  L                 990               -inf               1000                   -
res[2]                  L                 660               -inf               1000                   -
------------------------------------------------------------------------------------------------------
```

2.4.1.5 The knapsack problem

Given a set of items with specified weights and values, the problem is to find a combination of items that fills a knapsack (container, room, …) to maximize the value of the knapsack subject to its restricted capacity or to minimize the weight of items in the knapsack subject to a predefined minimum value.

In this example there are 10 boxes, which can be sold on the market at a defined price.

box number	price [€/box]	weight [pounds]
1	100	10
2	80	5
3	50	8
4	150	11
5	55	12
6	20	4
7	40	6
8	50	9
9	200	10
10	100	11

1. What is the optimal combination of boxes if you are seeking to maximize the total sales and are able to carry a maximum of 60 pounds?
2. What is the optimal combination of boxes if you are seeking to minimize the weight of the transported boxes bearing in mind that the minimum total sales must be at least €600?

Model 1: maximize the total sales

The mathematical model can be formulated as follows:

$$100 \cdot x_1 + 80 \cdot x_2 + 50 \cdot x_3 + 150 \cdot x_4 + 55 \cdot x_5 + 20 \cdot x_6 + 40 \cdot x_7 + 50 \cdot x_8 + 200 \cdot x_9 + 100 \cdot x_{10} \rightarrow max!$$
$$s.t.$$
$$10 \cdot x_1 + 5 \cdot x_2 + 8 \cdot x_3 + 11 \cdot x_4 + 12 \cdot x_5 + 4 \cdot x_6 + 6 \cdot x_7 + 9 \cdot x_8 + 10 \cdot x_9 + 11 \cdot x_{10} \leq 60$$
$$x_j \in \{0,1\} \quad ; j = 1(1)10$$

The basic data is saved in the CMPL file `knapsack-data.cdat`:

```
%boxes set < 1..10 >

#weight of the boxes
%w[boxes] < 10 5 8 11 12 4 6 9 10 11 >

#price per box
%p[boxes] <100 80 50 150 55 20 40 50 200 100 >

#max capacity
%maxWeight <60>

#min sales
%minSales <600>
```

A simple CMPL model `knapsack-max.cmpl` can be formulated as follows:

```
%data knapsack-data.cdat
#show only activities unequal to zero in the solution
%display nonZeros


var:
  x[boxes] : binary;
obj:
  sales: p^T * x ->max;
con:
  weight: w^T * x <= maxWeight;
```

CMPL command:

```
cmpl knapsack-max.cmpl
```

Solution:

```
-----------------------------------------------------------------------------------------------------
Problem              knapsack-max.cmpl
Nr. of variables     10
Nr. of constraints   1
Objective name       sales
Solver name          CBC
Display variables    nonzero variables (all)
Display constraints  nonzero constraints (all)
-----------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      700 (max!)
```

```
Variables
Name                 Type            Activity         Lower bound         Upper bound            Marginal
-------------------------------------------------------------------------------------------------------------
x[1]                    B                   1                   0                   1                   -
x[2]                    B                   1                   0                   1                   -
x[3]                    B                   1                   0                   1                   -
x[4]                    B                   1                   0                   1                   -
x[6]                    B                   1                   0                   1                   -
x[9]                    B                   1                   0                   1                   -
x[10]                   B                   1                   0                   1                   -
-------------------------------------------------------------------------------------------------------------

Constraints
Name                 Type            Activity         Lower bound         Upper bound            Marginal
-------------------------------------------------------------------------------------------------------------
weight                  L                  59                -inf                  60                   -
-------------------------------------------------------------------------------------------------------------
```

Model 2: minimize the weight

The mathematical model can be formulated as follows:

$$10 \cdot x_1 + 5 \cdot x_2 + 8 \cdot x_3 + 11 \cdot x_4 + 12 \cdot x_5 + 4 \cdot x_6 + 6 \cdot x_7 + 9 \cdot x_8 + 10 \cdot x_9 + 11 \cdot x_{10} \rightarrow min!$$

$$s.t.$$

$$100 \cdot x_1 + 80 \cdot x_2 + 50 \cdot x_3 + 150 \cdot x_4 + 55 \cdot x_5 + 20 \cdot x_6 + 40 \cdot x_7 + 50 \cdot x_8 + 200 \cdot x_9 + 100 \cdot x_{10} \geq 600$$

$$x_j \in \{0,1\} \quad ; j = 1(1)10$$

A simple CMPL model `knapsack-min-basic.cmpl` can be formulated as follows:

```
%data knapsack-data.cdat
#show only activities unequal to zero in the solution
%display nonZeros

var:
  x[boxes] : binary;
obj:
  weight: w^T * x ->min;
con:
  sales: p^T * x >= minSales;
```

CMPL command:

```
cmpl  knapsack-min.cmpl
```

Solution:

```
-------------------------------------------------------------------------------------------------------------
Problem              knapsack-min.cmpl
Nr. of variables     10
Nr. of constraints   1
Objective name       weight
Solver name          CBC
Display variables    nonzero variables (all)
Display constraints  nonzero constraints (all)
-------------------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      47 (min!)
```

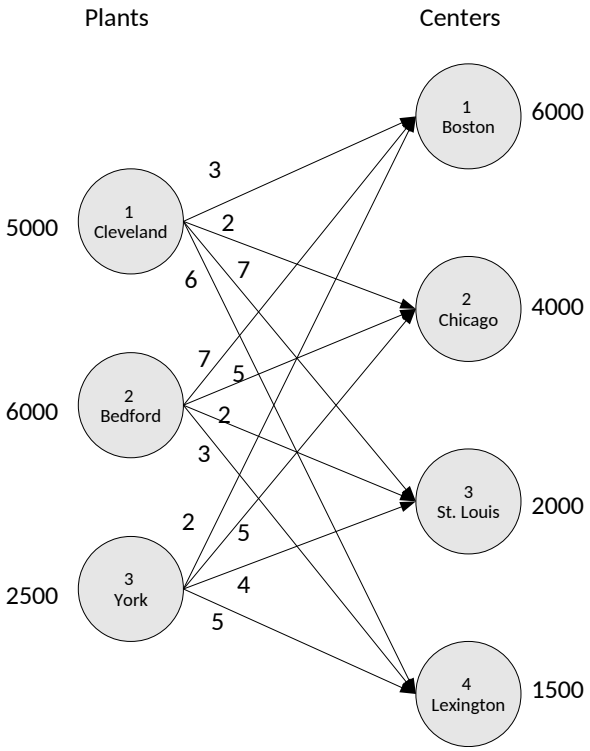
```
Variables
Name                  Type            Activity         Lower bound         Upper bound            Marginal
------------------------------------------------------------------------------------------------------------
x[1]                     B                   1                   0                   1                   -
x[2]                     B                   1                   0                   1                   -
x[4]                     B                   1                   0                   1                   -
x[9]                     B                   1                   0                   1                   -
x[10]                    B                   1                   0                   1                   -
------------------------------------------------------------------------------------------------------------

Constraints
Name                  Type            Activity         Lower bound         Upper bound            Marginal
------------------------------------------------------------------------------------------------------------
sales                    G                 630                 600                 inf                   -
------------------------------------------------------------------------------------------------------------
```

2.4.1.6 The standard transport problem

A transport problem is a special kind of linear programming problem which seeks to minimize the total shipping costs of transporting goods from several supply locations (origins or sources) to several demand locations (destinations).

The following example is taken from (Anderson et.al. 2011, p. 261ff). This problem involves the transportation of a product from three plants to four distribution centres. Foster Generators operates plants in Cleveland, Ohio; Bedford, Indiana; and York, Pennsylvania. The supplies are defined by the production capacities over the next three-month planning period for one particular type of generator.

The company distributes its generators through four regional distribution centres located in Boston, Chicago, St. Louis, and Lexington. It is to decide how much of its products should be shipped from each plant to each distribution centre. The objective is to minimize the transportation costs.



The problem can be formulated in the form of the general linear programme below

$$\sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij} \cdot x_{ij} \quad \rightarrow \min!$$

$$s.t.$$

$$\sum_{j=1}^{n} x_{ij} = s_i \qquad ; i = 1(1)m$$

$$\sum_{i=1}^{m} x_{ij} = d_j \qquad ; j = 1(1)n$$

$$x_{ij} \geq 0 \qquad ; i = 1(1)m\,,\, j = 1(1)n$$

x_{ij} – number of units shipped from plant i to center j
c_{ij} – cost per unit of shipping from plant i to center j
s_i – supply in units at plant i
d_j – demand in units at desitination j

The CMPL model `transportation.cmpl` can be formulated or by using an additional cmplData file `transportation.cdat` as follows:

```
%plants set < 1..3 >
%centres set < 1..4 >

%s[plants] <  5000 6000 2500 >
%d[centres] < 6000 4000 2000 1500 >

%c[plants, centres]  <  3 2 7 6
                        7 5 2 3
                        2 5 4 5 >
```

```
%data transportation.cdat
%display nonZeros

var:
  x[plants,centers]: integer;

obj:
  costs: sum{i in plants, j in centers : c[i,j] * x[i,j] } ->min;

con:
  supplies {i in plants  : sum{j in centers: x[i,j]} = s[i]; }
  demands  {j in centers : sum{i in plants : x[i,j]} =  d[j]; }
```

CMPL command:

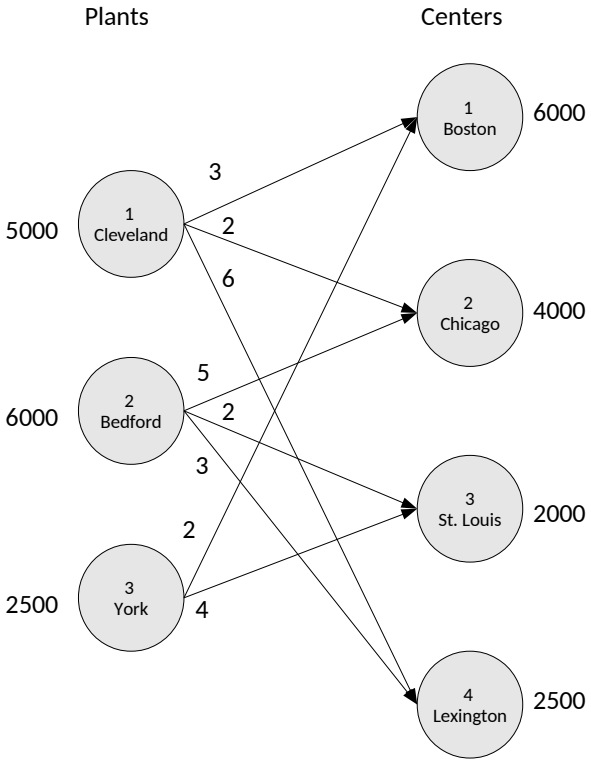
```
cmpl  transportation.cmpl
```

Solution:

```
-------------------------------------------------------------------------------------------------------------
Problem              transportation.cmpl
Nr. of variables     12
Nr. of constraints   7
Objective name       costs
Solver name          CBC
Display variables    nonzero variables (all)
Display constraints  nonzero constraints (all)
-------------------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      39500 (min!)

Variables
Name                 Type            Activity         Lower bound         Upper bound            Marginal
-------------------------------------------------------------------------------------------------------------
x[1,1]                  I                3500                   0                 inf                   -
x[1,2]                  I                1500                   0                 inf                   -
x[2,2]                  I                2500                   0                 inf                   -
x[2,3]                  I                2000                   0                 inf                   -
x[2,4]                  I                1500                   0                 inf                   -
x[3,1]                  I                2500                   0                 inf                   -
-------------------------------------------------------------------------------------------------------------

Constraints
Name                 Type            Activity         Lower bound         Upper bound            Marginal
-------------------------------------------------------------------------------------------------------------
supplies[1]             E                5000                5000                5000                   -
supplies[2]             E                6000                6000                6000                   -
supplies[3]             E                2500                2500                2500                   -
demands[1]              E                6000                6000                6000                   -
demands[2]              E                4000                4000                4000                   -
demands[3]              E                2000                2000                2000                   -
demands[4]              E                1500                1500                1500                   -
-------------------------------------------------------------------------------------------------------------
```

2.4.1.7 Transportation problem using a 2-tuple set

In the case that not all of the connections are possible for technological or commercial reasons (e.g. as in the picture below) then an alternative model to the model above has to be formulated. Additionally is assumed that the total demand is greater than the supplies.



The mathematical model is based on the 2-tuple set routes that contains only the valid connections between the plants and the centres.

$$\sum_{(i,j)\in routes} c_{ij} \cdot x_{ij} \quad \rightarrow \min!$$

$$s.t.$$

$$\sum_{\substack{(k,j)\in routes \\ k=i}} x_{kj} = s_i \quad ; i = 1(1)m$$

$$\sum_{\substack{(i,l)\in routes \\ l=j}} x_{il} \leq d_j \quad ; j = 1(1)n$$

$$x_{ij} \geq 0 \qquad ;(i,j)\in routes$$

Die sets and parameters are specified in `transportation-tuple.cdat`

```
%edges  set[2]  <  1 1  1 2  1 4
                   2 2  2 3   2 4
                   3 1  3 3  >

%plants  set < 1 .. 3 >
%centers  set < 1 .. 4 >
```

```
%s[plants] <  5000 6000 2500 >
%d[centers] < 6000 4000 2000 2500 >

%c[edges]  < 3  2  6  5  2  3  2  4 >
```

that is connected to the CMPL model `transportation-tuple.cmpl`:

```
%data : plants set, centers set, edges set[2], c[edges] , s[plants] , d[centers]
%display nonZeros

var:
  x[edges]: real;
obj:
  costs: sum{ [i,j] in edges : c[i,j]*x[i,j] } ->min;
con:
  supplies {i in plants : sum{j in edges *> [i,*] : x[i,j]} = s[i];}
  demands  {j in centers: sum{i in edges *> [*,j] : x[i,j]} <= d[j];}
```

The two constraints use a set pattern matching to generate the 1-tuple sets used for the sum-loops. An alternative formulation can used as follows:

```
supplies {i in plants : sum{j in centers, [i,j] in edges : x[i,j]} = s[i];}
demands  {j in centers: sum{i in plants, [i,j] in edges: x[i,j]} <= d[j];}
```

In this case, the sum-loops iterate over the entire set of the `centers` or `plants`, but its is checked whether the index-tuple `[i,j]` exists in the 2-tuple set `edges`.

Solution:

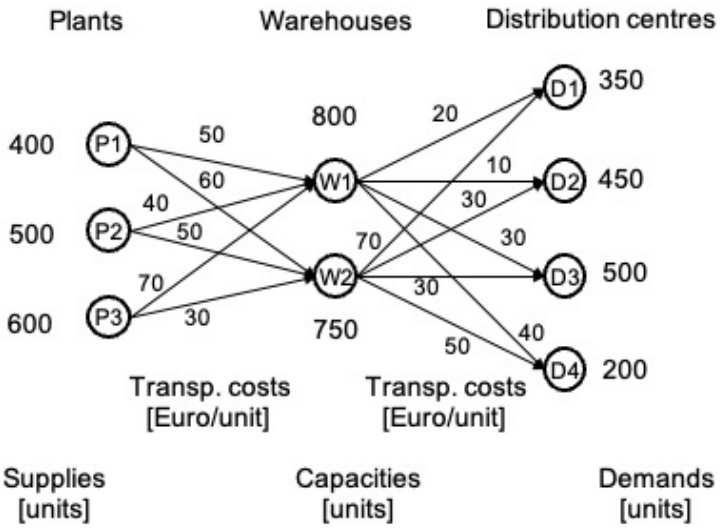
```
---------------------------------------------------------------------------------------------------------
Problem              transportation-tupel.cmpl
Nr. of variables     8
Nr. of constraints   7
Objective name       costs
Solver name          CBC
Display variables    nonzero variables (all)
Display constraints  nonzero constraints (all)
---------------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      36500 (min!)

Variables
Name                 Type            Activity         Lower bound         Upper bound            Marginal
---------------------------------------------------------------------------------------------------------
x[1,1]                  C                2500                   0                 inf                   0
x[1,2]                  C                2500                   0                 inf                   0
x[2,2]                  C                1500                   0                 inf                   0
x[2,3]                  C                2000                   0                 inf                   0
x[2,4]                  C                2500                   0                 inf                   0
x[3,1]                  C                2500                   0                 inf                   0
---------------------------------------------------------------------------------------------------------
```

```
Constraints
Name                 Type            Activity        Lower bound        Upper bound            Marginal
------------------------------------------------------------------------------------------------------
supplies[1]             E                5000               5000               5000                   3
supplies[2]             E                6000               6000               6000                   6
supplies[3]             E                2500               2500               2500                   2
demands[1]              L                5000               -inf               6000                   0
demands[2]              L                4000               -inf               4000                  -1
demands[3]              L                2000               -inf               2000                  -4
demands[4]              L                2500               -inf               2500                  -3
------------------------------------------------------------------------------------------------------
```

2.4.1.8 Transhipment problem

Logistical networks (e.g. distribution networks) often contain so called transhipment nodes beside sources and destination. A transhipment node has to assemble or divide the incoming shipments into the outgoing shipments. That means the incoming quantity has to be equal to the outgoing quantity. A transhipment model is intended to organise an optimal supply of a homogeneous good between a set of sources (origins, suppliers), a set of transhipment nodes and a set of sinks (destinations, customers) in order to minimise the total transportation cost (or distances, times, etc.).

In this example, a transport plan between three plants, two warehouses and four distribution centres is to determined in order to minimise the total transport costs. The unit transport costs are shown in the picture below as weights at the edges. The capacity of each possible road (edge) is restricted to 500 units due to the vehicle pool.



The first step is to determine the data (records and parameters) of the problem in a CmplData file `transhipment.cdat`. Please note that the transshipment nodes `W1` and `W2` have to be split (`W1a`, `W1b`, `W2a`, `W2b`) due to their capacities and the fact that the min-cost flow model does not allow capacities for nodes. Therefore, each transshipment node must be split into two, with a cost-free edge connecting the two. The maximum flow on such an edge equals the capacity of the transhipment node. Consequently, the

definition of the 2-tuple set edges also contains these two auxiliary edges `W1a` to `W1b` and `W2a` to `W2b` in addition to the normal edges.

```
%nodes set  < P1 P2 P3 W1a W2a W1b W2b D1 D2 D3 D4 >

%edges set[2] <
P1      W1a
P1      W2a
P2      W1a
P2      W2a
P3      W1a
P3      W2a
W1a     W1b
W1b     D1
W1b     D2
W1b     D3
W1b     D4
W2a     W2b
W2b     D1
W2b     D2
W2b     D3
W2b     D4
>

#supplies of the nodes
%s[nodes] = 0 indices <
P1      400
P2      500
P3      600
>

#demand of the nodes
%d[nodes] = 0 indices <
D1      350
D2      450
D3      500
D4      200
>

#unit transport costs per edge
%c[edges] = 0 indices <
P1      W1a     50
P1      W2a     60
P2      W1a     40
P2      W2a     50
P3      W1a     70
P3      W2a     30
W1b     D1      20
W1b     D2      10
W1b     D3      30
W1b     D4      40
W2b     D1      70
W2b     D2      30
W2b     D3      30
W2b     D4      50
>

#max flow on the edges
%maxCap[edges] = 500 indices <
W1a W1b 800.0
W2a W2b 750.0
>
```

The supply vector `s` contains a supply greater than zero only for the sources. Therefore, the definition of this vector starts with a default value equal to zero (`%s[nodes] =0`). All other values have to be indicated by their index (keyword `indices`). Each entry starts with the index of the node followed by its supply. All other arrays are specified in this way. In particular, the default value of the vector `maxCap` for the maximum flow of all edges is equal to 500, which corresponds to the capacity of the vehicle used. Only the edges

between the split transshipment nodes have a different upper bound due to the capacities of the warehouses.

This CmplData file have to be read into the Cmpl file `transhipment.cmpl` by using the Cmpl header entry `%data` in the first line of the following listing:

```
%data : nodes set, s[nodes], d[nodes], edges set[2], c[edges], maxCap[edges]

var:
    { [i,j] in edges: x[i,j] : real[0..maxCap[i, j]];  }

obj:
    costs: sum { [i,j] in edges: c[i,j] * x[i,j] } ->min;

con:
    { i in nodes :
          netFlow[i]:   sum{ j in edges *> [i,*] : x[i,j] } -
                        sum{ j in edges *> [*,i] : x[j,i] } = s[i] - d[i];
    }
```

The variables of the model are organised in an array `x` which is defined by using the 2-tuple set `edges`. They are all non-negative continuous variables with an upper bound defined in the vector `maxCap`. These variables are the flows of the uniform good on the edges. An objective function `costs` to be minimised is defined in the objective section as the sum over all `edges` of the product of the unit transport costs `c[i,j]` and the flow `x[i,j]` on the edge. For all nodes, a flow balance constraint `netFlow[i]` has to be created in which the difference of the outgoing and incoming flow on the left-hand side must be equal to the difference of the supply `s[i]` and the demand `d[i]` of this node on the right-hand side.

After running this problem, the following solution can be found.

```
-------------------------------------------------------------------------------------------------------------
Problem              transhipment.cmpl
Nr. of variables     16
Nr. of constraints   11
Objective name       costs
Solver name          CBC
Display variables    (all)
Display constraints  (all)
-------------------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      100500 (min!)

Variables
Name                 Type            Activity        Lower bound        Upper bound            Marginal
-------------------------------------------------------------------------------------------------------------
x[P1,W1a]               C                 200                  0                500                   0
x[P1,W2a]               C                 200                  0                500                   0
x[P2,W1a]               C                 500                  0                500                   0
x[P2,W2a]               C                   0                  0                500                   0
x[P3,W1a]               C                 100                  0                500                   0
x[P3,W2a]               C                 500                  0                500                 -50
x[W1a,W1b]              C                 800                  0                800                 -20
x[W2a,W2b]              C                 700                  0                750                   0
x[W1b,D1]               C                 350                  0                500                   0
x[W1b,D2]               C                 450                  0                500                   0
x[W1b,D3]               C                   0                  0                500                  10
x[W1b,D4]               C                   0                  0                500                   0
x[W2b,D1]               C                   0                  0                500                  40
x[W2b,D2]               C                   0                  0                500                  10
```

```
x[W2b,D3]              C                500                0                500                0
x[W2b,D4]              C                200                0                500                0
------------------------------------------------------------------------------------------------------------

Constraints
Name                   Type          Activity        Lower bound        Upper bound          Marginal
------------------------------------------------------------------------------------------------------------
netFlow[P1]            E                400                400                400                 70
netFlow[P2]            E                500                500                500                 60
netFlow[P3]            E                600                600                600                 90
netFlow[W1a]           E                  0                  0                  0                 20
netFlow[W2a]           E                  0                  0                  0                 10
netFlow[W1b]           E                  0                  0                  0                  0
netFlow[W2b]           E                  0                  0                  0                 10
netFlow[D1]            E               -350               -350               -350                -20
netFlow[D2]            E               -450               -450               -450                -10
netFlow[D3]            E               -500               -500               -500                -20
netFlow[D4]            E               -200               -200               -200                -40
------------------------------------------------------------------------------------------------------------
```

2.4.1.9 Transhipment problem using Excel via CmplXlsData

In this section the previous example is solved again but the data is to be read from an Excel sheet and the solution is to be written into it. This can be done by Cmpl's CmplXlsData interface. In the first step, a CmplXlsData file `transhipment1.xdat` is to be created instead an CmplData file as in the previous section. This is related to an Excel file `transhipment.xlsx` containing the sheet `transhipment`.

	A	B	C	D	E	F	G	H	I	J	K	L
1	**Nodes**					**Edges**						
2		supplies	demands			from	to	cost rate	min. cap.	max. cap	flow	costs
3	P1	400	0			P1	W1a	50	0	500		0
4	P2	500	0			P1	W2a	60	0	500		0
5	P3	600	0			P2	W1a	40	0	500		0
6	W1a	0	0			P2	W2a	50	0	500		0
7	W2a	0	0			P3	W1a	70	0	500		0
8	W1b	0	0			P3	W2a	30	0	500		0
9	W2b	0	0			W1a	W1b	0	0	800		0
10	D1	0	350			W1b	D1	20	0	500		0
11	D2	0	450			W1b	D2	10	0	500		0
12	D3	0	500			W1b	D3	30	0	500		0
13	D4	0	200			W1b	D4	40	0	500		0
14						W2a	W2b	0	0	750		0
15	total costs	100.500				W2b	D1	70	0	500		0
16						W2b	D2	30	0	500		0
17						W2b	D3	30	0	500		0
18						W2b	D4	50	0	500		0

The CmplXlsData file starts in the `source` section with the file `transhipment.xlsx` and the sheet `transhipment` from which the data is to be read and the results written.

```
@source
      %file < transhipment.xlsx >
      %sheet < transhipment>

@input

      %edges set[2] < F3:G18 >
      %nodes set < A3:A13 >
```

```
      %c[edges] < H3:H18 >
      %d[nodes] < C3:C13 >
      %s[nodes] < B3:B13 >

      %minCap[edges] < I3:I18  >
      %maxCap[edges] < J3:J18 >

@output
      %x[edges].activity < K3:K18 >
      %objValue < B15 >
```

The definition of the sets and the parameter arrays in the input section are similar to the corresponding definitions in the CmplData file in the previous section. The only difference is that the data cannot be specified within the angle brackets. In CmplXlsData the cell ranges have to be defined embedded in angle brackets. The values of the set edges, for example, are stored in the cells `F3:G18`. The output section is intended to specify the result elements to be written to the specified Excel sheet. Here, the activities of the flow variables x have to be written into the cells K3:K18. In addition, the value of the objective function after optimisation is to be found in cell `B15`.

The only difference to the previous Cmpl model is the first line. Instead of %data the entry %xlsdata is to be used.

```
%xlsdata : nodes set, s[nodes], d[nodes], edges set[2], c[edges], maxCap[edges]
```

The results can be found after the optimisation in the cells specified in the CmplXlsData file. It is the same solution as in the previous section, but now available in Excel (on Windows or macOS).

	A	B	C	D	E	F	G	H	I	J	K	L
1	**Nodes**					**Edges**						
2		supplies	demands			from	to	cost rate	min. cap.	max. cap	flow	costs
3	P1	400	0			P1	W1a	50	0	500	200	10.000
4	P2	500	0			P1	W2a	60	0	500	200	12.000
5	P3	600	0			P2	W1a	40	0	500	500	20.000
6	W1a	0	0			P2	W2a	50	0	500	0	0
7	W2a	0	0			P3	W1a	70	0	500	100	7.000
8	W1b	0	0			P3	W2a	30	0	500	500	15.000
9	W2b	0	0			W1a	W1b	0	0	800	800	0
10	D1	0	350			W1b	D1	20	0	500	350	7.000
11	D2	0	450			W1b	D2	10	0	500	450	4.500
12	D3	0	500			W1b	D3	30	0	500	0	0
13	D4	0	200			W1b	D4	40	0	500	0	0
14						W2a	W2b	0	0	750	700	0
15	total costs	100.500				W2b	D1	70	0	500	0	0
16						W2b	D2	30	0	500	0	0
17						W2b	D3	30	0	500	500	15.000
18						W2b	D4	50	0	500	200	10.000

2.4.1.10 Assignment problem

The following simple assignment problem is to be solved. A dispatcher has to plan the express transports of a homogeneous good starting from the three stations (S1-S3) to the four customers (D1-D4) for the next day in order to minimise the total transportation times.

	Transportation times [h]			
	D1	D2	D3	D4
S1	12	25	2	-
S2	20	-	12	-
S3	30	6	10	5

Which station should supply which customer in order to minimise the transportation time?

There are two problems. An assignment problem usually contains two groups of strong SOS1 constraints. A station must serve exactly one customer aand a customer should be served by exactly one station. Since there are more customers then stations, only three of the customers can be served. Therefore, a strong SOS1 must be formulated for each station and a week SOS1 for each of the customers. This means that a customer can be served by a maximum of one station. The second problem is that three of the possible assignments are not allowed and are therefore marked with a hyphen in the matrix. This can be done by formulating a set of allowed combinations via a 2-tuple set or by a Big M approach, i.e. very high assignment costs for the forbidden combinations.

Die sets and parameters are specified in `assignment.cdat`, where the assignment costs for the forbidden combinations are equal to `1000`.

```
%N2 set < D1 D2  D3 D4 >
%N1 set < S1 S2 S3 >
%c[N1,N2]   <
  12   25   2    1000
  20   1000 12   1000
  30   6    10   5 >
```

The CMPL model `assignment.cmpl` can be formulated as follows:

```
%data : N1 set, N2 set, c[N1,N2]
%display nonZeros


var:
  x[N1,N2]: real[0..1];

obj:
  sum{ i in N1, j in N2: c[i,j]*x[i,j] } -> min ;

con:
  sos_N1_ { i in N1: sum{ j in N2: x[i,j] } = 1; }
  sos_N2_ { j in N2: sum{ i in N1: x[i,j] } <= 1; }
```

The CmplData file was read in with %data in the first line. Since the Big M values are used for the assignment costs for the forbidden combinations, the variables are defined over all combinations from the set of wards N1 and the set of customers N2. The obj section defines that the total assignment costs must be minimised over all combinations N1xN2. The set of constraints named sos_N1_ defines the strong SOS1 for the stations, while the constraints sos_N2_ define the weak SOS1 for the customers.

After running this problem, the following solution can be found.

```
---------------------------------------------------------------------------------------------------------------
Problem              assignment.cmpl
Nr. of variables     12
Nr. of constraints   7
Objective name       line_1
Solver name          CBC
Display variables    nonzero variables (all)
Display constraints  nonzero constraints (all)
---------------------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      27 (min!)

Variables
Name                 Type            Activity         Lower bound         Upper bound            Marginal
---------------------------------------------------------------------------------------------------------------
x[S1,D3]                C                   1                   0                   1                   0
x[S2,D1]                C                   1                   0                   1                   0
x[S3,D4]                C                   1                   0                   1                   0
---------------------------------------------------------------------------------------------------------------

Constraints
Name                 Type            Activity         Lower bound         Upper bound            Marginal
---------------------------------------------------------------------------------------------------------------
sos_N1_[S1]             E                   1                   1                   1                  10
sos_N1_[S2]             E                   1                   1                   1                  20
sos_N1_[S3]             E                   1                   1                   1                   5
sos_N2_[D1]             L                   1                -inf                   1                   0
sos_N2_[D3]             L                   1                -inf                   1                  -8
sos_N2_[D4]             L                   1                -inf                   1                   0
---------------------------------------------------------------------------------------------------------------
```

Station S1 serves Customer D3, S2 takes over D1 and customer D4 is served by station S3. All constraints are satisfied.

Since Cplex, Gurobi, Scip and Cbc are able to use weak SOS1 and SOS2 directly, CMPL automatically generates native SOS when one of these solvers is selected.

The CMPL model `assignment1.cmpl` is silimar to the model above, but uses CMPL's predefined SOS1 class.

```
%data : N1 set, N2 set, c[N1,N2]
%display nonZeros


var:
   x[N1,N2]: real[0..1];

obj:
   sum{ i in N1, j in N2: c[i,j]*x[i,j] } -> min ;

con:
   sos_N1_ { i in N1: sum{ j in N2: x[i,j] } = 1; }

par:
   { j in N2:
      s[j] := sos.sos1().name("sos_N1_");
      s[j].add( x[,j] );
   }
```

Instead of the previous constraints `sos_N1_`, the class `sos1()` is now used within the last `par` section. For each customer `i in N2`, a SOS1 object is created and assigned to the parameter `s[j]`. Then the column for this customer `j` of the matrix of assignment variables `x` is added to this SOS1 object.

Since Cbc supports SOS directly, running this problem leads to the same solution as before, but with only 3 constraints instead of 7 in the previous solution.

```
-------------------------------------------------------------------------------------------------------------
Problem              assignment1.cmpl
Nr. of variables     12
Nr. of constraints   3
Objective name       line_1
Solver name          CBC
Display variables    nonzero variables (all)
Display constraints  nonzero constraints (all)
-------------------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      27 (min!)

Variables
Name                 Type            Activity         Lower bound         Upper bound             Marginal
-------------------------------------------------------------------------------------------------------------
x[S1,D3]                C                   1                   0                   1                   -
x[S2,D1]                C                   1                   0                   1                   -
x[S3,D4]                C                   1                   0                   1                   -
-------------------------------------------------------------------------------------------------------------

Constraints
Name                 Type            Activity         Lower bound         Upper bound             Marginal
-------------------------------------------------------------------------------------------------------------
sos_N1_[S1]             E                   1                   1                   1                   -
sos_N1_[S2]             E                   1                   1                   1                   -
sos_N1_[S3]             E                   1                   1                   1                   -
-------------------------------------------------------------------------------------------------------------
```

2.4.1.11 Quadratic assignment problem

Assignment problems are special types of linear programming problems which assign assignees to tasks or locations. The goal of this quadratic assignment problem is to find the cheapest assignments of *n* machines to *n* locations. The transport costs are influenced by

- the distance d_{jk} between location *j* and location *k* and
- the quantity t_{hi} between machine *h* and machine *i*, which is to be transported.

The assignment of a machine *h* to a location *j* can be formulated with the Boolean variables

$$x_{hj} = \begin{Bmatrix} 1 & \text{, if machine } h \text{ is assigned to location } j \\ 0 & \text{, if not} \end{Bmatrix}$$

The general model can be formulated as follows:

$$\sum_{h=1}^{n} \sum_{i=1 i \neq h}^{n} \sum_{j=1}^{n} \sum_{k=1 k \neq j}^{n} t_{hi} \cdot d_{jk} \cdot x_{hj} \cdot x_{ik} \rightarrow min!$$

$$s.t.$$

$$\sum_{j=1}^{n} x_{hj} = 1; h = 1(1)n$$

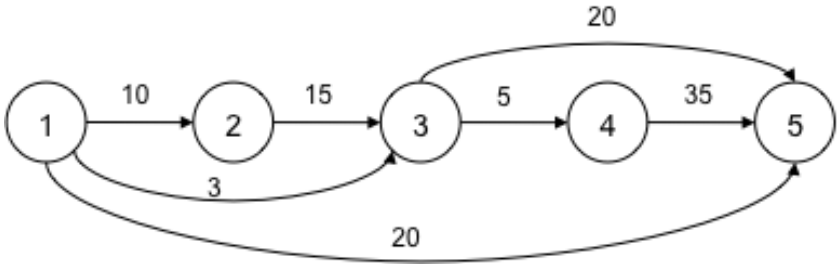
$$\sum_{h=1}^{n} x_{hj} = 1; j = 1(1)n$$

$$x_{hj} \in \{0,1\}; h = 1(1)n, j = 1(1)n$$

Because of the product $x_{hj} \cdot x_{ik}$ in the objective function the model is quadratic programming problem (QP). If the solver called does not support QP (CBC, GLPK), then the products of these variables are equivalently

transformation into a set of inequations by CMPL. If the solver supports quadratic optimisation (Cplex, Gurobi, Scip), then the linearisation is switched off automatically and the QP algorithm is used.

Consider the following case: There are 5 machines and 5 locations in the given factory. The quantities of goods which are to be transported between the machines are indicated in the figure below.



As shown in the picture below the machines are not fully connected. Therefore it makes sense to formulate the objective function with a sum over a 2-tuple set with the name `edges` for the valid combinations between the machines.

$$\sum_{(h,i)\in edges}^{n} \sum_{j=1}^{n} \sum_{k=1 k\neq j}^{n} t_{hi} \cdot d_{jk} \cdot x_{hj} \cdot x_{ik} \rightarrow min!$$

The distances between the locations are given in the following table:

from/to	1	2	3	4	5
1	M	1	2	3	4
2	2	M	1	2	3
3	3	1	M	1	2
4	2	3	1	M	1
5	5	3	2	1	M

The CMPL model `quadratic-assignment.cmpl` can be formulated as follows:

```
%display nonZeros

par:
      n:=5;
      d[,]:= (    ( 0, 1, 2, 3, 4),
                  ( 2, 0, 1, 2, 3),
                  ( 3, 1, 0, 1, 2),
                  ( 2, 3, 1, 0, 1),
                  ( 5, 3, 1, 1, 0) );

      edges := set ( [1,2] , [1,3], [1,5], [2,3] , [3,4] , [3,5] , [4,5]);
      t[edges] := (10,3,20,15,5,20,35);

var:
      x[1..n,1..n]: binary;

obj:
      costs: sum{  [h,i] in edges, j in 1..n,  k in 1..n , k<>j:
                  t[h,i]*d[j,k]*x[h,j]*x[i,k] } →min;
```

```
con:
      location { h in 1..n: sum{ j in 1..n: x[h,j] } = 1; }
      machine { j in 1..n: sum{  h in 1..n: x[h,j] } = 1; }
```

If this problem CMPL was run with CBC

```
Cmpl quadratic-assignment.cmpl
```

then the following solution are found.

```
--------------------------------------------------------------------------------------------------------------
Problem              quadratic-assignment.cmpl
Nr. of variables     165
Nr. of constraints   430
Objective name       costs
Solver name          CBC
Display variables    nonzero variables (all)
Display constraints  nonzero constraints (all)
--------------------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      134 (min!)

Variables
Name                 Type            Activity         Lower bound         Upper bound            Marginal
--------------------------------------------------------------------------------------------------------------
x[1,4]                  B                   1                   0                   1                   -
x[2,1]                  B                   1                   0                   1                   -
x[3,2]                  B                   1                   0                   1                   -
x[4,5]                  B                   1                   0                   1                   -
x[5,3]                  B                   1                   0                   1                   -
--------------------------------------------------------------------------------------------------------------

Constraints
Name                 Type            Activity         Lower bound         Upper bound            Marginal
--------------------------------------------------------------------------------------------------------------
location[1]             E                   1                   1                   1                   -
location[2]             E                   1                   1                   1                   -
location[3]             E                   1                   1                   1                   -
location[4]             E                   1                   1                   1                   -
location[5]             E                   1                   1                   1                   -
machine[1]              E                   1                   1                   1                   -
machine[2]              E                   1                   1                   1                   -
machine[3]              E                   1                   1                   1                   -
machine[4]              E                   1                   1                   1                   -
machine[5]              E                   1                   1                   1                   -
--------------------------------------------------------------------------------------------------------------
```

The optimal assignments of machines to locations are given in the table below:

machines \ locations	1	2	3	4	5
1				x	
2	x				
3		x			
4					x
5			x		

The problem size is 430 constraints and 165 variables including all auxiliary variables and constraints which are not shown in the solution by default.

If the CMPL is run with Cplex

```
%solver cplex
```

then a problem with only 10 constraints and 25 variables are generated and the same solution is found much faster:

```
------------------------------------------------------------------------------------------------------------
Problem              quadratic-assignment.cmpl
Nr. of variables     25
Nr. of constraints   10
Objective name       costs
Solver name          CPLEX
Display variables    nonzero variables (all)
Display constraints  nonzero constraints (all)
------------------------------------------------------------------------------------------------------------

Objective status     integer optimal solution
Objective value      134 (min!)

Variables
Name                 Type            Activity         Lower bound         Upper bound            Marginal
------------------------------------------------------------------------------------------------------------
x[1,4]                  B                   1                   0                   1                   -
x[2,1]                  B                   1                   0                   1                   -
x[3,2]                  B                   1                   0                   1                   -
x[4,5]                  B                   1                   0                   1                   -
x[5,3]                  B                   1                   0                   1                   -
------------------------------------------------------------------------------------------------------------

Constraints
Name                 Type            Activity         Lower bound         Upper bound            Marginal
------------------------------------------------------------------------------------------------------------
location[1]             E                   1                   1                   1                   -
location[2]             E                   1                   1                   1                   -
location[3]             E                   1                   1                   1                   -
location[4]             E                   1                   1                   1                   -
location[5]             E                   1                   1                   1                   -
machine[1]              E                   1                   1                   1                   -
machine[2]              E                   1                   1                   1                   -
machine[3]              E                   1                   1                   1                   -
machine[4]              E                   1                   1                   1                   -
machine[5]              E                   1                   1                   1                   -
------------------------------------------------------------------------------------------------------------
```

2.4.1.12 Quadratic assignment problem using the `solutionPool` option

It is for several reasons interesting to catch the feasible integer solutions found during a linear MIP (QP) optimisation. Gurobi and Cplex are able to generate and store multiple solutions for such a problem. With the display option `solutionPool` these feasible integer solutions can be shown in the solution report. It is recommended to control the behaviour of the solution pool by setting the particular Gurobi or Cplex solver options.

If the CMPL model for quadratic assignment problem above is extended by the following CMPL header entries, then all feasible integer solutions found by Cplex are shown in the solution. The option `%display ignoreCons` is intended to hide the constraints from the solution.

```
%solver cplex
%display solutionPool
%display ignoreCons
```

Solution:

```
-------------------------------------------------------------------------------------------------------------
Problem              quadratic-assignment.cmpl
Nr. of variables     25
Nr. of constraints   10
Objective name       costs
Nr. of solutions     5
Solver name          CPLEX
Display variables    nonzero variables (all)
Display constraints  nonzero constraints (all)
-------------------------------------------------------------------------------------------------------------

Solution nr.         1
Objective status     integer optimal solution
Objective value      134 (min!)

Variables
Name                 Type            Activity         Lower bound         Upper bound            Marginal
-------------------------------------------------------------------------------------------------------------
x[1,4]                  B                   1                   0                   1                   -
x[2,1]                  B                   1                   0                   1                   -
x[3,2]                  B                   1                   0                   1                   -
x[4,5]                  B                   1                   0                   1                   -
x[5,3]                  B                   1                   0                   1                   -
-------------------------------------------------------------------------------------------------------------

Constraints
Name                 Type            Activity         Lower bound         Upper bound            Marginal
-------------------------------------------------------------------------------------------------------------
location[1]             E                   1                   1                   1                   -
location[2]             E                   1                   1                   1                   -
location[3]             E                   1                   1                   1                   -
location[4]             E                   1                   1                   1                   -
location[5]             E                   1                   1                   1                   -
machine[1]              E                   1                   1                   1                   -
machine[2]              E                   1                   1                   1                   -
machine[3]              E                   1                   1                   1                   -
machine[4]              E                   1                   1                   1                   -
machine[5]              E                   1                   1                   1                   -
-------------------------------------------------------------------------------------------------------------

Solution nr.         2
Objective status     integer feasible solution
Objective value      134 (min!)

Variables
Name                 Type            Activity         Lower bound         Upper bound            Marginal
-------------------------------------------------------------------------------------------------------------
x[1,4]                  B                   1                   0                   1                   -
x[2,1]                  B                   1                   0                   1                   -
x[3,2]                  B                   1                   0                   1                   -
x[4,5]                  B                   1                   0                   1                   -
x[5,3]                  B                   1                   0                   1                   -
-------------------------------------------------------------------------------------------------------------

Constraints
Name                 Type            Activity         Lower bound         Upper bound            Marginal
-------------------------------------------------------------------------------------------------------------
location[1]             E                   1                   1                   1                   -
location[2]             E                   1                   1                   1                   -
location[3]             E                   1                   1                   1                   -
location[4]             E                   1                   1                   1                   -
location[5]             E                   1                   1                   1                   -
machine[1]              E                   1                   1                   1                   -
machine[2]              E                   1                   1                   1                   -
machine[3]              E                   1                   1                   1                   -
machine[4]              E                   1                   1                   1                   -
machine[5]              E                   1                   1                   1                   -
-------------------------------------------------------------------------------------------------------------

Solution nr.         3
Objective status     integer feasible solution
Objective value      171 (min!)

Variables
Name                 Type            Activity         Lower bound         Upper bound            Marginal
-------------------------------------------------------------------------------------------------------------
```

```
x[1,3]                  B                   1                   0                   1                   -
x[2,4]                  B                   1                   0                   1                   -
x[3,5]                  B                   1                   0                   1                   -
x[4,1]                  B                   1                   0                   1                   -
x[5,2]                  B                   1                   0                   1                   -
------------------------------------------------------------------------------------------------------------

Constraints
Name                    Type            Activity         Lower bound         Upper bound            Marginal
------------------------------------------------------------------------------------------------------------
location[1]               E                   1                   1                   1                   -
location[2]               E                   1                   1                   1                   -
location[3]               E                   1                   1                   1                   -
location[4]               E                   1                   1                   1                   -
location[5]               E                   1                   1                   1                   -
machine[1]                E                   1                   1                   1                   -
machine[2]                E                   1                   1                   1                   -
machine[3]                E                   1                   1                   1                   -
machine[4]                E                   1                   1                   1                   -
machine[5]                E                   1                   1                   1                   -
------------------------------------------------------------------------------------------------------------

Solution nr.          4
Objective status      integer feasible solution
Objective value       163 (min!)

Variables
Name                    Type            Activity         Lower bound         Upper bound            Marginal
------------------------------------------------------------------------------------------------------------
x[1,3]                  B                   1                   0                   1                   -
x[2,5]                  B                   1                   0                   1                   -
x[3,4]                  B                   1                   0                   1                   -
x[4,1]                  B                   1                   0                   1                   -
x[5,2]                  B                   1                   0                   1                   -
------------------------------------------------------------------------------------------------------------

Constraints
Name                    Type            Activity         Lower bound         Upper bound            Marginal
------------------------------------------------------------------------------------------------------------
location[1]               E                   1                   1                   1                   -
location[2]               E                   1                   1                   1                   -
location[3]               E                   1                   1                   1                   -
location[4]               E                   1                   1                   1                   -
location[5]               E                   1                   1                   1                   -
machine[1]                E                   1                   1                   1                   -
machine[2]                E                   1                   1                   1                   -
machine[3]                E                   1                   1                   1                   -
machine[4]                E                   1                   1                   1                   -
machine[5]                E                   1                   1                   1                   -
------------------------------------------------------------------------------------------------------------

Solution nr.          5
Objective status      integer feasible solution
Objective value       191 (min!)

Variables
Name                    Type            Activity         Lower bound         Upper bound            Marginal
------------------------------------------------------------------------------------------------------------
x[1,1]                  B                   1                   0                   1                   -
x[2,2]                  B                   1                   0                   1                   -
x[3,3]                  B                   1                   0                   1                   -
x[4,4]                  B                   1                   0                   1                   -
x[5,5]                  B                   1                   0                   1                   -
------------------------------------------------------------------------------------------------------------

Constraints
Name                    Type            Activity         Lower bound         Upper bound            Marginal
------------------------------------------------------------------------------------------------------------
location[1]               E                   1                   1                   1                   -
location[2]               E                   1                   1                   1                   -
location[3]               E                   1                   1                   1                   -
location[4]               E                   1                   1                   1                   -
location[5]               E                   1                   1                   1                   -
machine[1]                E                   1                   1                   1                   -
machine[2]                E                   1                   1                   1                   -
machine[3]                E                   1                   1                   1                   -
machine[4]                E                   1                   1                   1                   -
machine[5]                E                   1                   1                   1                   -
------------------------------------------------------------------------------------------------------------
```

2.4.1.13 Generic travelling salesman problem

The asymmetric travelling salesman problem is well known and often described. In the following CMPL model the `(x,y)` coordinates of the cities are defined by random numbers and the distances are calculated by the Euclidian distance of the `(x,y)` coordinates and disturbed by smaller random numbers to generated an asymmetric distance matrix. To reproduce the solution, a rand seed is set. The CMPL model `tsp.cmpl` can be formulated as follows:

```
%display nonZeros

par:
   seed:=srand(100);
   M:=10000;

   nrOfCities:=10;
   cities:=1..nrOfCities;

   {i in cities:
        xp[i]:=rand(100);
        yp[i]:=rand(100);
   }

   {i in cities, j in cities:
       {i==j:
          dist[i,j]:=M; |
       default:
          dist[i,j]:= sqrt( (xp[i]-xp[j])^2 + (yp[i]-yp[j])^2 );
          dist[j,i]:= dist[i,j]+rand(10)-rand(10);
       }
   }

var:
   x[cities,cities]: binary;
   u[cities]: real[1..];

obj:
   sum{i in cities, j in cities: dist[i,j]* x[i,j]} ->min;

con:
   in_edges_ {j in cities:  sum{i in cities: x[i,j]}=1; }
   out_edges_ {i in cities:  sum{j in cities: x[i,j]}=1; }

   {i in 2..nrOfCities, j in 2..nrOfCities, i<>j:
     subTourCon[i,j]: u[i] - u[j] + nrOfCities * x[i,j] <= nrOfCities-1;
   }
```

CMPL command:

```
cmpl  tsp.cmpl
```

Solution:

```
-----------------------------------------------------------------------------------------------------------
Problem              tsp.cmpl
Nr. of variables     109
Nr. of constraints   92
Objective name       line_1
Solver name          CBC
Display variables    nonzero variables (all)
Display constraints  nonzero constraints (all)
-----------------------------------------------------------------------------------------------------------
```

```
Objective status     optimal
Objective value      321.319 (min!)

Variables
Name                 Type            Activity         Lower bound         Upper bound            Marginal
-------------------------------------------------------------------------------------------------------------
u[2]                    C                   9                   1                 inf                   -
u[3]                    C                   3                   1                 inf                   -
u[4]                    C                   1                   1                 inf                   -
u[5]                    C                   6                   1                 inf                   -
u[6]                    C                   4                   1                 inf                   -
u[7]                    C                   8                   1                 inf                   -
u[8]                    C                   7                   1                 inf                   -
u[9]                    C                   5                   1                 inf                   -
u[10]                   C                   2                   1                 inf                   -
x[1,4]                  B                   1                   0                   1                   -
x[2,1]                  B                   1                   0                   1                   -
x[3,6]                  B                   1                   0                   1                   -
x[4,10]                 B                   1                   0                   1                   -
x[5,8]                  B                   1                   0                   1                   -
x[6,9]                  B                   1                   0                   1                   -
x[7,2]                  B                   1                   0                   1                   -
x[8,7]                  B                   1                   0                   1                   -
x[9,5]                  B                   1                   0                   1                   -
x[10,3]                 B                   1                   0                   1                   -
-------------------------------------------------------------------------------------------------------------

Constraints
Name                 Type            Activity         Lower bound         Upper bound            Marginal
-------------------------------------------------------------------------------------------------------------
in_edges_[1]            E                   1                   1                   1                   -
in_edges_[2]            E                   1                   1                   1                   -
in_edges_[3]            E                   1                   1                   1                   -
in_edges_[4]            E                   1                   1                   1                   -
in_edges_[5]            E                   1                   1                   1                   -
in_edges_[6]            E                   1                   1                   1                   -
in_edges_[7]            E                   1                   1                   1                   -
in_edges_[8]            E                   1                   1                   1                   -
in_edges_[9]            E                   1                   1                   1                   -
in_edges_[10]           E                   1                   1                   1                   -
out_edges_[1]           E                   1                   1                   1                   -
out_edges_[2]           E                   1                   1                   1                   -
out_edges_[3]           E                   1                   1                   1                   -
out_edges_[4]           E                   1                   1                   1                   -
out_edges_[5]           E                   1                   1                   1                   -
out_edges_[6]           E                   1                   1                   1                   -
out_edges_[7]           E                   1                   1                   1                   -
out_edges_[8]           E                   1                   1                   1                   -
out_edges_[9]           E                   1                   1                   1                   -
out_edges_[10]          E                   1                   1                   1                   -
subTourCon[2,3]         L                   6                -inf                   9                   -
subTourCon[2,4]         L                   8                -inf                   9                   -
subTourCon[2,5]         L                   3                -inf                   9                   -
subTourCon[2,6]         L                   5                -inf                   9                   -
subTourCon[2,7]         L                   1                -inf                   9                   -
subTourCon[2,8]         L                   2                -inf                   9                   -
subTourCon[2,9]         L                   4                -inf                   9                   -
subTourCon[2,10]        L                   7                -inf                   9                   -
subTourCon[3,2]         L                  -6                -inf                   9                   -
subTourCon[3,4]         L                   2                -inf                   9                   -
subTourCon[3,5]         L                  -3                -inf                   9                   -
subTourCon[3,6]         L                   9                -inf                   9                   -
subTourCon[3,7]         L                  -5                -inf                   9                   -
subTourCon[3,8]         L                  -4                -inf                   9                   -
subTourCon[3,9]         L                  -2                -inf                   9                   -
subTourCon[3,10]        L                   1                -inf                   9                   -
subTourCon[4,2]         L                  -8                -inf                   9                   -
subTourCon[4,3]         L                  -2                -inf                   9                   -
subTourCon[4,5]         L                  -5                -inf                   9                   -
subTourCon[4,6]         L                  -3                -inf                   9                   -
subTourCon[4,7]         L                  -7                -inf                   9                   -
subTourCon[4,8]         L                  -6                -inf                   9                   -
subTourCon[4,9]         L                  -4                -inf                   9                   -
subTourCon[4,10]        L                   9                -inf                   9                   -
subTourCon[5,2]         L                  -3                -inf                   9                   -
subTourCon[5,3]         L                   3                -inf                   9                   -
subTourCon[5,4]         L                   5                -inf                   9                   -
subTourCon[5,6]         L                   2                -inf                   9                   -
subTourCon[5,7]         L                  -2                -inf                   9                   -
subTourCon[5,8]         L                   9                -inf                   9                   -
subTourCon[5,9]         L                   1                -inf                   9                   -
subTourCon[5,10]        L                   4                -inf                   9                   -
subTourCon[6,2]         L                  -5                -inf                   9                   -
subTourCon[6,3]         L                   1                -inf                   9                   -
```

```
subTourCon[6,4]         L                3               -inf                9                -
subTourCon[6,5]         L               -2               -inf                9                -
subTourCon[6,7]         L               -4               -inf                9                -
subTourCon[6,8]         L               -3               -inf                9                -
subTourCon[6,9]         L                9               -inf                9                -
subTourCon[6,10]        L                2               -inf                9                -
subTourCon[7,2]         L                9               -inf                9                -
subTourCon[7,3]         L                5               -inf                9                -
subTourCon[7,4]         L                7               -inf                9                -
subTourCon[7,5]         L                2               -inf                9                -
subTourCon[7,6]         L                4               -inf                9                -
subTourCon[7,8]         L                1               -inf                9                -
subTourCon[7,9]         L                3               -inf                9                -
subTourCon[7,10]        L                6               -inf                9                -
subTourCon[8,2]         L               -2               -inf                9                -
subTourCon[8,3]         L                4               -inf                9                -
subTourCon[8,4]         L                6               -inf                9                -
subTourCon[8,5]         L                1               -inf                9                -
subTourCon[8,6]         L                3               -inf                9                -
subTourCon[8,7]         L                9               -inf                9                -
subTourCon[8,9]         L                2               -inf                9                -
subTourCon[8,10]        L                5               -inf                9                -
subTourCon[9,2]         L               -4               -inf                9                -
subTourCon[9,3]         L                2               -inf                9                -
subTourCon[9,4]         L                4               -inf                9                -
subTourCon[9,5]         L                9               -inf                9                -
subTourCon[9,6]         L                1               -inf                9                -
subTourCon[9,7]         L               -3               -inf                9                -
subTourCon[9,8]         L               -2               -inf                9                -
subTourCon[9,10]        L                3               -inf                9                -
subTourCon[10,2]        L               -7               -inf                9                -
subTourCon[10,3]        L                9               -inf                9                -
subTourCon[10,4]        L                1               -inf                9                -
subTourCon[10,5]        L               -4               -inf                9                -
subTourCon[10,6]        L               -2               -inf                9                -
subTourCon[10,7]        L               -6               -inf                9                -
subTourCon[10,8]        L               -5               -inf                9                -
subTourCon[10,9]        L               -3               -inf                9                -
------------------------------------------------------------------------------------------------------------
```

By analysing the position variables `u` and knowing that the start city is node 1, the following optimal tour is found:
1→4→10→3→6→9→5→8→7→2→1

2.4.2 Other selected examples

This section illustrates how CMPL can be used as simple solver or heuristic.

2.4.2.1 Solving the knapsack problem

CMPL can be used as a heuristic solver for knapsack problems.

The idea of the following models is to evaluate each item using the relation between the value and weight per item. The knapsack will be filled with the items sorted in descending order until the capacity limit or the minimum value is reached. Using the data from the examples in section 2.4.1.5 a CMPL model to maximize the total sales relative to capacity can be formulated as follows.

Model 1: maximize the total sales `knapsack-max-heuristic.cmpl`

```
%include knapsack-data.cmpl

#calculating the relative value of each box
{j in boxes: val[j]:= p[j]/w[j]; }

sumSales:=0;
sumWeight:=0;
```

```
#initial solution
x[]:=(0,0,0,0,0,0,0,0,0,0);

{ i in boxes:
  maxVal:=max(val[]);
  {j in boxes:
    { maxVal=val[j] :
      { sumWeight+w[j] <= maxWeight:
        x[j]:=1;
        sumSales:=sumSales + p[j];
        sumWeight:=sumWeight + w[j];
      }
      val[j]:=0;
      break j;
    }
  }
}

echo("Solution found");
echo("Optimal total sales: ", sumSales);
echo("Total weight : " ,sumWeight);
{j in boxes: echo( "x_"+ j + ": " + x[j]); }
```

Solution:

```
Solution found
Optimal total sales:  690
Total weight :  57
x_1: 1
x_2: 1
x_3: 0
x_4: 1
x_5: 0
x_6: 1
x_7: 1
x_8: 0
x_9: 1
x_10: 1
```

This solution is of course worse than the optimal solution in section 2.4.1.5 , but it is a feasible solution.

Model 2: minimize the total weight `knapsack-min-heuristic.cmpl`

```
%include knapsack-data.cmpl

#calculating the relative value of each box
{j in boxes: val[j]:= w[j]/p[j]; }

M:=10000;
sumSales:=0;
sumWeight:=0;
#initial solution
x[]:=(0,0,0,0,0,0,0,0,0,0);
{sumSales < minSales:
  maxVal:=min(val[]);
  {j in boxes:
    { maxVal=val[j] :
      { sumSales < minSales:
        x[j]:=1;
        sumSales:=sumSales + p[j];
```

```
          sumWeight:=sumWeight + w[j];
       }

       val[j]:=M;
       break j;
     }
   }
   repeat;
}
echo( "Solution found");
echo( "Optimal total weight : " + sumWeight);
echo( "Total sales: "+ sumSales);
{j in boxes: echo( "x_"+ j + ": " + x[j]); }
```

Solution:

```
Solution found
Optimal total weight : 47
Total sales: 630
x_1: 1
x_2: 1
x_3: 0
x_4: 1
x_5: 0
x_6: 0
x_7: 0
x_8: 0
x_9: 1
x_10: 1
```

This solution is identical to the optimal solution in section 2.4.1.5 .

2.4.2.2 Finding the maximum of a concave function using the bisection method

One of the alternative methods for finding the maximum of a negative convex function is the bisection method. (Hillier and Liebermann 2010, p. 554f.) A CMPL programme to find the maximum of $f(x)=12x-3x^4-2x^6$ can be formulated as follows (bisection.cmpl):

```
#distance epsilon
e:=0.00001;
#initial solution
xl:= 0;
xo:= 2;
xn:= (xl+xo)/2;

{ (xo-xl) > e :
  fd:= 12 - 12 * xn^3 - 12 * xn^5;
  { fd >= 0 : xl:=xn; |
    fd <= 0 : xo:=xn ;}
  xn:= (xl+xo)/2;

  fx := 12 * xn -3 * xn^4 - 2 * xn^6;

  echo( "f'(xn): " + fd + " xl: " + xl +
       " xo: " + xo + " xn: " + xn +
       " f(xn): " + fx);
```

```
  repeat;
}

echo("Solution found");
echo( "x: "+ xn);
echo( "function value: " + (12 * xn -3 * xn^4 - 2 * xn^6));
```

Solution:

```
f'(xn): -12 xl: 0 xo: 1 xn: 0.5 f(xn): 5.78125
f'(xn): 10.125 xl: 0.5 xo: 1 xn: 0.75 f(xn): 7.69482
f'(xn): 4.08984 xl: 0.75 xo: 1 xn: 0.875 f(xn): 7.84386
f'(xn): -2.19397 xl: 0.75 xo: 0.875 xn: 0.8125 f(xn): 7.86718
f'(xn): 1.31437 xl: 0.8125 xo: 0.875 xn: 0.84375 f(xn): 7.88291
f'(xn): -0.339699 xl: 0.8125 xo: 0.84375 xn: 0.828125 f(xn): 7.8815
f'(xn): 0.511253 xl: 0.828125 xo: 0.84375 xn: 0.835938 f(xn): 7.88387
f'(xn): 0.0918924 xl: 0.835938 xo: 0.84375 xn: 0.839844 f(xn): 7.88381
f'(xn): -0.122357 xl: 0.835938 xo: 0.839844 xn: 0.837891 f(xn): 7.88394
f'(xn): -0.0148481 xl: 0.835938 xo: 0.837891 xn: 0.836914 f(xn): 7.88393
f'(xn): 0.038618 xl: 0.836914 xo: 0.837891 xn: 0.837402 f(xn): 7.88394
f'(xn): 0.0119089 xl: 0.837402 xo: 0.837891 xn: 0.837646 f(xn): 7.88395
f'(xn): -0.00146357 xl: 0.837402 xo: 0.837646 xn: 0.837524 f(xn): 7.88395
f'(xn): 0.00522419 xl: 0.837524 xo: 0.837646 xn: 0.837585 f(xn): 7.88395
f'(xn): 0.00188068 xl: 0.837585 xo: 0.837646 xn: 0.837616 f(xn): 7.88395
f'(xn): 0.000208652 xl: 0.837616 xo: 0.837646 xn: 0.837631 f(xn): 7.88395
f'(xn): -0.000627434 xl: 0.837616 xo: 0.837631 xn: 0.837624 f(xn): 7.88395
f'(xn): -0.000209385 xl: 0.837616 xo: 0.837624 xn: 0.83762 f(xn): 7.88395
Solution found
x: 0.83762
function value: 7.88395
```

3 CMPL software package

3.1 CMPL software package in a glance

CMPL (<Coliop|Coin> Mathematical Programming Language) is a mathematical programming language and a system for mathematical programming and optimisation of linear optimisation problems.

CMPL executes CBC, GLPK, Gurobi, SCIP or CPLEX directly to solve the generated model instance. Because it is also possible to transform the mathematical problem into MPS or Free-MPS, alternative solvers can be used.

The CMPL distribution contains **Coliop** which is an IDE (Integrated Development Environment) for CMPL and also pyCMPL, jCMPL and CMPLServer.

pyCMPL is the CMPL application programming interface (API) for Python and **jCMPL** is CMPL's Java API. The main idea of this APIs is to define sets and parameters within the user application, to start and control the solving process and to read the solution(s) into the application if the problem is feasible. All variables, objective functions and constraints are defined in CMPL. These functionalities can be used with a local CMPL installation or a CMPLServer.

CMPLServer is an XML-RPC-based web service for distributed and grid optimisation that can be used with CMPL, pyCMPL and jCMPL. It is reasonable to solve large models remotely on the CMPLServer that is installed on a high performance system. CMPL provides specific XML-based file formats for the communication between a CMPLServer and its clients.

3.2 Download and installation

CMPL binaries for Windows, Linux and macOS are available at http://www.coliop.org.

Linux and Windows:

An installation is not required after unpacking the ZIP or tar.gz file. The CMPL package works out of the box in any folder.

macOS:

To use CMPL on macOS the following installation steps are necessary:

1) Download CMPL from http://www.coliop.org

2) Unzip CMPL package and copy (or move) the Cmpl2 folder to /Applications

3) Open Terminal

   The easiest way to open `Terminal` is to press `Cmd+Space` to open Spotlight Search. Afterward type type `Terminal` in the Spotlight input field. Simply select the Terminal entry in the search result list to open Terminal.

4) Run Cmpl setup script in Terminal (just copy and paste the following command and press enter)

```
/Applications/Cmpl2/cmpl_setup
```

To start CmplShell, the link `cmplShell` in `/Applications/Cmpl2` have to be double-clicked. In addition a user can do so in Coliop (Menu Actions -> Open CmplShell). To start `cmpl` on the command line please use it inside CmplShell. If `cmpl.opt` or `cmplServer.opt` need to be edited, just open them via the links in the `/Applications/Cmpl2/opt` subfolder.

3.3 CMPL

3.3.1 Running CMPL

It is recommanded to start `cmpl` inside `CmplShell`. On Windows and Linux, a user can also run CMPL by starting the `cmpl` script in the CMPL folder (not in `CMPLHOME/bin`). A CMPL model can be solved with the command `cmpl <problemname>.cmpl`.

3.3.2 Usage of the CMPL command line tool

CMPL can be controlled by options, which can be specified as command line arguments or as options within a CMPL header.

Usually, the first option is CMPL file followed by CMPL, solver and/or display options.

```
cmpl <cmplFile> [<options>]
```

The elements of CMPL header correspond to the command line options that can be used in the call to CMPL. Exceptions are only those command line options that must already be evaluated before the CMPL file is read and therefore cannot be used in CMPL header.

Each line for CMPL header starts with % as the first non-whitespace character. This is followed by the name of the command line option (without the -, which introduces a command line option in the command line). This is followed by the arguments of the command line option, separated by whitespace.

Alternatively, the line can begin with %arg. In this case, command line options and their arguments can be specified as on the command line itself (i.e. with - in front of the name of the command line option). Several command line options can then also be on one line.

Important options are described below.

Input options:

Command line:

`-i <cmplFile>`	Input file (the file can also specified it as the first option without -i)
`-include <file>`	Reads a Cmpl file additionally to the main Cmpl file
`-data <file>[: elements]`	Reads a CmplData file
`-xlsData <file>[: elements]`	Reads a CmplXlsData file

Cmpl header:

`%include <file>`	Reads a Cmpl file additionally to the main Cmpl file
`%data <file>[: elements]`	Reads a CmplData file
`%xlsData <file>[: elements]`	Reads a CmplXlsData file

Output options:

Command line:

`-m [<File>]`	Exports model in MPS format
`-fm [<File>]`	Exports model in Free-MPS format in a file or stdout
`-matrix [<File>]`	Exports the model as matrix in a file
`-p [<File>]`	Writes protocol messages into <file>
`-cmsg [<File>]`	Writes CMPL messages into <file>
`-solution [<File>]`	Writes the solution in CmplSolution XML format in a file

`-solutionAscii [<File>]`	Writes the solution in ASCII format in a file
`-solutionCsv [<File>]`	Writes the solution in CSV format in a file

Cmpl header:

`%m [<File>]`	Exports model in MPS format
`%fm [<File>]`	Exports model in Free-MPS format in a file or stdout
`%matrix [<File>]`	Exports the model as matrix in a file
`%p [<File>]`	Writes protocol messages into <file>
`%cmsg [<File>]`	Writes CMPL messages into <file>
`%solution [<File>]`	Writes the solution in CmplSolution XML format in a file
`%solutionAscii [<File>]`	Writes the solution in ASCII format in a file
`%solutionCsv [<File>]`	Writes the solution in CSV format in a file

Display options:

Command line:

`-display nonZeros`	Only activities with an value unequal to zero are shown in the solution.
`-display ignoreVars`	Variables are not shown in the solution.
`-display ingnoreCons`	Constraints are not shown in the solution.
`-display generatedElements`	Columns and rows generated by CMPL are shown.
`-display <var|con> <varOrConName=pattern>`	Only variables and/or constraints with a name matching the pattern are shown.
`-display solutionPool`	Shows multiple solutions (only Cplex or Gurobi)

Cmpl header:

`%display nonZeros`	Only activities with an value unequal to zero are shown in the solution.
`%display ignoreVars`	Variables are not shown in the solution.
`%display ingnoreCons`	Constraints are not shown in the solution.
`%display generatedElements`	Columns and rows generated by CMPL are shown.
`%display <var|con> <varOrConName=pattern>`	Only variables and/or constraints with a name matching the pattern are shown.
`%display solutionPool`	Shows multiple solutions (only Cplex or Gurobi)

Solver and solver options:

Command line:

`-solver <cbc|glpk|scip|cplex|gurobi>`	Specifies the solver to be invoked.
`-opt <cbc|glpk|scip|cplex|gurobi> <option>[=<val>]`	Specifies options for the solver.

Cmpl header:

`%solver <cbc|glpk|scip|cplex|gurobi>`	Specifies the solver to be invoked.
`%opt <cbc|glpk|scip|cplex|gurobi> <option>[=<val>]`	Specifies options for the solver.

CmplServer options:

Command line:

`-url <url>`	Url of a CmplServer - Without other arguments, the problem are solved on the CmplServer (synchronous mode)
`-send`	Sends a problem to a CmplServer which have to be specified with -url (asynchronous mode)
`-knock`	Obtains the status of a problem on the CmplServer and fetches the stdout and displays it (asynchronous mode).
`-retrieve`	Retrieves the results of the problem from the CmplServer (asynchronous mode)
`-cancel`	Cancels the problem at the CmplServer (asynchronous mode)
`-maxTries <x>`	Maximum number of tries of failed CmplServer calls
`-maxTime <x>`	Maximum time in <x> seconds that a problem waits in a CmplServer queue.

Cmpl header:

`%url <url>`	Url of a CmplServer - Without other arguments, the problem are solved on the CmplServer (synchronous mode)
`%send`	Sends a problem to a CmplServer which have to be specified with -url (asynchronous mode)
`%knock`	Obtains the status of a problem on the CmplServer and fetches the stdout and displays it (asynchronous mode).
`%retrieve`	Retrieves the results of the problem from the CmplServer (asynchronous mode)
`%cancel`	Cancels the problem at the CmplServer (asynchronous mode)
`%maxTries <x>`	Maximum number of tries of failed CmplServer calls
`%maxTime <x>`	Maximum time in <x> seconds that a problem waits in a CmplServer queue.

Other options:

Command line:

`-silent`	Suppresses CMPL and solver messages
`-int-relax`	Integer or binary variables are used as continues variables.
`-threads <n>`	Use maximal n running threads (0: no threading)
`-ordered`	Ordered execution in all explicit and implicit iterations
`-check-only`	Only syntax check
`-syntax-xml[<file>]`	Writes syntax structure of the Cmpl input as xml to <file>
`-help`	Prints all options to stdOut

Cmpl header:

`%silent`	Suppresses CMPL and solver messages
`%int-relax`	Integer or binary variables are used as continues variables.
`%threads <n>`	Use maximal n concurrently running threads (0: no threading)
`%ordered`	Ordered execution in all explicit and implicit iterations
`%check-only`	Only syntax check
`%syntax-xml[<file>]`	Writes syntax structure of the Cmpl input as xml to <file>
`%help`	Prints all options to stdOut

Examples:

`cmpl test.cmpl`	Solves the problem `test.cmpl` locally with the default solver and displays a standard solution report
`cmpl test.cmpl -solver glpk`	Solves the problem `test.cmpl` locally using GLPK and displays a standard solution report
`cmpl test.cmpl ↲` `  -url http://194.95.44.187:8008`	Solves the problem `test.cmpl` remotely with the defined CMPLServer and displays a standard solution report
`cmpl test.cmpl -solutionCsv`	Solves the problem `test.cmpl` locally with the default solver writes the solution in the CSV-file `test.csv` and displays a standard solution report
`cmpl "/Users/test/Documents/ ↲` `  Projects/Project 1/test.cmpl"`	If the file name or the path contains blanks then one can enclose the entire file name in double quotes.
`cmpl test.cmpl -m test.mps`	Reads the file `test.cmpl` and generates the MPS-file `test.mps`.
`cmpl test.cmpl -fm test.mps`	Reads the file `test.cmpl` and generates the Free-MPS-file `test.mps`.

3.3.3 Using CMPL with several solvers

There are two ways to interact with several solvers. It is recommended to use one of the solvers which are directly supported and executed by CMPL. The CMPL package contains CBC as the default solver. If you have installed Gurobi, CPLEX, SCIP, GLPK then you can also use these solvers directly. To invoke CPLEX, SCIP or GLPK, the file `cmpl.opt` in `CMPLHOME/bin` must be edited, specifying the full filename of the binary after the keyword for the solver.

Example for cmpl.opt:

```
CBC ../Thirdparty/CBC/cbc
GLPK    /Users/stegger/opt/GLPK/glpsol
SCIP    /Users/stegger/opt/SCIPOptSuite-7.0.2-Darwin/bin/scip
CPLEX   /Applications/CPLEX_Studio1210/cplex/bin/x86-64_osx/cplex
GUROBI gurobiCmpl
```

Because CMPL transforms a CMPL model into an MPS or a Free-MPS, the generated model instance can be solved by using most of the free or commercial solvers.

3.3.3.1 CBC

Cbc (Coin-or branch and cut) is an open-source mixed integer programming solver written in C++. It can be used as a callable library or stand-alone solver. The CMPL distribution contains the CBC binary. For more information please visit https://projects.coin-or.org/Cbc.

Since CBC is the default solver CBC doesn't need not to be specified:

```
cmpl <problem>.cmpl     #Solves the problem locally with CBC
```

It is possible to use most of the CBC solver options within the CMPL header. Please see Appendix 6.1 for a list of useful CBC parameters.

Usage of CBC parameters within the CMPL header:

```
%opt cbc <option>[=<val>]
```

3.3.3.2 GLPK

The GLPK (GNU Linear Programming Kit) package is intended for solving large-scale linear programming (LP), mixed integer programming (MIP), and other related problems. "The GLPK package includes the program glpsol, which is a stand-alone LP/MIP solver. This program can be invoked from the command line ... to read LP/MIP problem data in any format supported by GLPK, solve the problem, and write the problem solution obtained to an output text file." (GLPK 2014, p. 166.). For more information please visit the GLPK project website: http://www.gnu.org/software/glpk.

If GLPK is installed on the same computer as CMPL then GLPK can be connected to CMPL by changing the entry `GLPK` in the file `<CMPLHOME>/bin/cmpl.opt`.

The CMPL package contains GLPK and it can be used by the following command:

```
cmpl <problem>.cmpl -solver glpk
```

or by the CMPL header flag:

```
%solver glpk
```

Most of the GLPK solver options can be used by defining solver options within the CMPL header. Please see Appendix 6.2 for a list of useful GLPK parameters.

Usage of GLPK parameters within the CMPL header:

```
%opt glpk <option>[=<val>]
```

3.3.3.3 Gurobi

"The fastest and most powerful mathematical programming solver available for your LP, QP and MIP (MILP, MIQP, and MIQCP) problems. See why so many companies are choosing Gurobi for better performance, faster development and better support." (https://www.gurobi.com/products/gurobi-optimizer/)

If Gurobi is installed on the same computer as CMPL then Gurobi can be executed directly only by using the command:

```
cmpl <problem>.cmpl -solver gurobi
```

or by the CMPL header flag:

```
%solver gurobi
```

All Gurobi parameters (excluding NodefileDir, LogFile and ResultFile) described in the Gurobi manual can be used in the CMPL header.

Usage of Gurobi parameters within the CMPL header:

```
%opt gurobi <option>[=<val>]
```

3.3.3.4 SCIP

SCIP is a project of the Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB). "SCIP is a framework for Constraint Integer Programming oriented towards the needs of Mathematical Programming experts who want to have total control of the solution process and access detailed information down to the guts of the solver. SCIP can also be used as a pure MIP solver or as a framework for branch-cut-and-price. SCIP is implemented as C callable library and provides C++ wrapper classes for user plugins. It can also be used as a standalone program to solve mixed integer programs."
[http://scip.zib.de/whatis.shtml](Achterberg 2009)

SCIP can be used only for mixed integer programming (MIP) problems. If SCIP is chosen as solver and the problem is an LP then CBC is executed as solver.

If SCIP is installed on the same computer as CMPL then SCIP can be connected to CMPL by changing the entry `SCIP` in the file `<CMPLHOME>/bin/cmpl.opt`.
If this entry is correct then you can execute SCIP directly by using the command

```
cmpl <problem>.cmpl -solver scip
```

or by the CMPL header flag:

```
%solver scip
```

All SCIP parameters described in the SCIP Doxygen Documentation can be used in the CMPL header.
Please see: https://scipopt.org/doc/html/SHELL.php#TUTORIAL_PARAMETERS

Usage SCIP parameters within the CMPL header:

```
%opt scip <option>[=<val>]
```

3.3.3.5 CPLEX

CPLEX is a part of the IBM ILOG CPLEX optimisation Studio and includes simplex, barrier, and mixed integer optimizers. "IBM ILOG CPLEX optimisation Studio provides the fastest way to build efficient optimisation models and state-of-the-art applications for the full range of planning and scheduling problems. With its integrated development environment, descriptive modelling language and built-in tools, it supports the entire model development process." (IBM ILOG CPLEX optimisation Studio manual)

If CPLEX is installed on the same computer as CMPL then CPLEX can be connected to CMPL by changing the entry `CPLEX` in the file `<CMPLHOME>/bin/cmpl.opt`.
If this entry is correct then you can execute CPLEX directly by using the command

```
cmpl <problem>.cmpl -solver cplex
```

or by the CMPL header flag:

```
%solver cplex
```

All CPLEX parameters described in the CPLEX manual (Parameters of CPLEX → Parameters Reference Manual) can be used in the CMPL header.

Usage CPLEX parameters within the CMPL header:

```
%opt cplex <option>[=<val>]
```

You have to use the parameters for the Interactive Optimizer. The names of sub-parameters of hierarchical parameters are to be separated by slashes.

3.3.3.6 Other solvers

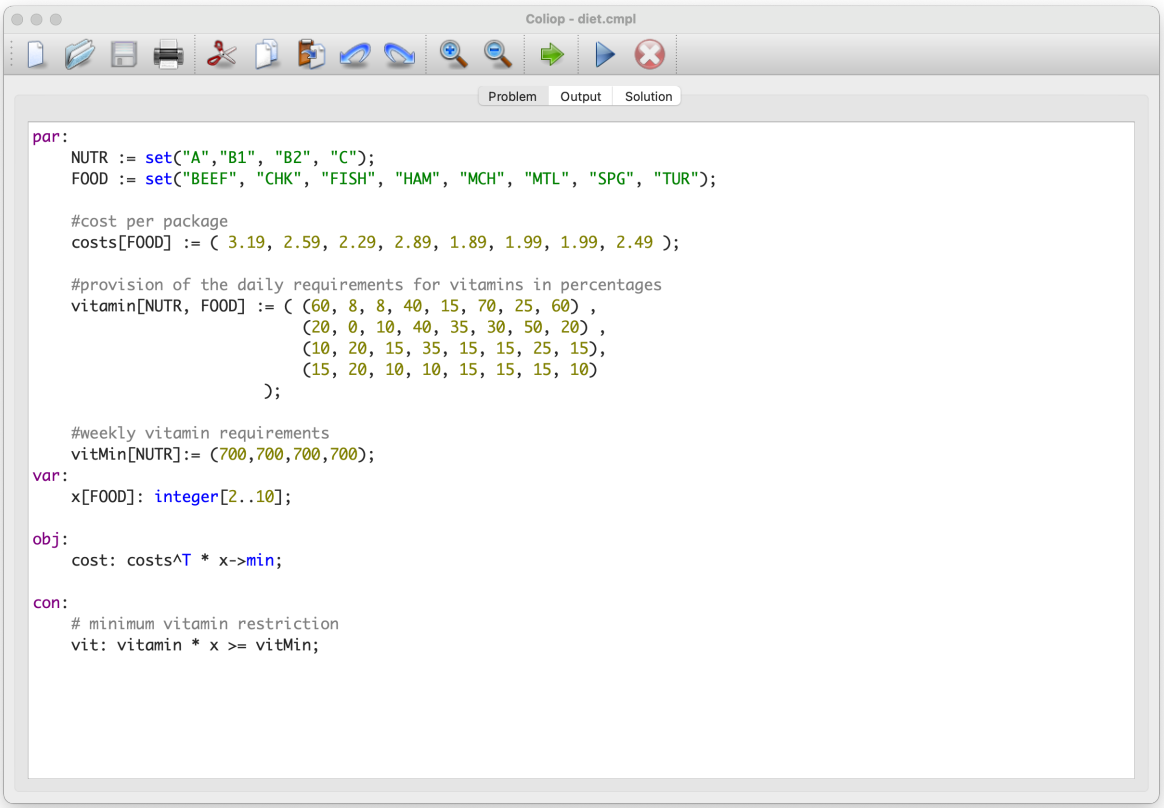
Since CMPL transforms a CMPL model into an MPS, a Free-MPS or an OSiL file, the model can be solved using most free or commercial solvers. To create MPS, Free-MPS or OSiL files please use the following commands:

```
cmpl <problemname>.cmpl <-m|-fm> <problemname>.mps      #MPS export
```

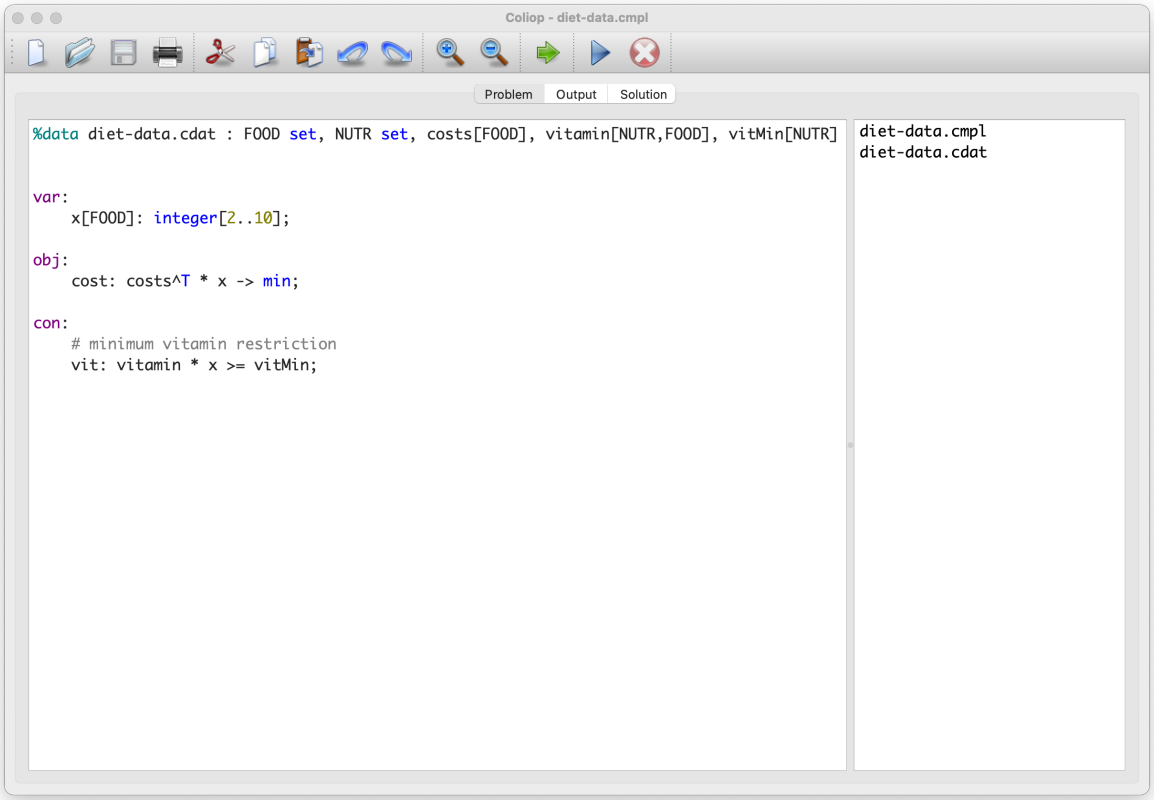
3.4 Coliop

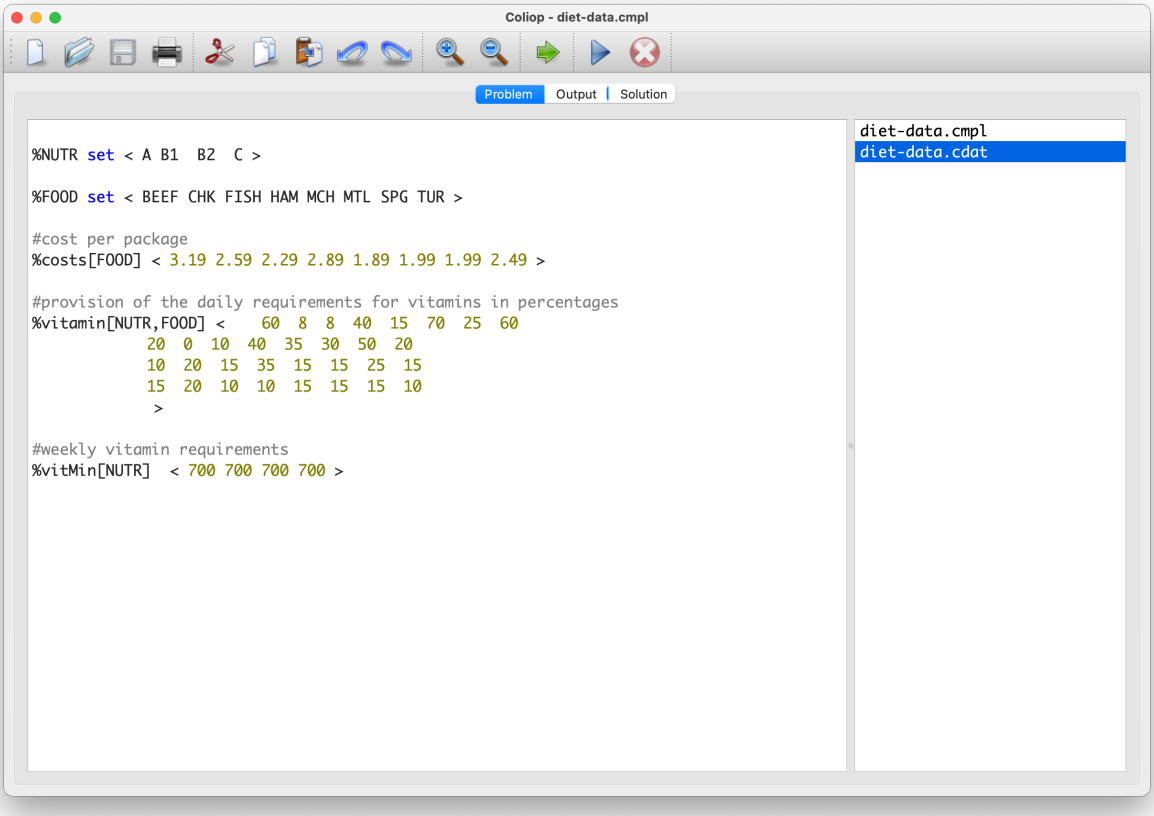
Coliop is an IDE (Integrated Development Environment) for CMPL . Coliop is an open-source project licensed under GPL. It is written in C++ and is as an integral part of the CMPL distribution available for most of the relevant operating systems (OS X, Linux and Windows). Coliop can be started by clicking the Coliop symbol in the CMPL folder (not in `CMPLHOME/bin`).

The first working step is to create or to open a CMPL model.

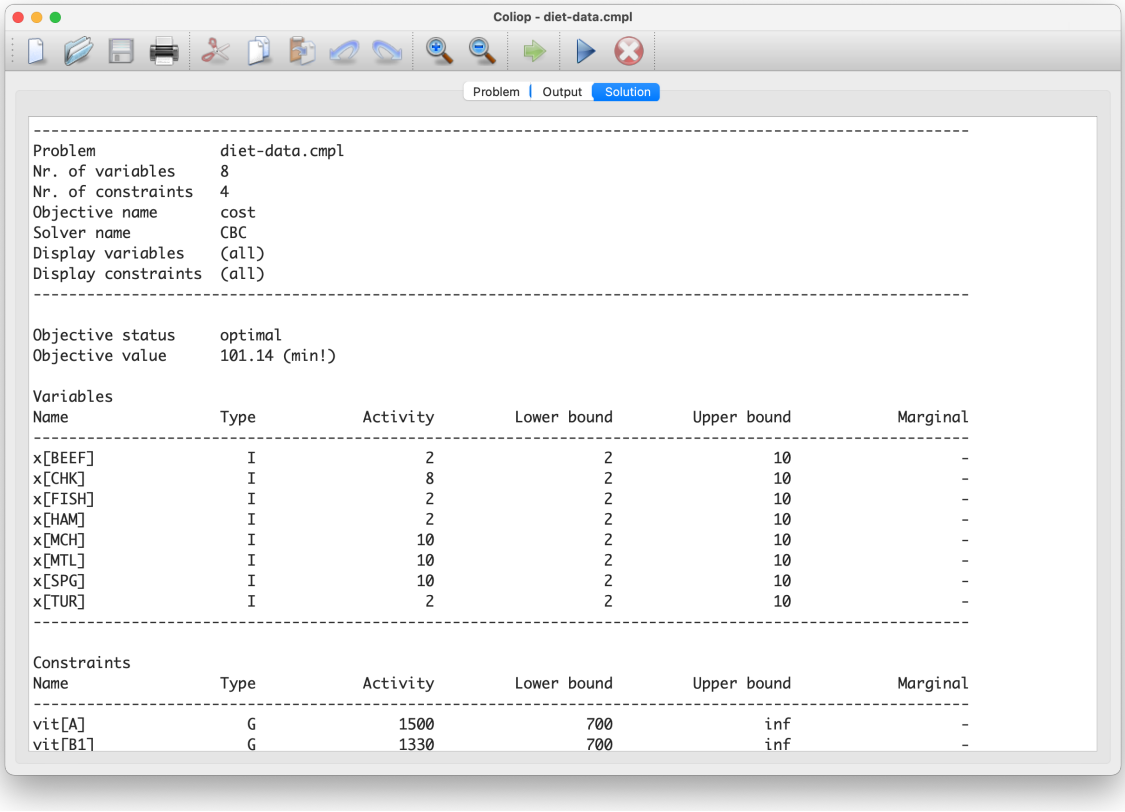


If the CMPL model imports an CmplData file by using the Cmpl header entry `%data` or the import of another CMPL file by using `%include` then a list of the involved files are shown right of the CMPL model. A user can switch between the files by clicking on the file names in this list. If a file does not exists then CMPL suggests to create the file.

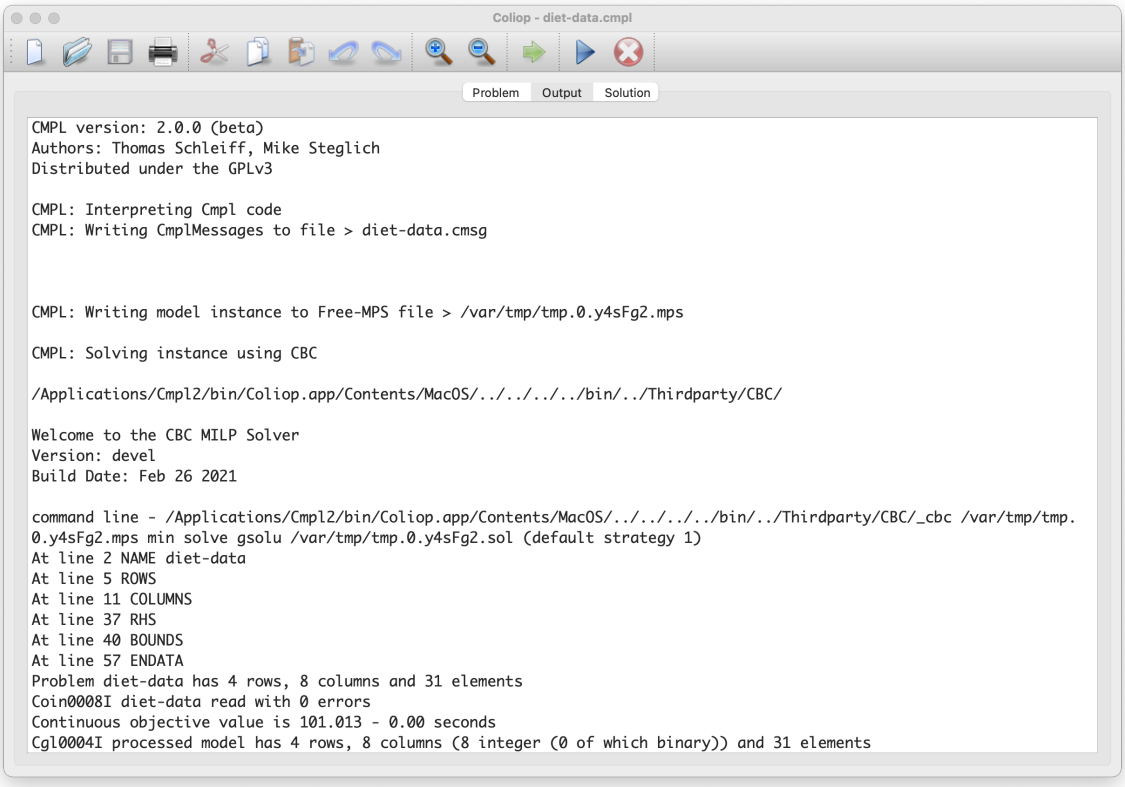

Coliop - diet-data.cmpl
Problem
Output
Solution
%data diet-data.cdat : FOOD set, NUTR set, costs[FOOD], vitamin[NUTR,FOOD], vitMin[NUTR]
var:
x[FOOD]: integer[2..10];
obj:
cost: costs^T * x -> min;
con:
minimum vitamin restriction
vit: vitamin * x >= vitMin;
diet-data.cmpl
diet-data.cdat


Coliop - diet-data.cmpl
Problem
Output
Solution
%NUTR set < A B1 B2 C >
%FOOD set < BEEF CHK FISH HAM MCH MTL SPG TUR >
#cost per package
%costs[FOOD] < 3.19 2.59 2.29 2.89 1.89 1.99 1.99 2.49 >
#provision of the daily requirements for vitamins in percentages
%vitamin[NUTR,FOOD] < 60 8 8 40 15 70 25 60
20 0 10 40 35 30 50 20
10 20 15 35 15 15 25 15
15 20 10 10 15 15 15 10
>
#weekly vitamin requirements
%vitMin[NUTR] < 700 700 700 700 >
diet-data.cmpl
diet-data.cdat

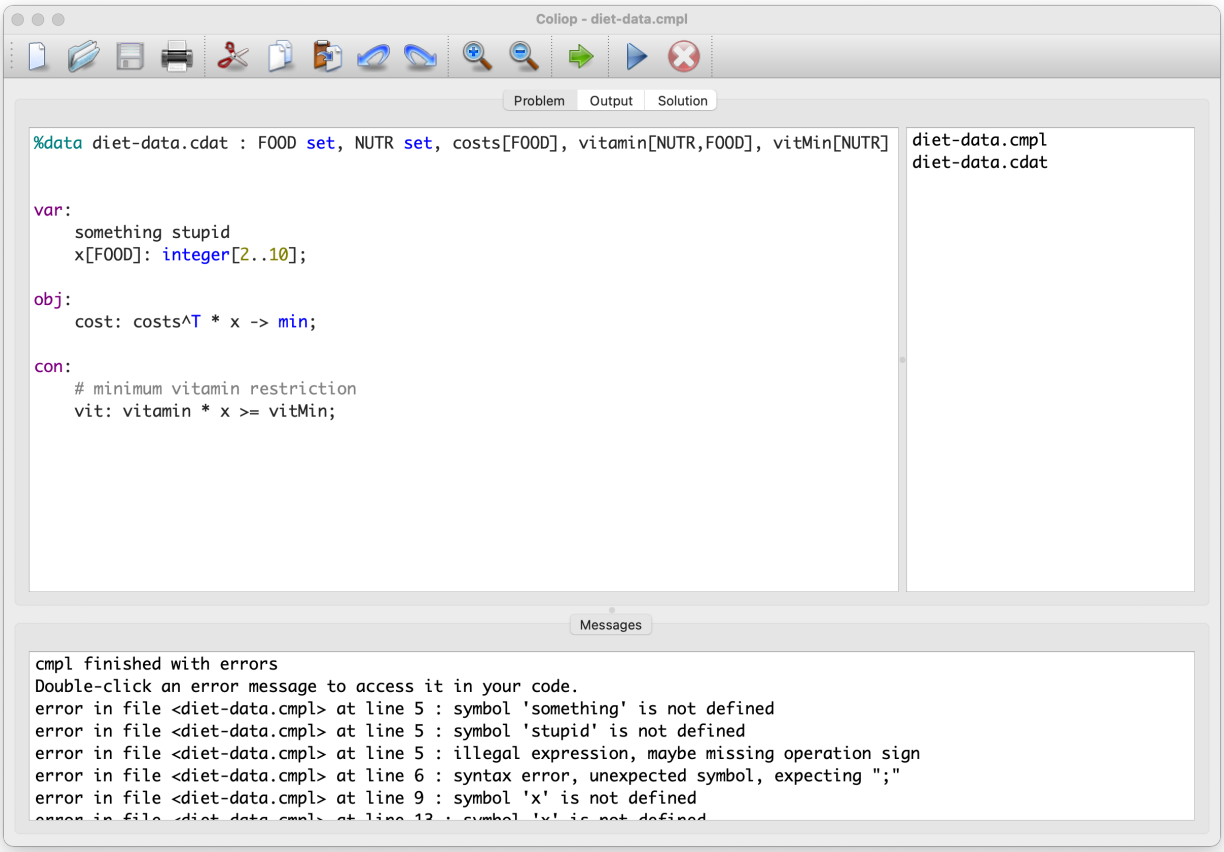
The model can be solved by clicking the button <Solve> in the toolbar or by choosing the menu entry <Action→Solve>. If the model is feasible and a solution is found the solution appears in the tab <Solution>.



It is possible to obtain the output of the invoked solver and CMPL's output in the tab <Output>.



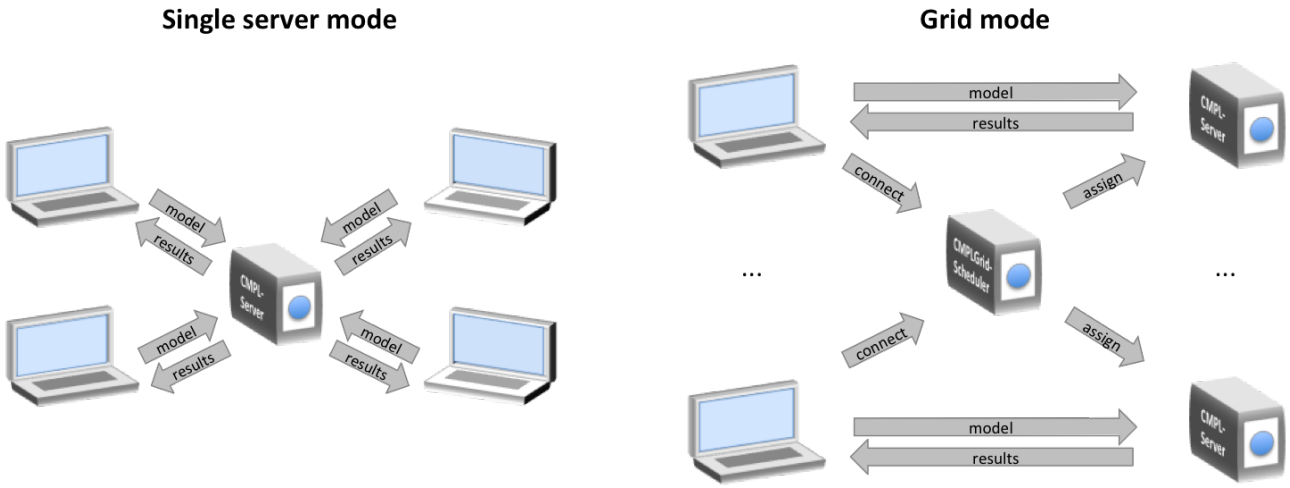
If a syntax error occurs then a user can analyse it by clicking on the error message in the CMPL message list below the CMPL model. The position in the CMPL model that occurs the error is shown automatically.



3.5 CMPLServer

The CMPLServer is an XML-RPC-based web service for distributed and grid optimisation. XML-RPC provides XML based procedures for Remote Procedure Calls (RPC), which are transmitted between a client and a server via HTTP. (St. Laurent et al. 2001, p. 1.) XML-RPC has been chosen since this it is less resource consuming than other protocols like SOAP or REST due to its simpler functionalities.

A CMPLServer can be used in a single server mode or in a grid mode:



Both modes can be understood as distributed systems "in which hardware and software components located at networks computers communicate and coordinate their actions only by passing messages". (Coulouris et al, 2012, p. 17) Distributed optimisation is in this meaning interpretable as a distributed system that can be used for solving optimisation problems. (cf. Kshemkalyani & Singhal, 2008, p. 1; Fourer et.al., 2010)

CMPL provides four XML-based file formats for the communication between a CMPLServer and its clients in both modes (`CmplInstance`, `CmplSolutions`, `CmplMessages`). A `CmplInstance` file contains an optimisation problem formulated in CMPL, the corresponding sets and parameters in the `CmplData` file format as well all CMPL and solver options that belong to the CMPL model. If the model is feasible and a solution is found then a `CmplSolutions` file contains the solution(s) and the status of the invoked solver. If the model is not feasible then only the solver's status and the solver messages are given in the solution file. The `CmplMessages` file is intended to provide the CMPL status and (if existing) the CMPL messages.

In the single server mode only one CMPLServer that can be accessed synchronously or asynchronously by the clients exists in the network. A model can be solved synchronously by executing the CMPL binary with the command line argument `-url <url>` or by running a pyCMPL or jCMPL programme by using the methods `Cmpl.connect(url)` for connecting the server and `Cmpl.solve()` for solving the model remotely.[1] The client sends the model to the CMPLServer and then waits for the results. If the model is feasible and an optimal solution is found the solution(s) can be received. If the model contains syntax or other errors or if the model is not feasible the CMPL and solver messages can be obtained. Whereby in the synchronous mode the client has to wait after sending the problem for the results and Messages in one process, a model can also be solved asynchronously with pyCMPL and jCMPL by using the methods `Cmpl.send()`, `Cmpl.knock()` and `Cmpl.retrieve()` in several steps. After sending the model to the CMPLServer via `Cmpl.send()` the server status can be obtained with `Cmpl.knock()`. If the CMPLServer is finished the solution, the CMPL and the solver states and messages can be received by `Cmpl.retrieve()`. It is reasonable to use the single server mode if a large model is formulated on a thin client in order to solve it remotely on the CMPLServer that is installed on a high performance system.

All these distributed optimisation procedures require a one-to-one connection between a CMPLServer and the client. The grid mode extends this approach by coupling CMPLServers from several locations and at least

1 Please take a look at the pyCMPL and jCMPL descriptions in chapter 4 .

one coordinating CMPLGridScheduler to one "virtual CMPLServer" as a grid computing system that can be defined "as a system that coordinates distributed resources using standard, open, general-purpose protocols and interfaces to deliver non-trivial qualities of service." (Forster & Kesselmann 2003, pos. 722) For the client there does not appear any difference whether there is a connection to a single CMPLServer or to a CMPLGrid. The client's model is to be connected with the same functionalities as for a single CMPLServer to a CMPLGridScheduler which is responsible for the load balancing within the CMPLGrid and the assignment of the model to one of the connected CMPLServers. After this step the client is automatically connected to the chosen CMPLServer and the model can be solved synchronously or asynchronously. A CMPLGrid should be used for handling a huge amount of large scale optimisation problems. An example can be a simulation in which each agent has to solve its own optimisation problem at several times. An additional example for such a CMPLGrid application is an optimisation web portal that provides a huge amount of optimisation problems.

Both modes can be controlled by the `cmplServer` script that can be started in the `CmplShell`.

cmplServer <*command*> [<*port*>] [**-showLog**]

command:

`-start`	starts as single CMPLServer
`-startInGrid`	starts CMPLServer and connects to CMPLGrid
`-startScheduler`	starts as CMPLGridScheduler
`-stop`	stops CMPLServer or CMPLGridScheduler
`-status`	returns the status of the CMPLServer or CMPLGridScheduler
port	defines CMPLServer's or CMPLGridScheduler's port
`-showLog`	shows the CMPLServer or CMPLGridScheduler log file

3.5.1 Single server mode

The first step to establish the single server mode is to start the CMPLServer by typing the command:

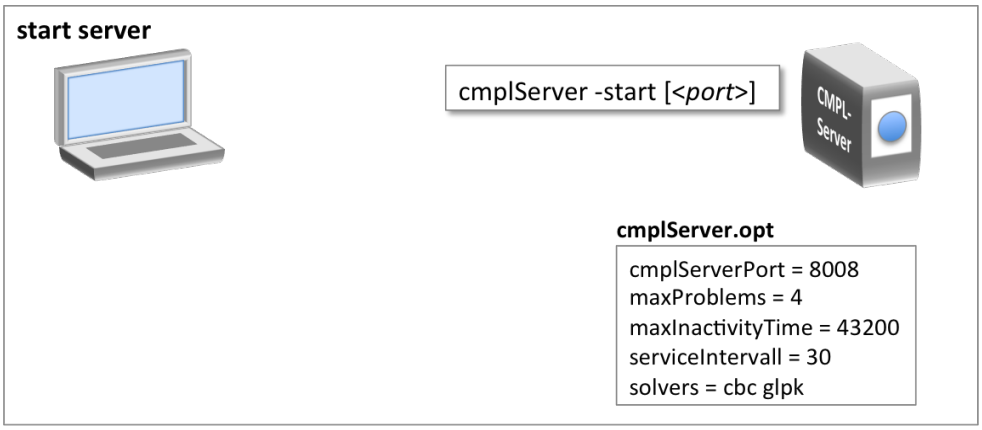
cmplServer -start [<*port*>]

Optionally a port can be specified as second argument. The behaviour of a CMPLServer can be influenced by editing the file `cmplServer.opt` that is located on Mac OS X in `/Applications/Cmpl/cmplServer`, on Linux in `/usr/share/Cmpl/cmplServer` and on Windows in `c:\program files[ (x86)]\Cmpl\cmplServer`. The example below shows the default values in this file.

```
cmplServerPort = 8008
maxProblems = 4
maxInactivityTime = 43200
serviceIntervall = 30
solvers = cbc glpk
```

The default port of the CMPLServer can be specified with the parameter `port`. The parameter `maxProblems` defines how many problems can be carried out simultaneously. If more problems than `maxProblems` are connected with the CMPLServer the supernumerary problems are assigned to the problem waiting queue and automatically started if a running problem is finished or cancelled. If a problem is longer inactive than defined by the parameter `maxInactivityTime` it is cancelled and deleted automatically by the CMPLServer. This procedure as well as the problem waiting queue handling are performed by a service thread

that works perpetual after a couple of seconds defined by the parameter `serviceIntervall`. With the parameter `solvers it` can be specified which solvers in the set of the installed solvers can be provided by the CMPLServer.



A running CMPLServer can be accessed by the CMPL binary or via CMPL's Python and Java APIs that contain CMPLServer clients. One can execute a CMPL model remotely on a CMPLServer by using the command line argument `-cmplUrl`.

```
cmpl <problem>.cmpl -url http://<ip-adress-or-Domain>:<port>
```

This command executes the problem on the CMPLServer synchronously. That means CMPL waits right after sending the problem for the results and messages in one process.

It is also possible to run a Cmpl Problem asynchronously on a CMPLServer. In a first step, the problem is sent to the server by coupling the `-cmplUrl` argument with the `-send` command line argument.

```
cmpl <problem>.cmpl -url http://<ip-adress-or-Domain>:<port> -send
```

Afterwards, the status of the problem can be obtained by using the command line argument `-knock`.

```
cmpl <problem>.cmpl -knock
```

The results can be retrieved by using the command line argument `-retrieve` after finishing the problem on the CMPLServer.

```
cmpl <problem>.cmpl -retrieve
```

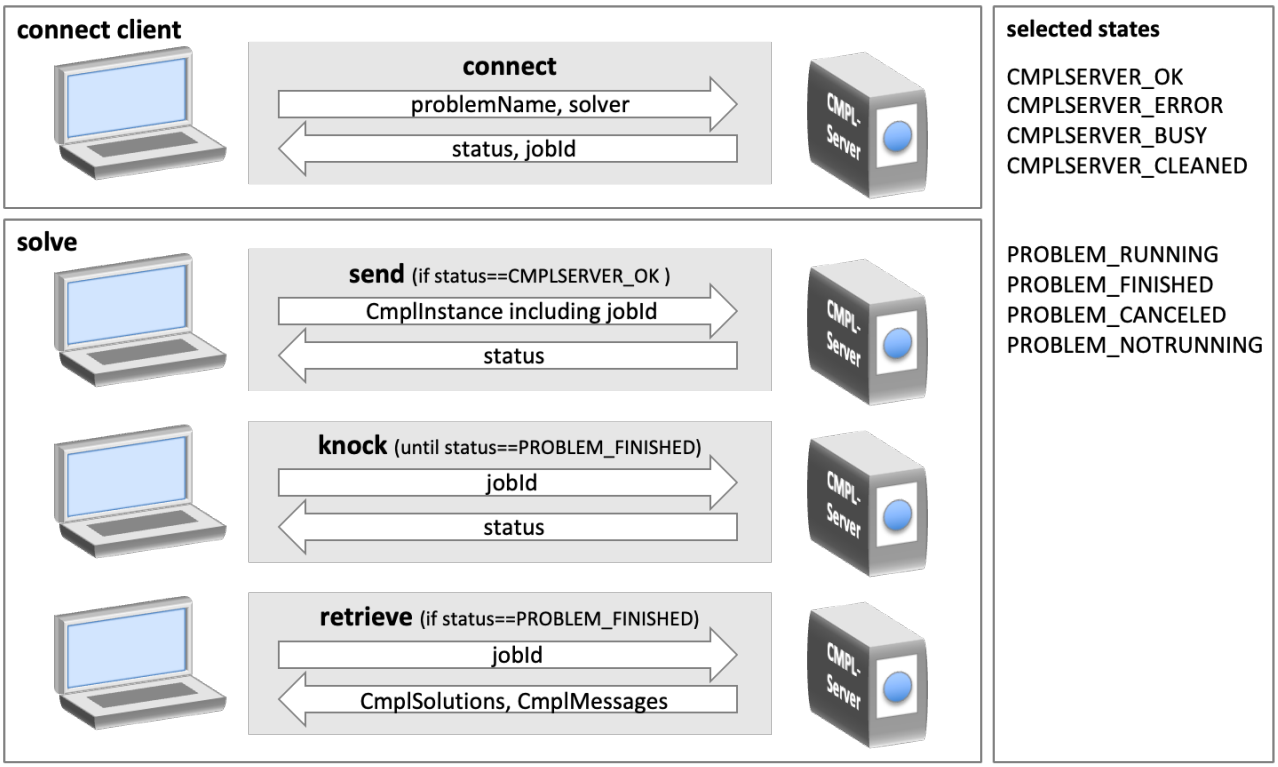
It is also possible to cancel the problem on the CmplServer if necessary by using the command line argument `-cancel`.

```
cmpl <problem>.cmpl -cancel
```

The status of a problem which is sent to a CMPLServer but not retrieved is saved automatically in a dump file in the temp folder. Therefore the computer could be switched off after sending the problem and later switched on to retrieve it.

In pyCMPL and jCMPL a CMPLServer can be connected by using the method `Cmpl.connect()`. Executing a model can be done synchronously by executing the method `Cmpl.solve()` or asynchronously by using the

methods `Cmpl.send()`, `Cmpl.knock()` and `Cmpl.retrieve()`. These main functionalities are illustrated in the following picture.



In the first step the client connects the CMPLServer, hands over its problem name and the solver with which the problem is to be solved. Then the client receives the status of the CMPLServer and if the status is `CMPLSERVER_OK` also the `jobId` is also sent. The status is `CMPLSERVER_ERROR` if the demanded solver is not supported or a CMPLServer occurs.

The synchronous method `Cmpl.solve()` is a bundle of the asynchronous methods `Cmpl.send()`, `Cmpl.knock()` and `Cmpl.retrieve()`.

`Cmpl.send()` sends a `CmplInstance` XML string that contains all relevant information about a CMPL model including the `jobId`, the CMPL and the solver options as well as the model itself and its data files to the CMPLServer. If the number of running problems including the model sent is greater than `maxProblems` the model is moved to the problem waiting queue and the CMPLServer returns the status `CMPLSERVER_BUSY`. If not the CMPLServer starts the solving process automatically if the CmplInstance string is completely received and the model and data files are written to the hard disc. In this case the status is set to `PROBLEM_RUNNING`.

A CMPLServer uses the home path of the user who is running it and saves all relevant data in `$HOME/CmplServer` (Mac and Linux) or `%HOMEPATH%\CmplServer` (Windows). The activities of the server can be obtained in the file `CmplServer.log`. Each problem is stored in an own folder specified by the `jobId` which is deleted automatically after disconnecting the problem.

In the next step the client asks the CMPLServer whether solving the problem is finished or not via `Cmpl.knock()` whereby the `jobId` identifies the problem and the CMPLServer returns the current status. The client has to knock until the status is `PROBLEM_RUNNING` (or `CMPLSERVER_ERROR`). If the status is `CM-`

`PLSERVER_BUSY` the problem is put into the problem waiting queue until an empty solving slot is available or the maximum queuing time (defined with the CMPL option `-maxQueuingTime` or by default 300 seconds) is reached. The procedure then stops automatically.

If the status is equal to `PROBLEM_RUNNING` the solution, the CMPL and the solver messages and if requested some statistics can be received by using `Cmpl.retrieve()`. The client sends its `jobId` and then retrieves the `CmplSolution`, `CmplMesages` and `CmplInfo` XML strings. If `Cmpl.knock()` returns `CMPLSERVER_ERROR` the process is stopped.

The CMPLServer can be stopped by typing the command:

```
cmplServer -stop [<port>]
```

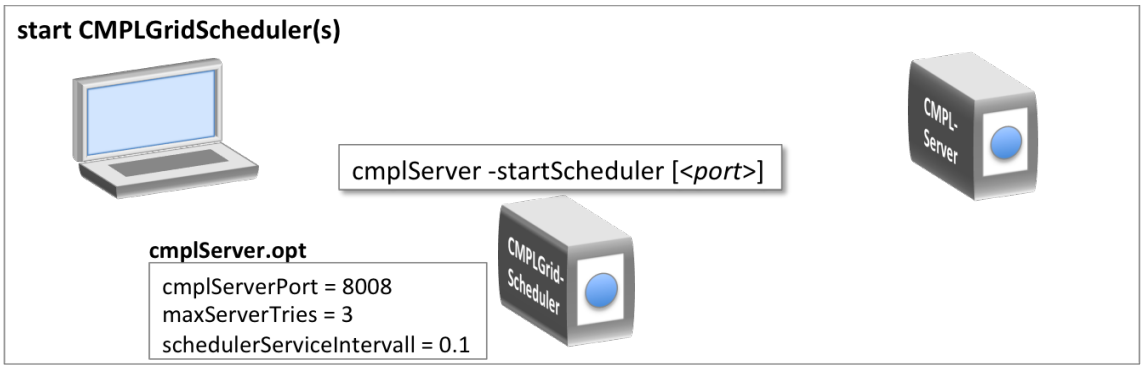
3.5.2 Grid mode

A CMPLGrid consists at least of one CMPLGridScheduler and usually a couple of CMPLServers that are connected to at least one scheduler. A CMPLGridScheduler is the gateway to the CMPLGrid for the clients and has to coordinate the traffic in the grid, that means it is responsible for the load balancing within the CMPLGrid and the assignment of the models to the connected CMPLServers. After receiving a model from a CMPLGridScheduler a CMPLServer has to communicate directly with the client to receive the model, to solve it and to send (if the problem is feasible) the solution(s), the CMPL and solver messages and if requested some information to the client. After these steps the client is disconnected automatically and the CMPLServers is waiting for the next problem from a CMPLGridScheduler.

The first step to start a CMPLGrid is to execute one or more CMPLGridScheduler by typing the command:

```
cmplServer -startScheduler [<port>]
```

As for the CMPLServers the parameter of a CMPLGridScheduler can be edited in the file `cmplServer.opt`.



The relevant parameters in `cmplServer.opt` for a CMPLGridScheduler with there default values are shown below.

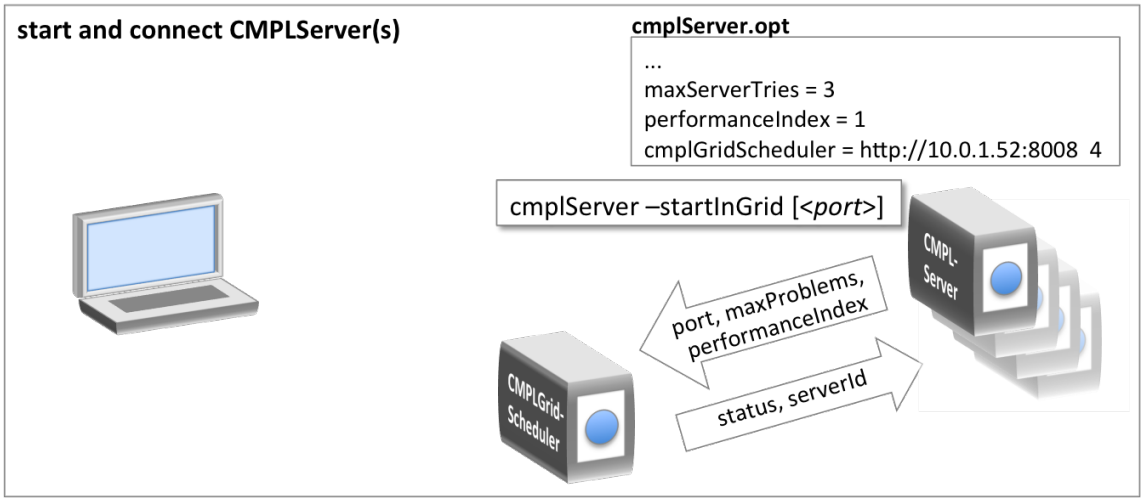
```
cmplServerPort = 8008
maxServerTries = 3
schedulerServiceIntervall = 0.1
```

The default port of the CMPLGridScheduler can be specified by the parameter `port`. If one wants to run a CMPLServer on the same computer as the CMPLGridScheduler then the server needs to be started with a dif-

ferent port via command line argument. Since the CMPLGridScheduler has to call functions provided by connected CMPLServers and additionally has to ensure a high availability and failover, the CMPLGridScheduler repeats failed CMPLServer calls whereby the number of tries are specified by the parameter `maxServerTries`. There is also a service thread that works permanently after a couple of seconds defined by the parameter `serviceIntervall`. Because this service thread is among others responsible for the problem waiting queue handling on the CMPLGridScheduler it makes sense to choose very short service intervals.

After running one or more CMPLGridSchedulers the involved CMPLServers can be started by typing the command:

```
cmplServer -startInGrid [<port>]
```



In addition to the described parameters in `cmplServer.opt` the following parameters are necessary for running a CMPLServer in a CMPLGrid.

```
...
maxServerTries = 3
performanceIndex = 1
cmplGridScheduler = http://10.0.1.52:8008  4
```

A CMPLServer in a CMPLGrid also has to call functions provided by a CMPLGridScheduler. Due to maximum availability and failover the maximum number of tries of failed CMPLGridScheduler calls are to be specified with the parameter `maxServerTries`. Assuming heterogeneous hardware for the CMPLServers in a CMPLGrid it is necessary for a reasonable load balancing to identify several performance levels of the invoked CMPLServers. This can be done by the parameter `performanceIndex` that influences the load balancing function directly. The involved operators of the CMPLServers and the CMPLGridScheduler(s) should specify standardised performance classes used within the entire CMPLGrid with the simple rule: the higher the performance class, the higher the `performanceIndex`. The parameter `cmplGridScheduler` is intended to specify the CMPLGridScheduler to which the CMPLServer is to be connected. The first argument is the URL of the scheduler. The second parameter defines the maximum number of problems that the CMPLServer provides to this CMPLGridScheduler. If a CMPLServer should be connected to more than one scheduler one entry per CMPLGridScheduler is required. In the following example the CMPLServer will be connected to two CMPLGridSchedulers with maximally two problems per scheduler.

```
...
cmplGridScheduler = http://10.0.1.52:8008  2
cmplGridScheduler = http://10.0.1.53:8008  2
```

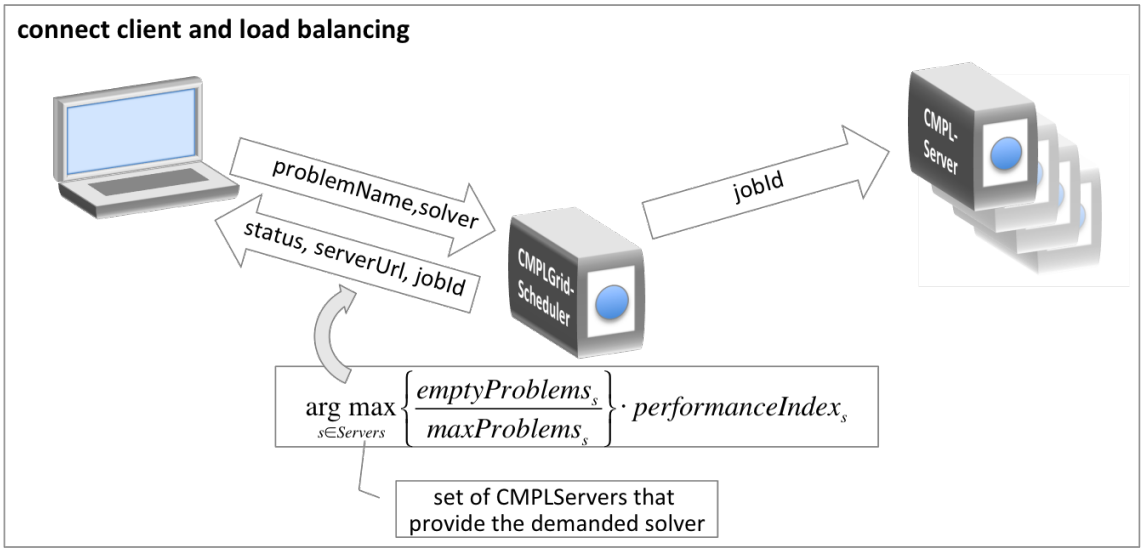
While connecting the CMPLGridScheduler the CMPLServer sends its port, the maximum number of provided problems and its performance index. It receives the status of the CMPLGridScheduler and a `serverId`. Possible states for connecting a CMPLServer are `CMPLGRID_SCHEDULER_OK` or `CMPLGRID_SCHEDULER _ERROR`.

Now a client can connect the CMPLGrid in the same way as a client connects a single CMPLServer either by using the CMPL binary

```
cmpl <problem>.cmpl -url http://<ip-adress-or-Domain>:<port>
```

or in pyCmpl and jCMPL programmes through the method `Cmpl.connect()`.

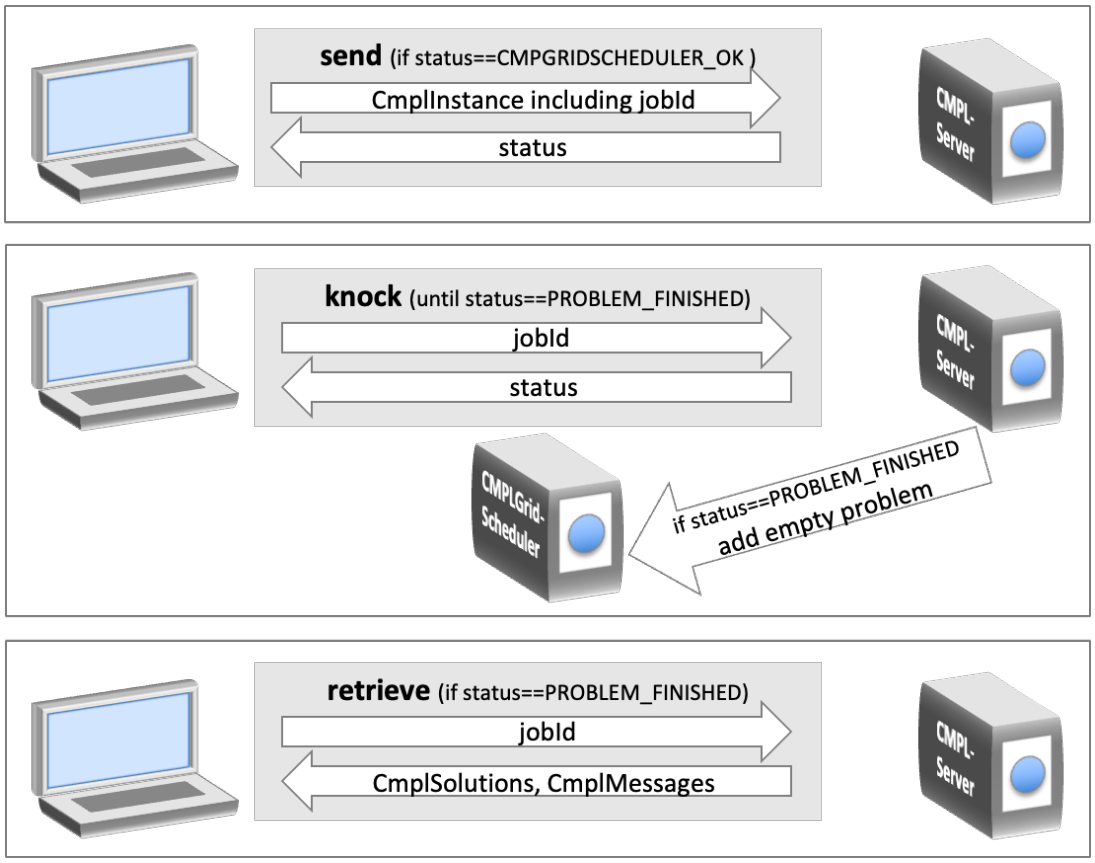
The client sends automatically the name of the problem and the name of the solver with which the problem should be solved to the CMPLGridScheduler.



If the name of the solver is unknown or this solver is not available in the CMPLGrid the CMPLGridScheduler returns `CMPLSERVER_ERROR`. In case the problem waiting queue is not empty the problem is then assigned to the problem waiting queue and the status is `CMPLGRID_SCHEDULER_BUSY`.

Otherwise the CMPLGridScheduler returns the status `CMPLGRID_SCHEDULER_OK`, the `serverUrl` of the CMPLServer on which the problem will be solved and the `jobId` of the problem. This CMPLServer is determined on the basis of the load balancing function that is shown in the picture below. Per server that is providing the solver the current capacity factor is to be calculated by the relationship between the current empty problems of this server and the maximum number of provided problems. The number of empty problems is controlled by the CMPLGridScheduler with a lower bound of zero and an upper bound equal to the maximum number of provided problems. This parameter is decreased if the CMPLServer is taking over a problem and it is increased when the CMPLServer has finished the problem or the problem is cancelled. The idea is to send problems tendentiously to those CMPLServer with the highest empty capacity. To include the different performance levels of the invoked CMPLServers in the load balancing decision, the current capacity factor is to

be multiplied by the performance index. The result is the load balancing factor and the CMPLServer with the highest load balancing factor is assigned to the client to solve the problem. This CMPLServer then gets the `jobId` of the CMPL problem by the CMPLGridServer in order to take over all relevant processes to solve this problem. Afterwards the client is automatically connected to this CMPLServer.



The problem waiting queue handling is organised by the CMPLGrid Scheduler service thread that assigns the waiting problems automatically to CMPLServers by using the same functionalities as described above. The waiting clients either ask automatically in the synchronous mode or manually in the asynchronous mode both through `Cmpl.knock()` until the received status is not equal to `CMPLGRID_SCHEDULER_BUSY`.

The next steps to solve the problem synchronously or asynchronously on the CMPLServer are similar to the procedures in the single server mode as shown in the following figure.

The methods `Cmpl.send()`, `Cmpl.knock()` and `Cmpl.retrieve()` are used to send the problem to the CMPLServer, to knock for the current status, to retrieve the solution and the CMPL and the solver messages and if requested some statistics. The main differences to the single server mode are that the CMPLServer calls the CMPLServerGrid to add an empty problem slot after finishing solving the problem and that the client is disconnected automatically from the CMPLServer after retrieving the solution, messages and statistics.

The CmplGridScheduler and the CmplServers can be stopped by typing the command:

```
cmplServer -stop [<port>]
```

3.5.3 Reliability and failover

A distributed optimisation system or a grid optimisation system is usually implemented in a heterogeneous environment. The network notes can be installed on different hardware as well as on different operating systems. This fact could cause some disturbances within the optimisation network that should be either avoided or reduced in their negative impact of the optimisation processes.

Beside ensuring a good performance, maximum reliability and failover are therefore important targets of the CMPLServer and the CMPLGrid implementations. They are ensured by:

(a) the problem queue handling on the CMPLGridScheduler and the CMPLServer,

(b) multiple executions of failed server calls and

(c) re-connections of problems to the CMPLGridScheduler if an assigned CMPLServer fails.

(a) Problem queue handling

If a problem is connected to a CMPLServer or a CMPLGridScheduler and the number of running problems including the model sent is greater than `maxProblems`, it neither makes sense to cancel the problem nor to interrupt the solving process. Especially in case of an iterating solving process with a couple of depending problems it is the better way to refer the supernumerary problems automatically to the problem waiting queue.

For the single server mode the problem queue handling is organised by the CMPLServer whilst in the grid mode the CMPLGridScheduler(s) are responsible for it. In both modes a problem stored in the problem waiting queue has to wait until an empty solving slot is available or the maximum queuing time is reached.

In the single server mode the number of problems that can be executed simultaneously on the particular CMPLServer are defined by the parameter `maxproblems` in `cmplServer.opt`. With this parameter it should be avoided to overwhelm the server and to avoid the super-proportional effort for managing a huge amount of parallel problems. The first empty solving slot that appears when a running problem is finished or cancelled, is taking over a waiting problem by using the FIFO approach.

The number of simultaneously running problems in a CMPLGrid is defined by the sum over all connected CMPLServer of the maximum number of problems provided by the servers. This parameter is to be defined per CMPLServer in `cmplServer.opt` as second argument in the entry `cmplGridScheduler = <`*`url`*`> <`*`maxProblems`*`>`. The CMPLGridScheduler counts the number of running problems per CMPLServer in relation to its maximum number of provided problems. If it is not possible to find a connected CMPLServer with an empty solving slot then the problem is put to the problem waiting queue. In contrast to the single server mode the problem which has been waiting longest is not executed by the first appearing free CMPLServer but it is organised by the described load balancing function over the set of CMPLServers that stated an empty solving slot during two iterations of the CMPLGridScheduler service thread.

The client's maximum queuing time in seconds can be specified with the CMPL command line argument `-maxTime <`*`sec`*`>`. This argument can also be set as CMPL header entry `%maxTime <`*`sec`*`>` or in pyCMPL and jCMPL with the method `Cmpl.setMaxServerQueuingTime(<`*`sec`*`>"`. The default value is 300 seconds.

(b) Multiple executions of failed server calls

To avoid that a single execution of a server method, which fails due to network problems like socket errors or others, cancels the entire process, all failed server calls can be executed again several times. The maximum number of executions of failed server calls can be specified for the clients by the CMPL command line argument `-maxTries <tries>`. It can also be used in a CMPL header entry `%maxTries <tries>` or in pyCMPL and jCMPL by using `Cmpl.setMaxServerTries(<tries>)`. The default value is 10. The number of maximum executions of failed server calls in the communication between the CMPLGridScheduler and CMPLServers is defined in `cmplServer.opt` with the entry `maxServerTries = <tries>`.

An exemplary and simplified implementation of this behaviour is shown in the pseudo code listing below:

```
1   serverTries=0
2   while True do
3       try
4         callServerMethod()
5       except
6         serverTries+=1
7         if serverTries>maxServerTries then
8              status=CMPLSERVER_ERROR
9              raise CmplException("calling CmplServer function … failed")
10        end if
11      end try
12      break
13  end while
```

In a first step the variable `serverTries` is assigned zero. The call of the server method (line 4) is imbedded in an infinite loop (lines 2-13) and in a try-except-block for the exception handling (lines 3-11). If no exception occurs then the loop is finished by the break command in line 12. Otherwise `serverTries` is incremented by 1. If the maximum number is not exceeded (line 7) the server method is called again (line 4). If `serverTries` is greater than `maxServerTries` then the class variable `Cmpl.status` is set to `CMPLSERVER_ERROR` and a `CmplException` is raised that have to be handled in the code in which the listing below is imbedded (lines 7-9).

(c) Re-connections of failed problems to the CMPLGridScheduler

Multiple server calls are mainly intended to prevent network problems. But it could be also possible that other problems caused by CMPLServers connected to a CMPLGridScheduler (e.g. a failed execution of a solver, file handling problems at a CMPLServer or the unpredictable shutdown of a CMPLServer) occur. The idea to handle such problems is that if the assigned CMPLServer fails the particular problem is then reconnected to the CMPLGridScheduler and is taken over by another CMPLServer automatically.

The following pseudo code listing describes a simplified implementation of `Cmpl.solve()` only for the grid mode to illustrate this approach:

```
1   serverTries=0
2   while True do
3       try
4         if status==CMPLSERVER_ERROR then
5              CmplGridScheduler.connect()
6         end if
7
8         if status==CMPLGRID_SCHEDULER_BUSY then
9              while status<>CMPLGRID_SCHEDULER_OK do
10                  CmplGridScheduler.knock()
11                  if waitingTime()>=maxQueuingTime then
12                       raise CmplException("max. queuing time is exceeded.")
13                  end if
14             end while
15        end if
16        connectedToServer=True
17
18        CmplServer.send()
19
20        while status<>PROBLEM_FINISHED do
21             CmplServer.knock()
22        end while
23
24        CmplServer.retrieve()
25        break
26
27      except CmplException
28        serverTries+=1
29        if status==CMPL_ERROR and connectedToServer==True then
30             CmplGridScheduler.cmplServerFailed()
31        end
32        if serverTries>maxServerTries or status==CMPLGRID_SCHEDULER_BUSY then
33             ExceptionHandling()
34             exit
35        end if
36      end try
37  end while
```

As in the listing of the multiple server calls the variable `serverTries` is assigned zero (line 1). The entire method is also imbedded in an infinite loop (lines 2-37) and the exception handling is organised as try-except-block (lines 3-36).

Before `Cmpl.solve()` is called the client has to execute `Cmpl.connect()` successfully. Therefore the class variable `Cmpl.status` has to be unequal to `CMPLSERVER_ERROR` and an additional `Cmpl.connect()` is not necessary in the first run of `Cmpl.solve()`(lines 4-6).It is possible that the entire CMPLGrid is busy, the status equals `CMPLGRID_SCHEDULER_BUSY` and the problem is moved to the CM-

PLGridScheduler problem waiting queue (line 8). In this case the problem has to wait for the next empty solving slot via `Cmpl.knock()` (line 10) until the CMPLGridScheduler returns the status `CMPLGRIDSCHEDULER_OK` (line 9) or the waiting time exceeds the maximum queuing time and a `CmplException` is raised (lines 11-13).

After this loop the problem is automatically connected to a CMPLServer within the CMPLGrid. The class variable `Cmpl.connectedToServer` is assigned `True` (line 16) and the problem is sent to this server through `Cmpl.send()` (line 18). The problem then has to wait until the problem status is `PROBLEM_FINISHED` (lines 20-22). As soon as the problem is finished, the solution(s), the CMPL and the solver messages as well as (if requested) some statistics can be retrieved via `Cmpl.retrieve()` (line 24). If no `CmplException` or another exception appeared during this procedures the infinite loop is left by the break command in line 25.

Otherwise the `CmplException` or other exceptions have to be handled in the except block in the lines 27-36. The first step is to increase the number of failed server call tries (line 28). If while executing `Cmpl.connect()`, `Cmpl.send()`, `Cmpl.knock()` or `Cmpl.retrieve()` an exception is raised and the problem is connected to a CMPLServer then the client calls the CMPLGridScheduler method `cmplServerFailed()` in order to report that this CMPLServer failed and to set the status of this server to `inactive` on the CMPLGridScheduler (line 30). This CMPLServer is then excluded from the CMPLGridScheduler load balancing until CMPLGridScheduler's service thread recognises that this CMPLServer is able to take over problems again.

If the number of failed server calls exceeds the maximum number of tries or the status is `CMPLGRID_SCHEDULER_BUSY` because the maximum queuing time is exceeded (line 32), the entire procedure stops by doing the necessary exception handling and by exiting the programme (lines 33-34).

Otherwise the problem has to pass the loop again. That means that the problem is reconnected to the CMPLGrid via `CMPLGridScheduler.connect()` (lines 4-6) and the solving process starts again.

3.6 pyCMPL

pyCMPL is the CMPL API for Python3. The main idea of this API is to define sets and parameters within the user application, to start and control the solving process and to read the solution(s) into the application if the problem is feasible. All variables, objective functions and constraints are defined in CMPL. These functionalities can be used with a local CMPL installation or a CMPLServer.

To execute a pyCmpl script, it is necessary to start the `cmplShell` script in the CMPL folder, which sets the CMPL environment (PATH, environment variables and library dependencies) and starts a command line window in which a pyCmpl script can be executed with the command `python <problemname>.py`. The CMPL package contains a Python environment with all necessary binaries, modules and packages. Other modules and packages can be added via the `PYTHONPATH` environment variable or installed directly in the Python environment supplied.

3.7 jCMPL

jCMPL is the CMPL API for Java. The main idea of this API is similar to pyCMPL to define sets and parameters within the user application, to start and control the solving process and to read the solution(s) into the application if the problem is feasible. All variables, objective functions and constraints are defined in CMPL.

These functionalities can be used with a local CMPL installation or a CMPLServer.

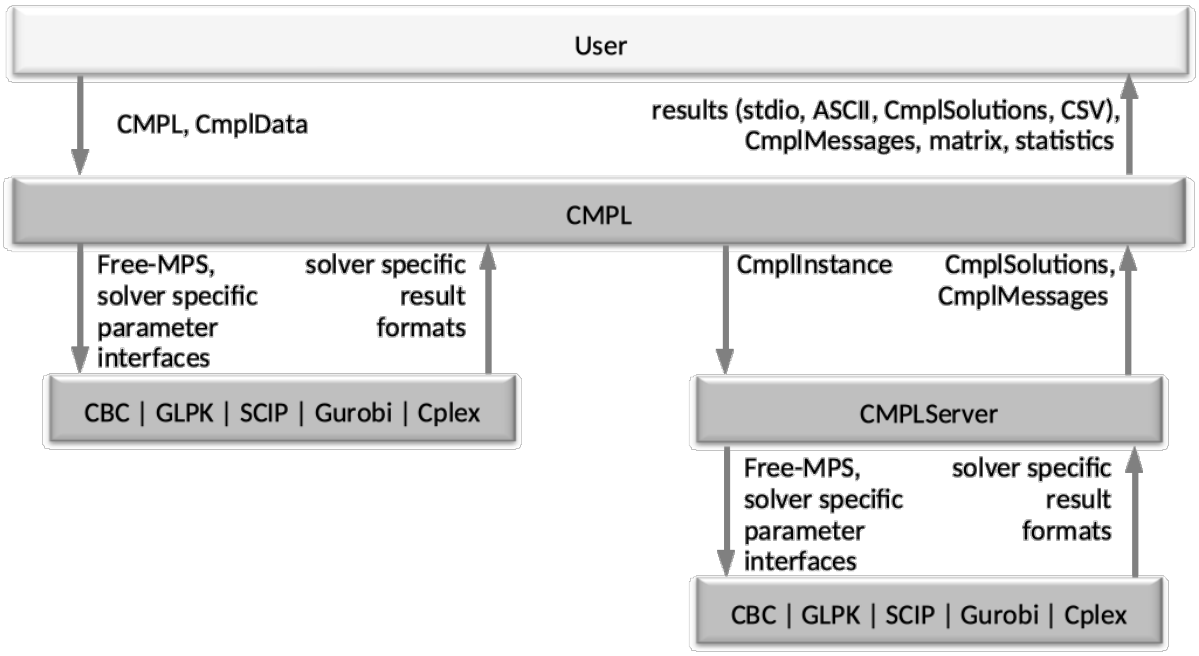
To use the jCMPL functionalities a Java programme has to import jCMPL by `import jCMPL.*;` and to link your application against `jCmpl.jar` and the following jar files, that you can find in the CMPL application folder in `jCmpl/lib` (Windows and Linux) or on GitHub (https://github.com/MikeSteglich/jCmpl).

Additionally, it is necessary to specify an environment variable `CMPLHOME` that contains the full path to the CMPL folder. This can be done by executing the `cmplShell` script in the Cmpl folder and to run the Java program in this environment.

3.8 Input and output file formats

3.8.1 Overview

As shown in the picture below CMPL uses several ASCII files for the communication with the user, the solvers and a CMPLServer.



CMPL	input file for CMPL - syntax as described above
CmplData	data file format for CMPL - syntax as described above
Free-MPS	output file for the generated model in Free-MPS format

CmplInstance	XML file that contains all relevant information about a CMPL model sent to a CMPLServer
Result files	solutions of a CMPL model can be obtained in the form of an ASCII, CSV or CmplSolutions file
CmplSolutions	solutions can be solved in CMPL's XML based solution file format
CmplMessages	XML file that contains the status and messages of a CMPL model

To describe the several file types it is necessary to distinguish between the local and the remote mode.

In the local mode a CMPL model and (if existing) the corresponding CmplData files are parsed and translated into a Free-MPS file (If no syntax or other error occur). If there are some errors in the CMPL model the CMPL messages are shown automatically or can be saved in a CmplMessages file. The Free-MPS file is together with solver specific parameter handed over to the chosen solver that is executed directly by CMPL. If the problem is feasible and an optimal solution is found CMPL reads the solution in form of the solver specific result format. A CMPL user can now obtain the standard solution report or can save the solution(s) as ASCII or CSV file or as CmplSolutions file. It is also possible to obtain the generated matrix and some statistics on the screen or in a plain text file.

A user can also process his or her CMPL model remotely on a CMPLServer. In the first step CMPL writes automatically all model relevant information (CMPL and CmplData files, CMPL and solver options) in a CmplInstance file and sends it to the connected CMPLServer. After solving the model CMPL receives two XML-based file formats (CmplSolutions, CmplMessages) and the user can obtain (if an optimal solution is found) the standard solution report or can save the solution(s) and also can get the generated matrix and some statistics. If the CMPL model contains errors then the user can retrieve the CMPL messages.

3.8.2 CMPL and CmplData

A CMPL file is an ASCII file that includes the user-defined CMPL code with a syntax as described in this manual.

The example

$1 \cdot x_1 + 2 \cdot x_2 + 3 \cdot x_3 \rightarrow max!$

$s.t.$

$5.6 \cdot x_1 + 7.7 \cdot x_2 + 10.5 \cdot x_3 \leq 15$

$9.8 \cdot x_1 + 4.2 \cdot x_2 + 11.1 \cdot x_3 \leq 20$

$0 \leq x_n ; n \in \{1,2,3\}$

can be formulated with the CmplData file `test.cdat`

```
%n set <1..2>
%m set <1..3>

%c[m] < 15 18 22 >
%b[n] < 175 200 >
%A[n,m] <   5 10 15
            10 5 10 >
```

and the CMPL file `test.cmpl`

```
%data test.cdat

var:
     x[m]: real[0..];
obj:
     profit: c^T * x[] -> max;
con:
     res: A * x <= b;
```

3.8.3 Free-MPS

The Free-MPS-format is internally used for the communication between CMPL and all local installed solvers.

The Free-MPS format is an improved version of the MPS format. There is no standard for this format but it is widely accepted. The structure of a Free-MPS file is the same as an MPS file. But most of the restricted MPS format requirements are eliminated, e.g. there are no requirements for the position or length of a field. For more information please visit the project website of the lp_solve project. [http://lpsolve.sourceforge.net]

The Free-MPS file for the given CMP example is generated as follows:

```
* CMPL - Free-MPS - Export
NAME test
* OBJNAME profit
* OBJSENSE MAX
ROWS
 N profit
 L res[1]
 L res[2]
COLUMNS
 x[1] profit 15 res[1] 5
 x[1] res[2] 10
 x[2] profit 18 res[1] 10
 x[2] res[2] 5
 x[3] profit 22 res[1] 15
 x[3] res[2] 10
RHS
 RHS res[1] 175 res[2] 200
BOUNDS
 PL BOUND x[1]
 PL BOUND x[2]
 PL BOUND x[3]
ENDATA
```

3.8.4 CmplInstance

CmplInstance is an XML-based format that contains all relevant information about a CMPL model (CMPL and CmplData files, CMPL and solver options) to be sent to a CMPLServer.

A CmplInstance file consists of three major sections. The `<general>` section contains the name of the problem and the jobId that is received automatically during connecting the CMPLServer. The `<options>` section consists of the CMPL and solver options that a user has specified on the command line. The `<problemFiles>` section is indented to store the CMPL file and all corresponding CmplData files. All CmplData

files no matter whether they are specified within the CMPL model or as command line argument are automatically included in the CmplInstance file. To avoid some misinterpretation of some special characters while reading the CmplInstance on the CMPLServer the content of the CMPL model and the CmplData files are automatically unescaped by CMPL.

The XSD (XML Schema Definition) of CmplInstance is defined as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:element name="CmplInstance">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="general" minOccurs="1" maxOccurs="1" />
                <xs:element ref="options" minOccurs="0" maxOccurs="1"/>
                <xs:element ref="problemFiles" minOccurs="1" maxOccurs="1"/>
            </xs:sequence>
            <xs:attribute name="version" type="xs:decimal" use="required"/>
        </xs:complexType>
    </xs:element>

    <xs:element name="general">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="name"  type="xs:string" minOccurs="1" maxOccurs="1"/>
                <xs:element  name="jobId"  type="xs:string" minOccurs="1" maxOccurs="1"/>
                <xs:element  name="preComp"  type="xs:string" minOccurs="1" maxOccurs="1"/>
               </xs:sequence>
            </xs:complexType>
    </xs:element>

    <xs:element name="options">
        <xs:complexType>
            <xs:sequence>
                <xs:element  name="opt" type="xs:string" minOccurs="0" maxOccurs="unbounded" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="problemFiles">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="file" minOccurs="1" maxOccurs="unbounded" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="file">
        <xs:complexType>
               <xs:simpleContent>
```

```
                    <xs:extension base="xs:string">
                    <xs:attribute name="name" type="xs:string" use="required"/>
                </xs:extension>
                </xs:simpleContent>
            </xs:complexType>
        </xs:element>

</xs:schema>
```

If the given example is run with `cmpl test.cmpl -url http://127.0.0.1:8008 -solver glpk` the CmplInstance file `test.cinst` is automatically created by CMPL, sent to the CmplServer and the model is executed remotely on a CMPLServer.

```
<?xml version = "1.0" encoding="UTF-8" standalone="yes"?>
<CmplInstance version="2.0">
<general>
<name>test.cmpl</name>
<jobId>S127.0.0.1-2021-05-15-16-36-52-809667</jobId>
<preComp>yes</preComp>
</general>
<options>
<opt>-solver glpk</opt>
</options>
<problemFiles>
<file name="test.cmpl" >
%data test.cdat

var:
    x[m]: real[0..];
obj:
    profit: c^T * x -&gt; max;
con:
    res: A * x &lt;= b;
</file>
<file name="test.cdat" >
%n set &lt;1..2&gt;
%m set &lt;1..3&gt;

%c[m] &lt; 15 18 22 &gt;
%b[n] &lt; 175 200 &gt;
%A[n,m] &lt; 5 10 15
      10 5 10 &gt;
</file>
</problemFiles>
</CmplInstance>
```

3.8.5 ASCII or CSV result files

If the problem is feasible and an optimal solution is found a user can obtain this optimal solution in the form of an ASCI or CSV file by using the command line arguments `-solutionAscii [<file>]` or `-solutionCsv [<file>]`. This files can additionally contain all integer feasible solutions if Cplex or Gurobi are used and the the CMPL header option `%display solutionPool` is defined.

The ASCII result file `test.sol` for the given CMPL example is generated as follows:

```
-------------------------------------------------------------------------------------------------------------
Problem              test.cmpl
Nr. of variables     3
Nr. of constraints   2
Objective name       profit
Solver name          GLPK
Display variables    (all)
Display constraints  (all)
-------------------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      405.00              (max!)

Variables
Name                 Type            Activity          LowerBound          UpperBound            Marginal
-------------------------------------------------------------------------------------------------------------
x[1]                    C               15.00                0.00                 inf                0.00
x[2]                    C               10.00                0.00                 inf                0.00
x[3]                    C                0.00                0.00                 inf               -7.00
-------------------------------------------------------------------------------------------------------------

Constraints
Name                 Type            Activity          LowerBound          UpperBound            Marginal
-------------------------------------------------------------------------------------------------------------
res[1]                  L              175.00                -inf              175.00                1.40
res[2]                  L              200.00                -inf              200.00                0.80
-------------------------------------------------------------------------------------------------------------
```

The corresponding CSV result file `test.csv` is generated as follows:

```
CMPL csv export

Problem;test.cmpl
Nr. of variables;3
Nr. of constraints;2
Objective name;profit
Solver name;GLPK
Display variables;(all)
Display constraints;(all)

Objective status;optimal
Objective value;405.000000;(max!)
Variables
Name;Type;Activity;LowerBound;UpperBound;Marginal
x[1];C;15.000000;0.000000;inf;0.000000
x[2];C;10.000000;0.000000;inf;0.000000
x[3];C;0.000000;0.000000;inf;-7.000000
Constraints
Name;Type;Activity;LowerBound;UpperBound;Marginal
res[1];L;175.000000;-inf;175.000000;1.400000
res[2];L;200.000000;-inf;200.000000;0.800000
```

3.8.6 CmplSolutions

CmplSolutions is an XML-based format for representing the general status and the solution(s) if the problem is feasible and one or more solutions are found. A user can save it by using the command line argument `-solution [<File>]`. It is also internally used for receiving solution(s) from a CMPLServer.

As shown in the corresponding XSD below A CmplSolutions file contains a `<general>` block for general information about the solved problem and a `<solutions>` block for the results of all solutions found including the variables and constraints.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">

  <xs:element name="CmplSolutions">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="general" minOccurs="1" maxOccurs="1"/>
        <xs:element ref="solution" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="version" use="required" type="xs:decimal"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="general">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="instanceName" type="xs:string" minOccurs="1" maxOccurs="1" />
        <xs:element name="nrOfVariables" type="xs:nonNegativeInteger" minOccurs="1" maxOccurs="1"/>
        <xs:element name="nrOfConstraints" type="xs:nonNegativeInteger" minOccurs="1" maxOccurs="1"
/>
        <xs:element name="objectiveName" type="xs:string" minOccurs="1" maxOccurs="1" />
        <xs:element name="objectiveSense" type="xs:string" minOccurs="1" maxOccurs="1" />
        <xs:element name="nrOfSolutions" type="xs:nonNegativeInteger" minOccurs="1" maxOccurs="1"/>
        <xs:element name="solverName" type="xs:string" minOccurs="1" maxOccurs="1" />
        <xs:element name="variablesDisplayOptions" type="xs:string" minOccurs="1" maxOccurs="1" />
        <xs:element name="constraintsDisplayOptions" type="xs:string" minOccurs="1" maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="solution">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="variables" minOccurs="1" maxOccurs="1"/>
        <xs:element ref="linearConstraints"  minOccurs="1" maxOccurs="1" />
      </xs:sequence>
      <xs:attribute name="idx" use="required" type="xs:nonNegativeInteger"/>
      <xs:attribute name="status" use="required" type="xs:string"/>
      <xs:attribute name="value" use="required" type="xs:decimal"/>
```

```
  </xs:complexType>
</xs:element>

<xs:element name="variables">
  <xs:complexType>
    <xs:sequence>
      <xs:element  minOccurs="1" maxOccurs="unbounded" ref="variable"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="linearConstraints">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="unbounded" ref="constraint"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="variable">
  <xs:complexType>
    <xs:attribute name="idx" use="required" type="xs:nonNegativeInteger"/>
    <xs:attribute name="name" use="required" type="xs:string"/>
    <xs:attribute name="type" use="required" type="varType"/>
        <xs:attribute name="activity" use="required" type="xs:double"/>
    <xs:attribute name="lowerBound" use="required" type="xs:double"/>
    <xs:attribute name="upperBound" use="required" type="xs:double"/>
    <xs:attribute name="marginal" use="required" type="xs:double"/>
  </xs:complexType>
</xs:element>

<xs:element name="constraint">
  <xs:complexType>
    <xs:attribute name="idx" use="required" type="xs:nonNegativeInteger"/>
    <xs:attribute name="name" use="required" type="xs:string"/>
    <xs:attribute name="type" use="required" type="conType"/>
        <xs:attribute name="activity" use="required" type="xs:double"/>
    <xs:attribute name="lowerBound" use="required" type="xs:double"/>
    <xs:attribute name="upperBound" use="required" type="xs:double"/>
    <xs:attribute name="marginal" use="required" type="xs:double"/>
  </xs:complexType>
</xs:element>

<xs:simpleType name="varType">
      <xs:restriction base="xs:string">
             <xs:enumeration value="C"/>
             <xs:enumeration value="I"/>
             <xs:enumeration value="B"/>
      </xs:restriction>
```

```
  </xs:simpleType>

  <xs:simpleType name="conType">
        <xs:restriction base="xs:string">
               <xs:enumeration value="L"/>
               <xs:enumeration value="E"/>
               <xs:enumeration value="G"/>
        </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

The CmplSolutions file `test.csol` for the given CMPL example is generated as follows:

```
<?xml version = "1.0" encoding="UTF-8" standalone="yes"?>
<CmplSolutions version="1.1">
   <general>
       <instanceName>test.cmpl</instanceName>
       <nrOfVariables>3</nrOfVariables>
       <nrOfConstraints>2</nrOfConstraints>
       <objectiveName>profit</objectiveName>
       <objectiveSense>max</objectiveSense>
       <nrOfSolutions>1</nrOfSolutions>
       <solverName>GLPK</solverName>
       <solverMsg>normal</solverMsg>
       <variablesDisplayOptions>(all)</variablesDisplayOptions>
       <constraintsDisplayOptions>(all)</constraintsDisplayOptions>
   </general>
   <solution idx="0" status="optimal" value="405">
       <variables>
           <variable idx="0" name="x[1]" type="C" activity="15" lowerBound="0"
              upperBound="inf" marginal="0"/>
           <variable idx="1" name="x[2]" type="C" activity="10" lowerBound="0"
              upperBound="inf" marginal="0"/>
           <variable idx="2" name="x[3]" type="C" activity="0" lowerBound="0"
              upperBound="inf" marginal="-7"/>
       </variables>
       <linearConstraints>
           <constraint idx="0" name="res[1]" type="L" activity="175"
              lowerBound="-inf" upperBound="175" marginal="1.4"/>
           <constraint idx="1" name="res[2]" type="L" activity="200"
              lowerBound="-inf" upperBound="200" marginal="0.8"/>
       </linearConstraints>
   </solution>
</CmplSolutions>
```

3.8.7 CmplMessages

CmplMessages is an XML-based format for representing the general status and/or errors of the transformation of a CMPL model in one of the described output files. CmplMessages is intended for communication with other software that uses CMPL for modelling linear optimisation problems and can be obtained by the command line argument `-cmsg [<file>]`.

It is also internally used for receiving CMPL messages from a CMPLServer.

An CmplMessages file consists of two major sections. The `<general>` section describes the general status and the name of the model and a general message after the transformation. The `<messages>` section consists of one or more messages about specific lines in the CMPL model.

The XSD is defined as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">

  <xs:element name="CmplMessages">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="general" minOccurs="1" maxOccurs="1"/>
        <xs:element ref="messages" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="version" use="required" type="xs:decimal"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="general">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="generalStatus"  type="xs:string" minOccurs="1" maxOccurs="1"/>
        <xs:element name="instanceName" type="xs:string" minOccurs="1" maxOccurs="1"/>
        <xs:element name="message" type="xs:string" minOccurs="0" maxOccurs="1"/>
         <xs:element name="cmplVersion" type="xs:string" minOccurs="1" maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="messages">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="message" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="numberOfMessages" use="required" type="xs:nonNegativeInteger"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="message">
       <xs:complexType>
```

```
            <xs:attribute name="type" type="msgType" use="required"/>
            <xs:attribute name="module" type="xs:string" use="required"/>
            <xs:attribute name="location" type="xs:string" use="required"/>
            <xs:attribute name="description" type="xs:string" use="required"/>
      </xs:complexType>
  </xs:element>

  <xs:simpleType name="msgType">
      <xs:restriction base="xs:string">
            <xs:enumeration value="error"/>
            <xs:enumeration value="warning"/>
      </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

After excecuting the given CMPL model, CMPL will finish without errors. The general status is represented in the following CmplMesages file `test.cmsg`.

```
<?xml version="1.0" encoding="UTF-8"?>
<CmplMessages version="1.2">
    <general>
      <instanceName>test.cmpl</instanceName>
      <generalStatus>normal</generalStatus>
      <message>cmpl finished normal</message>
      <cmplVersion>2.0.0</cmplVersion>
    </general>
</CmplMessages>
```

If a wrong symbol name for the matrix `A` (e.g. `a`) is used in line 10 , CMPL would be finish with errors represented in CmplMesages file `test.cmsg`.

```
<?xml version="1.0" encoding="UTF-8"?>
<CmplMessages version="1.2">
    <general>
      <instanceName>test.cmpl</instanceName>
      <generalStatus>error</generalStatus>
      <message>cmpl finished with errors</message>
      <cmplVersion>2.0.0 (beta)</cmplVersion>
    </general>
    <messages numberOfMessages="1">
      <message type ="error" module ="compile" location="test.cmpl:10.11,
called from: command line:$1" description="symbol &apos;a&apos; is not
defined"/>
    </messages>
</CmplMessages>
```

4 CMPL's APIs

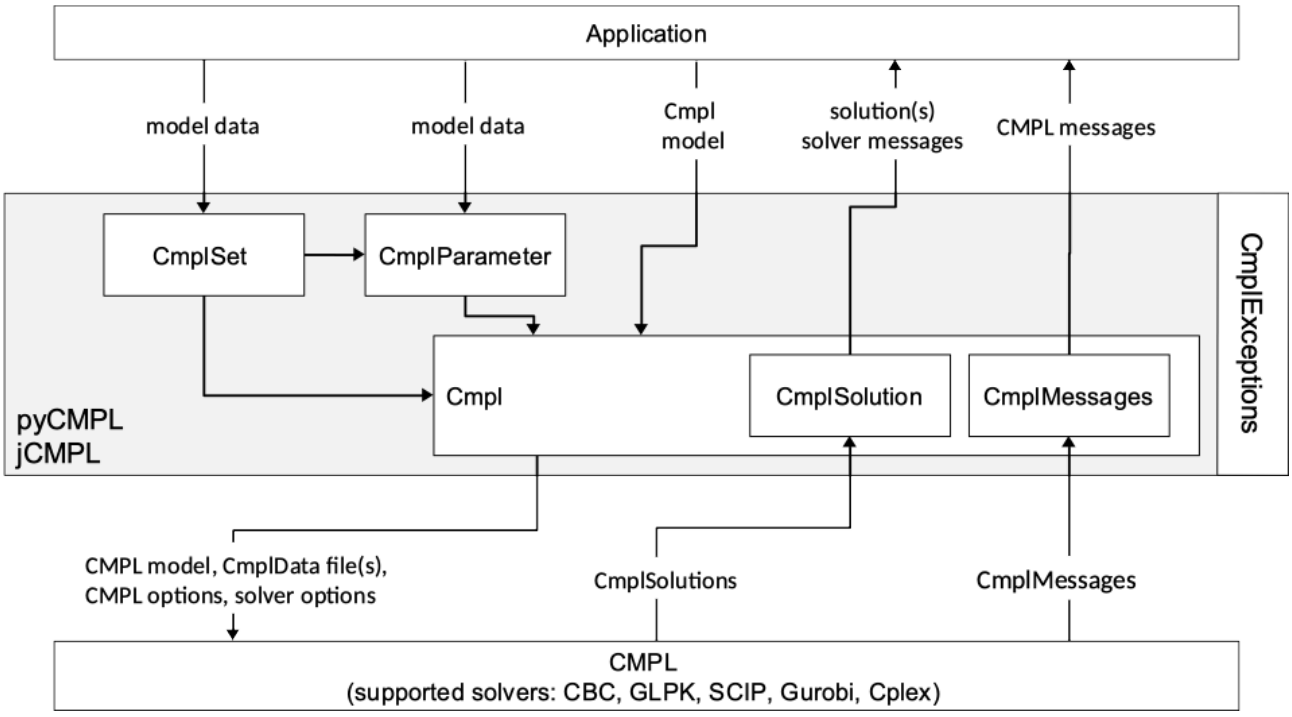
CMPL provides two APIs: pyCMPL for Python and jCMPL for Java.

The main idea of this APIs is to define sets and parameters within the user application, to start and control the solving process and to read the solution(s) into the application if the problem is feasible. All variables, objective functions and constraints are defined in CMPL. These functionalities can be used with a local CMPL installation or a CMPLServer.

The structure and the classes including the methods and attributes are mostly identical or very similar in both APIs. The main difference are the attributes of a class that can be obtained in pyCmpl by r/o attributes and in jCMPL by getter methods.

4.1 Creating Python and Java applications with a local CMPL installation

pyCMPL and jCMPL contain a couple of classes to connect a Python or Java application with CMPL as shown in the figure below. .



The classes `CmplSet` and `CmplParameter` are intended to define sets and parameters that can be used with several `Cmpl` objects. With the `Cmpl` class it is possible to define a CMPL model, to commit sets and parameters to this model, to start and control the solving process and to read the CMPL and solver messages and to have access to the solution(s) via `CmplMessages` and `CmplSolutions` objects.

To illustrate the formulation of a pyCmpl script and the corresponding java programme an example taken from (Hillier/Liebermann 2010, p. 334f.) is used. Consider a simple assignment problem that deals with the assignment of three machines to four possible locations. There is no work flow between the machines. The total material handling costs are to be minimised. The hourly material handling costs per machine and location are given in the following table.

		Locations			
		1	2	3	4
Machines	1	13	16	12	11
	2	15	-	13	20
	3	5	7	10	6

The mathematical model

$$\sum_{(i,j)\in A} c_{ij} \cdot x_{ij} \quad \rightarrow \min!$$

$$s.t.$$

$$\sum_{\substack{(k,j)\in A \\ k=i}} x_{kj} = 1 \quad ; i = 1(1)m$$

$$\sum_{\substack{(i,l)\in A \\ l=j}} x_{il} \leq 1 \quad ; j = 1(1)n$$

$$x_{ij} \in \{0,1\} \quad ;(i,j) \in A$$

with

Parameters

A	- set of the possible combination of machines and locations
m	- number of machines
n	- number of locations
c_{ij}	- hourly material handling costs of machine i at location j

Variables

x_{ij}	- assignment variable of machine i at location j

can be formulated in CMPL as follows:

```
%data : machines set, locations set, A set[2], c[A]

var:
  x[A]: binary;

obj:
  costs: sum{  [i,j] in A : c[i,j]*x[i,j] } -> min ;

con:
  { i in machines: sos_m[i]: sum{ j in (A *> [i,*])  : x[i,j] } = 1; }
  { j in locations: sos_l[j]:  sum{ i in (A *> [*,j])  : x[i,j] } <= 1; }
```

The interface for the sets and parameters provided by a pyCmpl script or jCMPL programme is the CMPL header entry `%data`.

4.1.1 pyCMPL

The first step to formulate this problem as a pyCmpl script after importing the pyCmpl package is to create a `Cmpl` object where the argument of the constructor is the name of the CMPL file.

```
from pyCmpl import *

m = Cmpl("assignment.cmpl")
```

As in the `%data` entry two 1-tuple sets `machines` and `locations` and one 2-tuple set `A` are necessary for the CMPL model. To create a `CmplSet` a name and for *n*-tuple sets with *n*>1 the rank are needed as arguments for the constructor. The name has to be identical to the corresponding name in the CMPL header entry `%data`. The set data is specified by the `CmplSet` method `setValues`. This is an overloaded method with different arguments for several types of sets.

```
locations = CmplSet("locations")
locations.setValues(1,4)

machines = CmplSet("machines")
machines.setValues(1,3)

combinations = CmplSet("A", 2)
combinations.setValues([  [1,1],[1,2],[1,3],[1,4], [2,1],[2,3],[2,4],\
                          [3,1],[3,2],[3,3],[3,4]])
```

As shown in the listing above the set `locations` is assigned `(1,2,..,4)` and the set `machines` consists of `(1,2,3)` because the first argument of `setValues` for this kind of sets is the starting value and the second argument is the end value while the increment is by default equal to one. The values of the 2-tuple set `combinations` are defined in the form of a list that consists of lists of valid combinations of machines and locations.

For the definition of a CMPL parameter a user has to create a `CmplParameter` object where the first argument of the constructor is the name of the parameter. If the parameter is an array it is also necessary to specify the set or sets through which the parameter array is defined. Therefore it is necessary to commit the `CmplSet combinations` (beside the name `"c"`) to create the CmplParameter array `c` .

```
c = CmplParameter("c",combinations)
c.setValues([13,16,12,11,15,13,20,5,7,10,6])
```

`CmplSet` objects and `CmplParameter` objects can be used in several CMPL models and have to be committed to a `Cmpl` model by the `Cmpl` methods `setSets` and `setParameters`. After this step the problem can be solved by using the `Cmpl` method `solve`.

```
m.setSets(machines,locations,combinations)
m.setParameters(c)

m.solve()
```

After solving the model the status of CMPL and the invoked solver can be analysed through the `Cmpl` attributes `solution.solverStatus` and `solution.cmplStatus`.

```
print("Objective value: " , m.solution.value)
print("Objective status: " , m.solution.status)
```

If the problem is feasible and a solution is found it is possible to read the names, the types, the activities, the lower and upper bounds and the marginal values of the variables and the constraints into the Python application. The `Cmpl` attributes `solution.variables` and `solution.constraints` contain a list of variable and constraint objects.

```
print("Variables:")
  for v in m.solution.variables:
    print(("%10s %3s %8i %8i %8i" % (v.name, v.type, v.activity, v.lowerBound,
         v.upperBound  )))

  print("Constraints:")
  for c in m.solution.constraints:
    print(("%10s %3s %8.0f %8.0f %8.0f" % (c.name, c.type, c.activity,
         c.lowerBound,c.upperBound)))
```

pyCmpl provides its own exception handling through the class `CmplException` that can be used in a `try` and `except` block.

```
try:
  ...
except CmplException as e:
  print(e.msg)
```

The entire pyCmpl script assignment.py shows as follows:

```
from pyCmpl import *

try:
  m = Cmpl("assignment.cmpl")

  locations = CmplSet("locations")
  locations.setValues(1,4)

  machines = CmplSet("machines")
  machines.setValues(1,3)

  combinations = CmplSet("A", 2)
  combinations.setValues([ [1,1],[1,2],[1,3],[1,4],[2,1],[2,3],[2,4],
                           [3,1],[3,2],[3,3],[3,4]])

  c = CmplParameter("c",combinations)
  c.setValues([13,16,12,11,15,13,20,5,7,10,6])

  m.setSets(machines,locations,combinations)
  m.setParameters(c)

  m.solve()

  print("Objective value: " , m.solution.value)
  print("Objective status: " , m.solution.status)
```

```
  print("Variables:")
  for v in m.solution.variables:
    print(("%10s %3s %8i %8i %8i" % (v.name, v.type, v.activity,
         v.lowerBound,v.upperBound  )))

  print("Constraints:")
  for c in m.solution.constraints:
    print(("%10s %3s %8.0f %8.0f %8.0f" % (c.name, c.type, c.activity,
         c.lowerBound,c.upperBound)))

except CmplException as e:
  print(e.msg)
```

and can be executed by typing the command

```
python assignment.py
```

in the CmplShell and prints the following solution to stdOut.

```
Objective value:  29.0
Objective status:  optimal
Variables:
    x[1,1]   B        0        0        1
    x[1,2]   B        0        0        1
    x[1,3]   B        0        0        1
    x[1,4]   B        1        0        1
    x[2,1]   B        0        0        1
    x[2,3]   B        1        0        1
    x[2,4]   B        0        0        1
    x[3,1]   B        1        0        1
    x[3,2]   B        0        0        1
    x[3,3]   B        0        0        1
    x[3,4]   B        0        0        1
Constraints:
  sos_m[1]   E        1        1        1
  sos_m[2]   E        1        1        1
  sos_m[3]   E        1        1        1
  sos_l[1]   L        1     -inf        1
  sos_l[2]   L        0     -inf        1
  sos_l[3]   L        1     -inf        1
  sos_l[4]   L        1     -inf        1
```

4.1.2 jCMPL

To use the jCMPL functionalities a Java programme has to import jCMPL by `import jCMPL.*;` and to link your application against `jCmpl.jar` and the following jar files, that you can find in the CMPL applicatiopn folder in `jCmpl/Libs` or on GitHub (https://github.com/MikeSteglich/jCmpl).

The first step to formulate this problem as a jCmpl programme after importing the jCmpl package is to create a `Cmpl` object where the argument of the constructor is the name of the CMPL file. Since jCMPL provides it own exception handling the main method has to throw `CmplExeptions`.

```
import jCMPL.*;

public class Assignment {
    public static void main(String[] args) throws CmplException {
        try {
            Cmpl m = new Cmpl("assignment.cmpl");
```

As in pyCMPL to create a `CmplSet` a name and for *n*-tuple sets with *n*>1 the rank are needed as arguments for the constructor whereby the name has to be identical to the corresponding name in the CMPL header entry `%data`. The set data is specified by the `CmplSet.setValues()`. This is an overloaded method with different arguments for several types of sets.

```
CmplSet locations = new CmplSet("locations");
locations.setValues(1, 4);

CmplSet machines = new CmplSet("machines");
machines.setValues(1, 3);

CmplSet combinations = new CmplSet("A", 2);
int[][] combiVals = {{1, 1}, {1, 2}, {1, 3}, {1, 4}, {2, 1},
                     {2, 3}, {2, 4}, {3, 1}, {3, 2}, {3, 3}, {3, 4}};
combinations.setValues(combiVals);
```

In the listing above the set `locations` is assigned `(1,2,..,4)` and the set `machines` consists of `(1,2,3)`. The first argument of `setValues` for this algorithmic sets is the starting value and the second argument is the end value while the increment is by default equal to one. The values of the 2-tuple set `combinations` are defined in the form of a matrix of integers that consists all valid combinations of machines and locations.

To create a CMPL parameter a user has to define a `CmplParameter` object whereby the first argument of the constructor is the name of the parameter. For parameter arrays it is also necessary to specify the set or sets through which the parameter array is defined. Therefore it is necessary to commit the `CmplSet combinations` (beside the name `"c"`) to create the CmplParameter array `c`.

```
CmplParameter costs = new CmplParameter("c", combinations);
int[] costVals = {13, 16, 12, 11, 15, 13, 20, 5, 7, 10, 6};
costs.setValues(costVals);
```

In the next step the sets and parameters have to be committed to a `Cmpl` model by the `Cmpl` methods `setSets` and `setParameters` and the problem can be solved by using the `Cmpl` method `solve`.

```
m.setSets(machines, locations, combinations);
m.setParameters(costs);
m.solve();
```

After solving the model the status of CMPL and the invoked solver can be analysed through the methods `Cmpl.solution().solverStatus()` and `Cmpl.solution().cmplStatus()`.

```
System.out.printf("Objective value:  %f %n", m.solution().value());
System.out.printf("Objective status: %s %n",  m.solution().status());
```

If the problem is feasible and a solution is found it is possible to read the names, the types, the activities, the lower and upper bounds and the marginal values of the variables and the constraints into the Python application. The methods `Cmpl.solution().variables()` and `Cmpl.solution().constraints()` return a list of variable and constraint objects.

```
System.out.println("Variables:");
for (CmplSolElement v :  m.solution().variables()) {
  System.out.printf("%10s %3s %10d %10.0f %10.0f%n", v.name(), v.type(),
         v.activity(), v.lowerBound(), v.upperBound());
}
System.out.println("Constraints:");
for (CmplSolElement c :  m.solution().constraints()) {
  System.out.printf("%10s %3s %10.0f %10.0f %10.0f%n", c.name(), c.type(),
       c.activity(), c.lowerBound(), c.upperBound());
}
```

The entire jCmpl programme `assignment.java` shows as follows:

```
import jCMPL.*;

public class Assignment1 {
  public static void main(String[] args) throws CmplException {
    try {
      Cmpl m = new Cmpl("assignment.cmpl");

      CmplSet locations = new CmplSet("locations");
      locations.setValues(1, 4);

      CmplSet machines = new CmplSet("machines");
      machines.setValues(1, 3);

      CmplSet combinations = new CmplSet("A", 2);
      int[][] combiVals = {{1, 1}, {1, 2}, {1, 3}, {1, 4},{2, 1}, {2, 3},
                           {2, 4},{3, 1}, {3, 2}, {3, 3}, {3, 4}};
      combinations.setValues(combiVals);

      CmplParameter costs = new CmplParameter("c", combinations);
      int[] costVals = {13, 16, 12, 11, 15, 13, 20, 5, 7, 10, 6};
      costs.setValues(costVals);

      m.setSets(machines, locations, combinations);

      m.setParameters(costs);

      m.solve();

      System.out.printf("Objective value:  %f %n", m.solution().value());
      System.out.printf("Objective status: %s %n",  m.solution().status());
```

```
      System.out.println("Variables:");
      for (CmplSolElement v :  m.solution().variables()) {
               System.out.printf("%10s %3s %10d %10.0f %10.0f%n", v.name(),
                    v.type(), v.activity(), v.lowerBound(), v.upperBound());
      }
      System.out.println("Constraints:");
      for (CmplSolElement c :  m.solution().constraints()) {
        System.out.printf("%10s %3s %10.0f %10.0f %10.0f%n", c.name(),
                    c.type(), c.activity(), c.lowerBound(), c.upperBound());
      }
    } catch (CmplException e) {
      System.out.println(e);
    }
  }
}
```

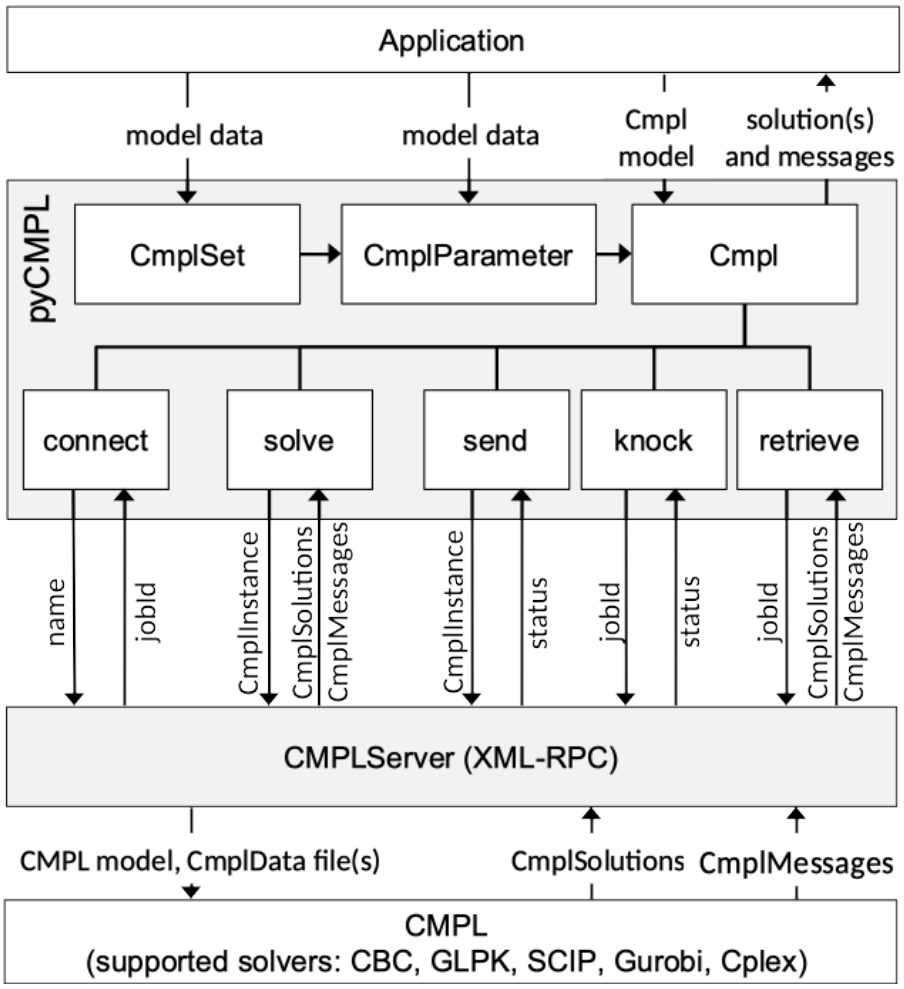
and prints after starting the following solution to stdOut.

```
Objective value:  29.000000
Objective status: optimal
Variables:
    x[1,1]   B          0          0          1
    x[1,2]   B          0          0          1
    x[1,3]   B          0          0          1
    x[1,4]   B          1          0          1
    x[2,1]   B          0          0          1
    x[2,3]   B          1          0          1
    x[2,4]   B          0          0          1
    x[3,1]   B          1          0          1
    x[3,2]   B          0          0          1
    x[3,3]   B          0          0          1
    x[3,4]   B          0          0          1
Constraints:
  sos_m[1]   E          1          1          1
  sos_m[2]   E          1          1          1
  sos_m[3]   E          1          1          1
  sos_l[1]   L          1  -Infinity          1
  sos_l[2]   L          0  -Infinity          1
  sos_l[3]   L          1  -Infinity          1
  sos_l[4]   L          1  -Infinity          1
```

4.2 Creating Python and Java applications using CMPLServer

The class `Cmpl` also provides the functionality to communicate with a CMPLServer or a CMPLGridScheduler whereas it doesn't matter for the client whether it is connected to a single CMPLServer or to a CMPLGrid. As

shown in the figure below the first step to communicate with a CMPLServer is the `Cmpl.connect` method that returns (if connected) a `jobId`. After connecting, a problem can be solved synchronously or asynchronously.



The `Cmpl` method `solve` sends a `CmplInstance` string to the connected CMPLServer and waits for the returning `CmplSolutions`, `CmplMessages` XML strings. After this synchronous process a user can access the solution(s) if the problem is feasible or if not it can be analysed, whether the CMPL formulations or the solver is the cause of the problem. To execute the solving process asynchronously the `Cmpl` methods `send`, `knock` and `retrieve` have to be used. `Cmpl.send` sends a `CmplInstance` string to the CMPLServer and starts the solving process remotely. `Cmpl.knock` asks for a CMPL model with a given `jobId` whether the solving process is finished or not. If the problem is finished the `CmplSolutions` and the `CmplMessages` strings can be read into the user application with `Cmpl.retrieve`.

4.2.1 pyCMPL

The first step to create a distributed optimisation application is to start the CMPLServer. Assuming that a CMPLServer is running on `127.0.0.1:8008` the assignment problem can be solved remotely only by including

```
m.connect("http://127.0.0.1:8008")
```

in the source code before `Cmpl.solve` is executed.

The pyCmpl script `assignment-remote.py` shows as follows:

```
from pyCmpl import *

try:
  m = Cmpl("assignment.cmpl")

  locations = CmplSet("locations")
  locations.setValues(1,4)

  machines = CmplSet("machines")
  machines.setValues(1,3)

  combinations = CmplSet("A", 2)
  combinations.setValues([  [1,1],[1,2],[1,3],[1,4],  [2,1],[2,3],[2,4],  [3,1],
[3,2],[3,3],[3,4]])

  c = CmplParameter("c",combinations)
  c.setValues([13,16,12,11,15,13,20,5,7,10,6])

  m.setSets(machines,locations,combinations)
  m.setParameters(c)

  m.connect("http://127.0.0.1:8008")
  m.solve()

  print("Objective value: " , m.solution.value)
  print("Objective status: " , m.solution.status)

  print("Variables:")
  for v in m.solution.variables:
    print(("%10s %3s %8i %8i %8i" % (v.name, v.type, v.activity,
           v.lowerBound,v.upperBound  )))

  print("Constraints:")
  for c in m.solution.constraints:
    print(("%10s %3s %8.0f %8.0f %8.0f" % (c.name, c.type, c.activity,
         c.lowerBound,c.upperBound)))

except CmplException as e:
  print(e.msg)
```

4.2.2 jCMPL

The jCMPL programme `assignment-remote.java` shows as follows:

```
import jCMPL.*;

public class Assignment1 {
  public static void main(String[] args) throws CmplException {
    try {
      Cmpl m = new Cmpl("assignment.cmpl");

      CmplSet locations = new CmplSet("locations");
      locations.setValues(1, 4);

      CmplSet machines = new CmplSet("machines");
      machines.setValues(1, 3);

      CmplSet combinations = new CmplSet("A", 2);
      int[][] combiVals = {{1, 1}, {1, 2}, {1, 3}, {1, 4},{2, 1}, {2, 3},
                           {2, 4},{3, 1}, {3, 2}, {3, 3}, {3, 4}};
      combinations.setValues(combiVals);

      CmplParameter costs = new CmplParameter("c", combinations);
      int[] costVals = {13, 16, 12, 11, 15, 13, 20, 5, 7, 10, 6};
      costs.setValues(costVals);

      m.setSets(machines, locations, combinations);
      m.setParameters(costs);

      m.connect("http://127.0.0.1:8008");
      m.solve();

      System.out.printf("Objective value:  %f %n", m.solution().value());
      System.out.printf("Objective status: %s %n",  m.solution().status());


      System.out.println("Variables:");
      for (CmplSolElement v :  m.solution().variables()) {
               System.out.printf("%10s %3s %10d %10.0f %10.0f%n", v.name(),
                    v.type(), v.activity(), v.lowerBound(), v.upperBound());
      }
      System.out.println("Constraints:");
      for (CmplSolElement c :  m.solution().constraints()) {
        System.out.printf("%10s %3s %10.0f %10.0f %10.0f%n", c.name(),
                    c.type(), c.activity(), c.lowerBound(), c.upperBound());
      }
```

```
    } catch (CmplException e) {
      System.out.println(e);
    }
  }
}
```

4.3 pyCMPL reference manual

4.3.1 CmplSets

The class `CmplSet` is intended to define sets that can be used with several `Cmpl` objects.

Methods:

`CmplSet(`*`setName[,rank]`***`)`**

Description: Constructor

Parameter:
- `str` *`setName`* — name of the set, Has to be equal to the corresponding name in the CMPL model.
- `int` *`rank`* — optional - rank *n* for a *n*-tuple set (default 1)

Return: `CmplSet` object

`CmplSet``.`**`setValues(`***`setList`***`)`**

Description: Defines the values of an enumeration set

Parameter:
- `list` *`setList`* — for a set of *n*-tuples with *n*=1 - `list` of single indexing entries `int|long|str`
  for a set of *n*-tuples with *n*>1 - `list` of `list`(s) that contain `int|long|str` tuples

Return: -

`CmplSet``.`**`setValues(`***`startNumber,endNumber`***`)`**

Description: Defines the values of an algorithmic set
`(startNumber, startNumber+1, ..., endNumber)`

Parameter:
- `int` *`startNumber`* — start value of the set
- `int` *`endNumber`* — end value of the set

Return: -

`CmplSet``.`**`setValues(`***`startNumber,step,endNumber`***`)`**

Description: Defines the values of an algorithmic set
`(startNumber, startNumber+step, ..., endNumber)`

Parameter:	`int` *`startNumber`*	start value of the set
	`int` *`step`*	positive value for increment negative value for decrement
	`Int` *`endNumber`*	end value of the set
Return:	-	

R/o attributes:

`CmplSet``.`**`values`**

Description:	List of the indexing entries of the set	
Return:	`list`	of single indexing entries - for a set of *n*-tuples with *n*=1 of `tuple`(s) - for a set of *n*-tuples with *n*>1

`CmplSet``.`**`name`**

Description:	Name of the set	
Return:	`str`	name of the CMPL set (not the name of the `CmplSet` object)

`CmplSet``.`**`rank`**

Description:	Rank of the set	
Return:	`int`	number of *n* of a *n*-tuple set

`CmplSet``.`**`len`**

Description:	Length of the set	
Return:	`int`	number of indexing entries

Examples:

`s = CmplSet("s")` `s.setValues(0,4)` `print( s.rank )` `print( s.len )` `print( s.name )` `print( s.values )`	`s` is assigned $s \in (0,1,\ldots,4)$ `1` `4` `s` `[0, 1, 2, 3, 4]`
`s = CmplSet("a")` `s.setValues(10,-2,0)` `print( s.rank )` `print( s.len )` `print( s.name )` `print( s.values )`	`s` is assigned $s \in (10,8,\ldots,0)$ `1` `6` `s` `[10, 8, 6, 4, 2, 0]`

`s = CmplSet("FOOD")` `s.setValues(["BEEF","CHK","FISH"])` `print( s.rank )` `print( s.len )` `print( s.name )` `print( s.values )`	`s` is assigned $s \in ('BEEF','CHK','FISH')$ `1` `3` `FOOD` `['BEEF', 'CHK', 'FISH']`
`s = CmplSet("c",3)` `s.setValues([ [1,1,1], [1,1,2], \` `  [1,2,1] ])` `print( s.rank )` `print( s.len )` `print( s.name )` `print( s.values )`	`s` is assigned a 3-tuple set of integers `3` `3` `c` `[(1, 1, 1), (1, 1, 2), (1, 2, 1)]`

4.3.2 CmplParameters

The class `CmplParameters` is intended to define parameters that can be used with several `Cmpl` objects.

Methods:

`CmplParameter(`*`paramName`*` [,`***`set1`***`,`***`set2`***`,...])`**

Description: Constructor

Parameter:

`str` *`paramName`*	name of the parameter Has to be equal to the corresponding name in the CMPL model.
`CmplSet` *`set1,set2,...`*	optional - set or sets through which the parameter array is defined (default `None`)

Return: `CmplParameter` object

`CmplParameter``.`**`setValues(`***`val`***`)`**

Description: Defines the values of a scalar parameter

Parameter:

`int|long|float|` `str` *`val`*	value of the scalar parameter

Return: -

`CmplParameter``.`**`setValues(`***`valList`***`)`**

Description: Defines the values of a parameter array

Parameter:

`list` *`valList`*	`list` of `int|long|float|str|list` - value list of the parameter array

Return: -

R/o attributes:

CmplParameter.**values**

Description: List of the values of a parameter

Return: `list` of `int|long|float|str|list | dict` - value list of the parameter array

CmplParameter.**value**

Description: Value of a scalar parameter

Return: `int|long|float|str` - value of the scalar parameter

CmplParameter.**setList**

Description: List of sets through which the parameter array is defined

Return: `list` of `CmplSet` objects through which the parameter array is defined

CmplParameter.**name**

Description: Name of the parameter

Return: `str` - name of the CMPL parameter (not the name of the `CmplParameter` object)

CmplParameter.**rank**

Description: Rank of the parameter

Return: `int` - rank of the CMPL parameter

CmplParameter.**len**

Description: Length of the parameter array

Return: `long` - number of elements in the parameter array

Examples:

`p = CmplParameter("p")` `p.setValues(2)` `print( p.values )` `print( p.value )` `print( p.name )` `print( p.rank )` `print( p.len )`	`p` is assigned `2` `[2]` `2` `p` `0` `1`
`s = CmplSet("s")` `s.setValues(0,4)` `p = CmplParameter("p",s)` `p.setValues([1,2,3,4,5])` `print( p.values )` `print( p.name )` `print( p.rank )` `print( p.len )`	`p` is assigned $(1,2,\ldots,5)$ `[1, 2, 3, 4, 5]` `p` `1` `5`

products = CmplSet("products") products.setValues(1,3) machines = CmplSet("machines") machines.setValues(1,2) a=CmplParameter("a",machines, products) a.setValues([[8,15,12],[15,10,8]]) print(a.values) print(a.name) print(a.rank) print(a.len) for e in a.setList: print(e.values)	a is assigned a 2X3 matrix of integers [[8, 15, 12], [15, 10, 8]] a 2 6 [1, 2] [1, 2, 3]
s = CmplSet("s",2) s.setValues([[1,1],[2,2]]) p = CmplParameter("p",s) p.setValues([1,1]) prin(t p.values) print(p.name) print(p.rank) print(p.len)	s is assigned the indices of a matrix diagonal s is assigned a 2x2 identity matrix [1, 1] p 2 2
combinations = CmplSet("A", 2) combinations.setValues([(1,1),(1,2),\ (1,3),(1,4),(2,1),(2,3),(2,4),(3,1),\ (3,2),(3,3),(3,4)]) costs = {(1,1):13,(1,2):16,(1,3):12, \ (1,4):11, (2,1):15,(2,3):13,(2,4):20, \ (3,1):5,(3,2):7,(3,3):10,(3,4):6} c = CmplParameter("c",combinations) c.setValues(costs)	Creates a CmplSet object and assigns a 2-tuple set to it. Creates a dictionary with keys corresponding to the combinations above and costs as values . The dict costs is assigned as values for the CmplParameter c.

4.3.3 Cmpl

With the Cmpl class it is possible to define a CMPL model, to commit sets and parameters to this model, to start and control the solving process and to read the CMPL and solver messages and to have access to the solution(s) via CmplMessages and CmplSolutions objects.

4.3.3.1 Establishing models

Methods:

`Cmpl(`*`name`***`)`**

Description: Constructor

Parameter: `str` *`name`* filename of the CMPL model

Return: `Cmpl` object

`Cmpl``.`**`setSets(`***`set1[,set2,...]`***`)`**

Description: Committing `CmplSet` objects to the `Cmpl` model

Parameter: `CmplSet` *`set1[,set2,...]`* `CmplSet` object(s)

Return: -

`Cmpl``.`**`setParameters(`***`par1[,par2,...]`***`)`**

Description: Committing `CmplParameter` objects to the `Cmpl` model

Parameter: `CmplParameter` *`par1[,par2,...]`* `CmplParameter` object(s)

Return: -

Examples:

```
m = Cmpl("prodmix.cmpl")

products = CmplSet("products")
products.setValues(1,3)

machines = CmplSet("machines")
machines.setValues(1,2)

c = CmplParameter("c",products)
c.setValues([75,80,50])

b = CmplParameter("b",machines)
b.setValues([1000,1000])

a = CmplParameter("a",machines, products)
a.setValues([[ 8,15,12],[15,10,8]])

m.setSets(products,machines)

m.setParameters(c,a,b)
```

Commits the sets `products, machines` to the Cmpl object `m`

Commits the parameter `c,a,b` to the Cmpl object m

4.3.3.2 Manipulating models

Methods:

Cmpl.**setOption(***option***)**

Description: Sets a CMPL, display or solver option

Parameter: `str` *option* option in CmplHeader syntax

Return: `int` option id

Cmpl.**delOption(***optId***)**

Description: Deletes an option

Parameter: `int` *optId* option id

Return: -

Cmpl.**delOptions()**

Description: Deletes all options

Parameter: -

Return: -

Cmpl.**setOutput(***ok[,leadString]]***)**

Description: Turns the output of CMPL and the invoked solver on or off

Parameter: `bool` *ok* `True|False`

`str` *leadString* optional - Leading string for the output (default - model name)

Return: -

Cmpl.**setRefreshTime(***rTime***)**

Description: Refresh time for getting the output of CMPL and the invoked solver from a CMPLServer if the model is solved synchronously.

Parameter: `float` *rTime* refresh time in seconds (default 0.1)

Return: -

R/o attributes:

Cmpl.**refreshTime**

Description: Returns the refresh time for getting the output of CMPL and the invoked solver from a CMPLServer if the model is solved synchronously.

Return: `float` Refresh time

Examples:

`m = Cmpl("assignment.cmpl")` `c1=m.setOption("-display nonZeros")` `m.setOption("-solver cplex")` `m.setOption("-display solutionPool")` `m.delOption(c1)` `m.delOptions()`	 Setting some options Deletes the first option Deletes all options
`m = Cmpl("assignment.cmpl")` `m.setOutput(True)` `m.setOutput(True,"my special model")`	 The stdOut and stdErr of CMPL and the invoked solver are shown for the `Cmpl` object `m`. As above but the output starts with the leading string `"my special model>"`.
`m = Cmpl("assignment.cmpl")` `m.connect("http://194.95.45.70:8008")` `m.setOutput(True)` `m.setRefreshTime(1)`	 The stdOut and stdErr of CMPL and the invoked solver located at the specified CMPLServer will be refreshed every second.

4.3.3.3 Solving models

Methods:

Cmpl.**solve()**

Description: Solves a `Cmpl` model either with a local installed CMPL or if the model is connected with a CMPLServer remotely.

Parameter: -

Return: - The status of the model and the solver can be obtained by the attributes `cmplStatus`, `cmplStatusText`,`solverStatus` and `solverStatusText`.

Cmpl.**start()**

Description: Solves a `Cmpl` model in a separate thread either with a local installed CMPL or if the model is connected with a CMPLServer remotely.

Parameter: -

Return: - The status of the model and the solver can be obtained by the attributes `cmplStatus`, `cmplStatusText`,`solverStatus` and `solverStatusText`.

Cmpl.**join()**

Description: Waits until the solving thread terminates.

Parameter: -

Return: - The status of the model and the solver can be obtained by the attributes `cmplStatus`, `cmplStatusText`, `solverStatus` and `solverStatusText`.

Cmpl.**isAlive()**

Description: Return whether the thread is alive

Parameter: -

Return: `bool` `True` or `False` - return whether the thread is alive or not

Cmpl.**connect(***cmplUrl***)**

Description: Connects a CMPLServer or CMPLGridScheduler under *`cmplUrl`* – first step of solving a model on a CMPLServer remotely

Parameter: `str` *`cmplUrl`* URL of the CMPLServer or CMPLGridScheduler

Return: – The status of the model can be obtained by the attributes `cmplStatus` and `cmplStatusText`.

Cmpl.**disconnect()**

Description: Disconnects the connected CMPLServer or CMPLGridScheduler

Parameter: –

Return: – The status of the model can be obtained by the attributes `cmplStatus` and `cmplStatusText`.

Cmpl.**send()**

Description: Sends the `Cmpl` model instance to the connected CMPLServer - first step of solving a model on a CMPLServer asynchronously (after connect())

Parameter: –

Return: – The status of the model can be obtained by the attributes `cmplStatus` and `cmplStatusText`.

Cmpl.**knock()**

Description: Knocks on the door of the connected CMPLServer or CMPLGridScheduler and asks whether the model is finished - second step of solving a model on a CMPLServer asynchronously

Parameter: –

Return: – The status of the model can be obtained by the attributes `cmplStatus and cmplStatusText`.

Cmpl.**retrieve()**

Description: Retrieves the `Cmpl` solution(s) if possible from the connected CMPLServer - last step of solving a model on a CMPLServer asynchronously

Parameter: -

Return: - The status of the model and the solver can be obtained by the attributes `cmplStatus`, `cmplStatusText`,`solverStatus` and `solverStatusText`.

Cmpl.**cancel()**

Description: Cancels the `Cmpl` solving process on the connected CMPLServer

Parameter: -

Return: - The status of the model can be obtained by the attributes `cmplStatus` and `cmplStatusText`.

Cmpl.**setMaxServerQueuingTime(***time***)**

Description: Sets the maximum queuing time

Parameter: `float` *time*

Return: -

Cmpl.**setMaxServerTries(***nr***)**

Description: Sets the maximum tries of unsuccessful server calls

Parameter: `int` *nr*

Return: -

R/o attributes:

Cmpl.**cmplStatus**

Description: Returns the CMPL related status of the `Cmpl` object

Return: `int`

```
CMPL_UNKNOWN = 0
CMPL_OK = 1
CMPL_WARNINGS = 2
CMPL_FAILED = 3
CMPLSERVER_OK = 6
CMPLSERVER_ERROR = 7
CMPLSERVER_BUSY = 8
CMPLSERVER_CLEANED = 9
CMPLSERVER_WARNING = 10
PROBLEM_RUNNING = 11
PROBLEM_FINISHED = 12
PROBLEM_CANCELED = 13
PROBLEM_NOTRUNNING = 14
CMPLGRID_SCHEDULER_UNKNOWN = 15
```

```
CMPLGRID_SCHEDULER_OK = 16
CMPLGRID_SCHEDULER_ERROR = 17
CMPLGRID_SCHEDULER_BUSY = 18
CMPLGRID_SCHEDULER_SOLVER_NOT_AVAILABLE = 19
CMPLGRID_SCHEDULER_WARNING = 20
CMPLGRID_SCHEDULER_PROBLEM_DELETED = 21
```

Cmpl.**cmplStatusText**

Description: Returns the CMPL related status text of the `Cmpl` object

Return: `str`

```
CMPL_UNKNOWN
CMPL_OK
CMPL_WARNINGS
CMPL_FAILED
CMPLSERVER_OK
CMPLSERVER_ERROR
CMPLSERVER_BUSY
CMPLSERVER_CLEANED
CMPLSERVER_WARNING
PROBLEM_RUNNING
PROBLEM_FINISHED
PROBLEM_CANCELED
PROBLEM_NOTRUNNING
CMPLGRID_SCHEDULER_UNKNOWN
CMPLGRID_SCHEDULER_OK
CMPLGRID_SCHEDULER_ERROR
CMPLGRID_SCHEDULER_BUSY
CMPLGRID_SCHEDULER_SOLVER_NOT_AVAILABLE
CMPLGRID_SCHEDULER_WARNING
CMPLGRID_SCHEDULER_PROBLEM_DELETED
```

Cmpl.**solverStatus**

Description: Returns the solver related status of the `Cmpl` object

Return: `int`

```
SOLVER_OK = 4
SOLVER_FAILED = 5
```

Cmpl.**solverStatusText**

Description: Returns the solver related status text of the `Cmpl` object

Return: `str`

```
SOLVER_OK
SOLVER_FAILED
```

Cmpl.**jobId**

Description: Returns the jobId of the `Cmpl` problem at the connected CMPLServer

Return: `str` string of the jobId

Cmpl.**maxServerQueuingTime**

Description: Returns the maximum queuing time

Return: `float` Max time

Cmpl.**maxServerTries**

Description: Returns the maximum server tries

Return: `int` Max tries

Examples:

`m = Cmpl("assignment.cmpl")` `m.solve()`	Solves the `Cmpl` object `m` locally
`m = Cmpl("assignment.cmpl")` `m.connect("http://194.95.45.70:8008")` `m.solve()`	Solves the `Cmpl` object `m` remotely and synchronously on the specified CMPLServer
`m = Cmpl("assignment.cmpl")` `m.connect("http://194.95.45.70:8008")` `m.send()` `m.knock()` `m.retrieve()`	Solves the `Cmpl` object `m` remotely and asynchronously on the specified CMPLServer
`models= []` `models.append(Cmpl("m1.cmpl"))` `models.append(Cmpl("m2.cmpl"))` `models.append(Cmpl("m3.cmpl"))` `for m in models:` `  m.start()` `for m in models:` `  m.join()`	Starts all models in separate threads. Waits until the all solving threads are terminated.
`m = Cmpl("assignment.cmpl")` `m.solve()` `if m.solverstatus!=SOLVER_OK:` `  m.solutionReport()`	Displays the optimal solution if the solver didn't fail.

4.3.3.4 Reading solutions

Methods:

Cmpl.**solutionReport()**

Description: Writes a standard solution report to stdOut

Parameter: -

Return: -

Cmpl.**saveSolution(***[solFileName]***)**

Description: Saves the solution(s) as `CmplSolutions` file

Parameter: `str` *solFileName* optional file name (default `<modelname>.csol`)

Return: -

Cmpl.**saveSolutionAscii(***[solFileName]***)**

Description: Saves the solution(s) as ASCII file

Parameter: `str` *solFileName* optional file name (default `<modelname>.sol`)

Return: -

Cmpl.**saveSolutionCsv(***[solFileName]***)**

Description: Saves the solution(s) as CSV file

Parameter: `str` *solFileName* optional file name (default `<modelname>.csv`)

Return: -

Access to variables and constraints

After a problem has been solved and a solution obtained, each variable and constraint can be accessed by its name defined in the Cmpl model as attributes of the `CMPL` object. Each of these newly created attributes returns a `CmplSolution` object.

R/o attributes:

Cmpl.**nrOfVariables**

Description: Returns the number of variables of the generated and solved CMPL model

Return: `int` number of variables

Cmpl.**nrOfConstraints**

Description: Returns the number of constraints of the generated and solved CMPL model

Return: `int` number of constraints

Cmpl.**objectiveName**

Description: Returns the name of the objective function of the generated and solved CMPL model

Return: `str` objective name

Cmpl.**objectiveSense**

Description: Returns the objective sense of the generated and solved CMPL model

Return: `str` objective sense

Cmpl.**nrOfSolutions**

Description: Returns the number of solutions of the generated and solved CMPL model

Return: `int` number of solutions

Cmpl.**solver**

Description: Returns the name of the invoked solver of the generated and solved CMPL model

Return: `str` invoked solver

Cmpl.**solverMessage**

Description: Returns the message of the invoked solver of the generated and solved CMPL model

Return: `str` message of the invoked solver

Cmpl.**varDisplayOptions**

Description: Returns a string with the display options for the variables of the generated and solved CMPL model

Return: `str` display options for the variables

Cmpl.**conDisplayOptions**

Description: Returns a string with the display options for the constraints of the generated and solved CMPL model

Return: `str` display options for the constraints

Cmpl.**solution**

Description: Returns the first (optimal) `CmplSolutions` object

Return: `CmplSolutions` first (optimal) solution

Cmpl.**solutionPool**

Description: Returns a list of `CmplSolutions` objects

Return: `list` of `CmplSolutions` objects — list of `CmplSolution` object for solutions found

CmplSolutions.**status**

Description: Returns a string with the status of the current solution provided by the invoked solver

Return: `str` solution status

CmplSolutions.**value**

Description: Returns the value of the objective function of the current solution

Return: `float` objective function value

CmplSolutions.**idx**

Description: Returns the index of the current solution

Return: `int` index of the current solution

CmplSolutions.**variables**

Description: Returns a list of `CmplSolElement` objects for the variables of the current solution

Return: `list` of *`CmplSol-Line`* objects — list of variables

CmplSolutions.**constraints**

Description: Returns a list of `CmplSolElement` objects for the constraints of the current solution

Return: `list` of `CmplSolElement` objects — list of constraints

`CmplSolElement`.**idx**

Description: Index of the variable or constraint

Return: `int` index of the variable or constraint

`CmplSolElement`.**name**

Description: Name of the variable or constraint

Return: `str` name of the variable or constraint

`CmplSolElement`.**type**

Description: Type of the variable or constraint

Return: *`str`* type of the variable or constraint
`C|I|B` for variables
`L|E|G` for constraints

`CmplSolElement`.**activity**

Description: Activity of the variable or constraint

Return: *`long|float`* activity of the variable or constraint

`CmplSolElement`.**lowerBound**

Description: Lower bound of the variable or constraint

Return: `float` lower bound of the variable or constraint

`CmplSolElement`.**upperBound**

Description: Upper bound of the variable or constraint

Return: `float` upper bound of the variable or constraint

CmplSolElement.**marginal**

Description: Marginal value (shadow prices or reduced costs) bound of the variable or constraint

Return: `float` marginal value of the variable or constraint

Examples:

```
m = Cmpl("assignment.cmpl")
...

m.solve()

print( m.solver )
print( m.solverMessage )
print( m.nrOfVariables )
print( m.nrOfConstraints )

print( m.varDisplayOptions )
print( m.conDisplayOptions    )
print( m.objectiveName )
print( m.objectiveSense )
print( m.solution.value )
print( m.solution.status )
print( m.nrOfSolutions )
print( m.solution.idx )
```

Solves the example from subchapter 4.1 and displays some information about the generated and solved model

```
CBC

11
7

(all)
(all)
costs
min
29.0
optimal
1
0
```

```
for v in m.solution.variables:
  print( v.idx,v.name, v.type, \
    v.activity,v.lowerBound, v.upperBound)
```

Displays all information about variables and constraints of the optimal solution

Variables:

```
0 x[1,1] B 0 0.0 1.0
1 x[1,2] B 0 0.0 1.0
2 x[1,3] B 0 0.0 1.0
3 x[1,4] B 1 0.0 1.0
4 x[2,1] B 0 0.0 1.0
5 x[2,3] B 1 0.0 1.0
6 x[2,4] B 0 0.0 1.0
7 x[3,1] B 1 0.0 1.0
8 x[3,2] B 0 0.0 1.0
9 x[3,3] B 0 0.0 1.0
10 x[3,4] B 0 0.0 1.0
```

```
for c in m.solution.constraints:
  print( c.idx, c.name, c.type, \
    c.activity,c.lowerBound, c.upperBound )
```

Constraints:

```
0 sos_m[1] E 1.0 1.0 1.0
1 sos_m[2] E 1.0 1.0 1.0
```

```
2 sos_m[3] E 1.0 1.0 1.0
3 sos_l[1] L 1.0 -inf 1.0
4 sos_l[2] L 0.0 -inf 1.0
5 sos_l[3] L 1.0 -inf 1.0
6 sos_l[4] L 1.0 -inf 1.0
```

```
m = Cmpl("assignment.cmpl")
...
m.setOption("-display nonZeros")
  m.setOption("-solver cplex")
  m.setOption("-display solutionPool")

  m.setOutput(True)
  m.solve()

  for s in m.solutionPool:
    print("\nSolution number: " , s.idx+1)
    print("Objective value: " , s.value)
    print("Objective status: " , s.status)

    print("Variables:")
    for v in s.variables:
      print("%10s %3s %8i %8i %8i" % \
           ( v.name,v.type,v.activity, \
             v.lowerBound, v.upperBound  ))

    print("Constraints:")
    for c in s.constraints:
      print("%10s %3s %8.0f %8.0f %8.0f" \
           %(c.name,c.type,c.activity,  \
           c.lowerBound,c.upperBound))
```

Solves the example from subchapter 4.1 and displays all information about variables and constraints of all solutions found

```
Solution number:  1
Objective value:  29.0
Objective status:  integer optimal solution
Variables:
    x[1,4]   B        1        0        1
    x[2,3]   B        1        0        1
    x[3,1]   B        1        0        1
Constraints:
  sos_m[1]   E        1        1        1
  sos_m[2]   E        1        1        1
  sos_m[3]   E        1        1        1
  sos_l[1]   L        1     -inf        1
  sos_l[3]   L        1     -inf        1
  sos_l[4]   L        1     -inf        1

Solution number:  2
Objective value:  29.0
Objective status:  integer feasible solution
Variables:
    x[1,4]   B        1        0        1
    x[2,3]   B        1        0        1
    x[3,1]   B        1        0        1
Constraints:
  sos_m[1]   E        1        1        1
  sos_m[2]   E        1        1        1
  sos_m[3]   E        1        1        1
  sos_l[1]   L        1     -inf        1
  sos_l[3]   L        1     -inf        1
  sos_l[4]   L        1     -inf        1

Solution number:  3
Objective value:  33.0
Objective status:  integer feasible solution
Variables:
    x[1,1]   B        1        0        1
    x[2,3]   B        1        0        1
    x[3,2]   B        1        0        1
Constraints:
  sos_m[1]   E        1        1        1
  sos_m[2]   E        1        1        1
  sos_m[3]   E        1        1        1
  sos_l[1]   L        1     -inf        1
```

	sos_l[2] L 1 -inf 1 sos_l[3] L 1 -inf 1
for s in m.solutionPool: print("Variables:" for c in combinations.values: print(m.x[c].name,m.x[c].type, \ m.x[c].activity,\ m.x[c].lowerBound,\ m.x[c].upperBound) print("Constraints:" for i in m.sos_m: print(m.sos_m[i].name,\ m.sos_m[i].type, \ m.sos_m[i].activity,\ m.sos_m[i].lowerBound,\ m.sos_m[i].upperBound) for j in m.sos_l: print(m.sos_l[j].name,\ m.sos_l[j].type,\ m.sos_l[j].activity,\ m.sos_l[j].lowerBound,\ m.sos_l[j].upperBound)	As above but with direct access to the variable and constraint by their names. Iterates the variables `x[i,j]` over the value list of the `CmplSet` object `combinations` Iterates over the internal list of the indexing entries of the constraints with the name `sos_m` Iterates over the internal list of the indexing entries of the constraints with the name `sos_l`

4.3.3.5 Reading CMPL messages

R/o attributes:

Cmpl.**cmplMessages**

Description: Returns a list of `CmplMsg` objects that contain the CMPL messages

Return: `list` of *CmplMsg* objects — list of CMPL messages

CmplMsg.**type**

Description: Returns the type of the messages

Return: `str` — message type `warning|error`

CmplMsg.**module**

Description: Returns the name of the CMPL module in that the error or warning occurs

Return: `str` — CMPL module name

CmplMsg.**location**

Description: Returns the location where the error or warning occurs

Return: `str` location

CmplMsg.**description**

Description: Returns a description of the error or warning message

Return: `str` description of the error or warning

Examples:

```
model = Cmpl("diet.cmpl")
...
model.solve()

if model.cmplStatus==CMPL_WARNINGS:
   for m in model.cmplMessages:
     print( m.type, \
        m.module,\
        m.location, \
        m.description )
```

If some warnings for the CMPL model `diet.cmpl` appear the messages will be shown.

4.3.4 CmplExceptions

pyCMPL provides its own exception handling. If an error occurs either by using pyCmpl classes or in the CMPL model a `CmplException` is raised by pyCmpl automatically. This exception can be handled through using a try-except block.

```
try:
   # do something
except CmplException as e:
   print( e.msg )
```

4.4 jCMPL reference manual

To use the jCMPL functionalities a Java programme has to import jCMPL by `import jCMPL.*;` and to link your application against `jCmpl.jar` and the following jar files, that you can find in the CMPL applicatiopn folder in `jCmpl/lib` or on GitHub (https://github.com/MikeSteglich/jCmpl).

4.4.1 CmplSets

The class `CmplSet` is intended to define sets that can be used with several `Cmpl` objects.

Setter methods:

CmplSet(*setName[,rank]***)**

Description: Constructor

Parameter:

`String` *setName*	name of the set Has to be equal to the corresponding name in the CMPL model.
`int` *rank*	optional - rank *n* for a *n*-tuple set (default 1)

Return: `CmplSet` object

CmplSet.**setValues(***setList***)**

Description: Defines the values of an enumeration set

Parameter:

`Object` *setList*	for a set of *n*-tuples with *n*=1 - `List|Array` of single indexing entries `int|Integer|long|Long|String` for a set of *n*-tuples with *n*>1 – 2-dimensional `List|Array` that contain `int|Integer|long|Long|String` tuples

Return: -

CmplSet.**setValues(***startNumber,endNumber***)**

Description: Defines the values of an algorithmic set
(startNumber, startNumber+1, ..., endNumber)

Parameter:

`int` *startNumber*	start value of the set
`int` *endNumber*	end value of the set

Return: -

CmplSet.**setValues(***startNumber,step,endNumber***)**

Description: Defines the values of an algorithmic set
(startNumber, startNumber+step, ..., endNumber)

Parameter:

`int` *startNumber*	start value of the set
`int` *step*	positive value for increment negative value for decrement
`int` *endNumber*	end value of the set

Return: -

Getter methods:

CmplSet.**values()**

Description: List of the indexing entries of the set

Return:

`List | Array of Object`	one-dimensional `List` or `Array` of single `int|Integer|long|Long|String` - for a set of *n*-tuples with *n*=1 two-dimensional `List` or `Array` of `int|Integer|long|Long|String` - for a set of *n*-tuples with *n*>1

CmplSet.**name()**

Description: Name of the set

Return: `String` name of the CMPL set (not the name of the `CmplSet` object)

CmplSet.**rank()**

Description: Rank of the set

Return: `int` number of *n* of a *n*-tuple set

CmplSet.**len()**

Description: Length of the set

Return: `int` number of indexing entries

Examples:

`CmplSet s = new CmplSet("s");` `s.setValues(0,4);` `System.out.println( s.rank() );` `System.out.println( s.len() );` `System.out.println( s.name() );` `System.out.println( s.values() );`	`s` is assigned $s \in (0,1,\ldots,4)$ `1` `5` `s` `[0, 1, 2, 3, 4]`
`CmplSet s = new CmplSet("a");` `s.setValues(10,-2,0);` `System.out.println( s.rank() );` `System.out.println( s.len() );` `System.out.println( s.name() );` `System.out.println( s.values() );`	`s` is assigned $s \in (10,8,\ldots,0)$ `1` `6` `a` `[10, 8, 6, 4, 2, 0]`
`CmplSet s = new CmplSet("FOOD");` `Sring[] sVals` `  = {"BEEF","CHK","FISH"};` `s.setValues(sVals);` `System.out.println( s.rank() );` `System.out.println( s.len() );` `System.out.println( s.name() );` `for (String e: (String[]) s.values())` `  System.out.println( e);`	`s` is assigned $s \in ('BEEF','CHK','FISH')$ `1` `3` `FOOD` `BEEF` `CHK` `FISH`
`CmplSet s = new CmplSet("FOOD");`	

`ArrayList nutrLst =` `  new ArrayList<String>();` `nutrLst.add("BEEF");` `nutrLst.add("CHK");` `nutrLst.add("FISH");` `s.setValues(nutrLst);` `System.out.println( s.rank() );` `System.out.println( s.len() );` `System.out.println( s.name() );` `System.out.println( s.values());`	`s` is assigned $s \in ('BEEF','CHK','FISH')$ `1` `3` `FOOD` `[BEEF, CHK, FISH]`
`CmplSet s = new CmplSet("c",3);` `int[][] sVals = {  {1,1,1}, {1,1,2},` `  {1,2,1} };` `s.setValues(sVals);` `System.out.println( s.rank() );` `System.out.println( s.len() );` `System.out.println( s.name() );` `for (int i=0; i<s.len(); i++) {` `  for (int j=0; j<s.rank(); j++)` `  System.out.print( s.get(i,j) );` `}`	`s` is assigned a 3-tuple set of integers `3` `3` `c` `111` `112` `121`

4.4.2 CmplParameters

The class `CmplParameters` is intended to define parameters that can be used with several `Cmpl` objects.

Setter methods:

`CmplParameter(`*`paramName`***`"[,`***`set1`***`,`***`set2`***`,...])`**

Description: Constructor

Parameter:

`String` *`paramName`*	name of the parameter Has to be equal to the corresponding name in the CMPL model.
`CmplSet` *`set1,set2,...`*	optional - set or sets through which the parameter array is defined (default `None`)

Return: `CmplParameter` object

CmplParameter.**setValues(***val***)**

Description: Defines the values of a scalar parameter

Parameter: `int|Integer|long|Long|float|Float|double|Double|String` *val* — value of the scalar parameter

Return: -

CmplParameter.**setValues(***vals***)**

Description: Defines the values of a parameter array

Parameter: `Object` *vals* — one- our multidimensional `List|Array` of `int|Integer|long|Long|float|Float|double|Double|String`

Return: -

Getter methods:

CmplParameter.**values()**

Description: List of the values of a parameter

Return: `Object` - one- our multidimensional `List|Array` of `int|Integer|long|Long|float|Float|double|Double|String` - value list of the parameter array

CmplParameter.**value()**

Description: Value of a scalar parameter

Return: `int|Integer|long|Long|float|Float|double|Double|String` - value of the scalar parameter

CmplParameter.**setList()**

Description: List of sets through which the parameter array is defined

Return: `list` of `CmplSet` objects through which the parameter array is defined

CmplParameter.**name()**

Description: Name of the parameter

Return: `String` - name of the CMPL parameter (not the name of the `CmplParameter` object)

CmplParameter.**rank()**

Description: Rank of the parameter

Return: `int` - rank of the CMPL parameter

CmplParameter.**len()**

Description: Length of the parameter array

Return: `long` number of elements in the parameter array

Examples:

`CmplParameter p = new CmplParameter("p");` `p.setValues(2);`	`p` is assigned `2`
`System.out.println( p.values());` `System.out.println( p.value());` `System.out.println( p.name());` `System.out.println( p.rank());` `System.out.println( p.len());`	`2` `2` `p` `0` `1`
`CmplSet s = new CmplSet("s");` `s.setValues(0,4);` `CmplParameter p = new CmplParameter("p",s);` `int[] pVals = { 1,2,3,4,5 };` `p.setValues(pVals);`	`p` is assigned $(1,2,\ldots,5)$
`for ( int val : (int[])p.values() )` `  System.out.println(val);`	`1` `2` `3` `4` `5`
`System.out.println( p.name());` `System.out.println( p.rank());` `System.out.println( p.len());`	`p` `1` `5`
`CmplSet products =  new CmplSet("products");` `products.setValues(1,3);` `CmplSet machines =  new CmplSet("machines");` `machines.setValues(1,2);` `CmplParameter a = new` `  CmplParameter("a",machines,products);` `int[][] aVals = { {8,15,12}, {15,10,8} };` `a.setValues(aVals);`	`a` is assigned a `2X3` matrix of integers
`for ( int i=0; i<machines.len(); i++ ) {` `  for (int j=0; j<products.len(); j++)` `  System.out.print( " " +` `  ((int[][])a.values())[i][j]  );` `  System.out.println();` `}`	`8 15 12` `15 10 8`

`System.out.println( a.name());` `System.out.println( a.rank());` `System.out.println( a.len());` `for (CmplSet s : a.setList())` `  System.out.println( s.values());`	`a` `2` `6` `[1, 2]` `[1, 2, 3]`
`CmplSet s = new CmplSet("s",2);` `int[][] sVals = { {1,1}, {2,2} };` `s.setValues(sVals);` `CmplParameter p = new CmplParameter("p",s);` `int[] pVals = { 1 , 1} ;` `p.setValues(pVals);` `for ( int val : (int[])p.values() )` `  System.out.println(val);` `System.out.println( a.name());` `System.out.println( a.rank());` `System.out.println( a.len());`	 `s` is assigned the indices of a matrix diagonal `s` is assigned a `2x2` identity matrix `1` `1` `p` `2` `2`

4.4.3 Cmpl

With the `Cmpl` class it is possible to define a CMPL model, to commit sets and parameters to this model, to start and control the solving process and to read the CMPL and solver messages and to have access to the solution(s) via `CmplMessages` and `CmplSolutions` objects.

4.4.3.1 Establishing models

Setter methods:

`Cmpl(`*`name`***`)`**

Description: Constructor
Parameter: `String` *`name`* filename of the CMPL model
Return: `Cmpl` object

***`Cmpl`*`.``setSets(`***`set1[,set2,...]`***`)`**

Description: Committing `CmplSet` objects to the `Cmpl` model
Parameter: `CmplSet` *`set1[,set2,...]`* `CmplSet` object(s)
Return: -

Cmpl.**setParameters(***par1[,par2,...]***)**

Description: Committing `CmplParameter` objects to the `Cmpl` model

Parameter: `CmplParameter` *par1[,par2,...]* `CmplParameter` object(s)

Return: -

Examples:

```
Cmpl m = new Cmpl("prodmix.cmpl");

CmplSet products =
  new CmplSet("products");
products.setValues(1,3);

mplSet machines = new CmplSet("machines");
machines.setValues(1,2);

CmplParameter c =
  new CmplParameter("c",products);
int[] cVals = {75,80,50};
c.setValues(cVals);

CmplParameter b =
   new CmplParameter("b",machines);
int[] bVals = {1000,1000};
b.setValues(bVals);

CmplParameter a =
    new CmplParameter("a",machines,products);
int[][] aVals = { {8,15,12}, {15,10,8} };

a.setValues(aVals);

m.setSets(products,machines);

m.setParameters(c,a,b);
```

Commits the sets `products,machines` to the Cmpl object `m`

Commits the parameter `c,a,b` to the Cmpl object m

4.4.3.2 Manipulating models

Setter methods:

Cmpl.**setOption(***option***)**

Description: Sets a CMPL, display or solver option

Parameter:	`String` *option*	option in CmplHeader syntax
Return:	`int`	option id

Cmpl.**delOption(***optId***)**

Description: Deletes an option

Parameter:	`Int` *optId*	option id
Return:	-	

Cmpl.**delOptions()**

Description: Deletes all options

Parameter: -

Return: -

Cmpl.**setOutput(***ok[,leadStr]]***)**

Description: Turns the output of CMPL and the invoked solver on or off

Parameter:	`boolean` *ok*	`true|false`
	`String` *leadStr*	optional - Leading string for the output (default - model name)
Return:	-	

Cmpl.**setRefreshTime(***rTime***)**

Description: Refresh time for getting the output of CMPL and the invoked solver from a CMPLServer if the model is solved synchronously.

Parameter:	`long` *rTime*	refresh time in milliseconds (default 400)
Return:	-	

Getter methods:

Cmpl.**refreshTime()**

Description: Returns the refresh time for getting the output of CMPL and the invoked solver from a CMPLServer if the model is solved synchronously.

Return:	`long`	Refresh time in milliseconds

Examples:

`Cmpl m = new Cmpl("assignment.cmpl");` `long c1=m.setOption("%display nonZeros");` `m.setOption("%arg -solver cplex");` `m.setOption("%display solutionPool");` `m.delOption(c1);` `m.delOptions();`	Setting some options Deletes the first option Deletes all options
`Cmpl m = new Cmpl("assignment.cmpl");` `m.setOutput(True);` `m.setOutput(True,"my special model");`	The stdOut and stdErr of CMPL and the invoked solver are shown for the `Cmpl` object `m`. As above but the output starts with the leading string `"my special model>"`.
`Cmpl m = new Cmpl("assignment.cmpl");` `m.connect("http://194.95.45.70:8008");` `m.setOutput(True);` `m.setRefreshTime(500);`	The `stdOut` and `stdErr` of CMPL and the invoked solver located at the specified CMPLServer will be refreshed every 500 millisecond.

4.4.3.3 Solving models

Setter Methods:

Cmpl.**solve()**

Description: Solves a `Cmpl` model either with a local installed CMPL or if the model is connected with a CMPLServer remotely.

Parameter: -

Return: - status of the model and the solver can be obtained by the methods `cmplStatus`, `cmplStatusText`,`solverStatus` and `solverStatusText`

Cmpl.**start()**

Description: Solves a `Cmpl` model in a separate thread either with a local installed CMPL or if the model is connected with a CMPLServer remotely.

Parameter: -

Return: - status of the model and the solver can be obtained by the methods `cmplStatus`, `cmplStatusText`,`solverStatus` and `solverStatusText`

Cmpl.**join()**

Description: Waits until the solving thread terminates.

Parameter: -

Return: - status of the model and the solver can be obtained by the methods `cmplStatus`, `cmplStatusText`,`solverStatus` and `solverStatusText`

Cmpl.**isAlive()**

Description: Return whether the thread is alive

Parameter: -

Return: `boolean` `true` or `false` - return whether the thread is alive or not

Cmpl.**connect(***cmplUrl***)**

Description: Connects a CMPLServer or CMPLGridScheduler under *`cmplUrl`* – first step of solving a model on a CMPLServer remotely

Parameter: `String` *`cmplUrl`* URL of the CMPLServer or CMPLGridScheduler

Return: –

Cmpl.**disconnect()**

Description: Disconnects the connected CMPLServer or CMPLGridScheduler

Parameter: –

Return: –

Cmpl.**send()**

Description: Sends the `Cmpl` model instance to the connected CMPLServer - first step of solving a model on a CMPLServer asynchronously (after connect())

Parameter: –

Return: – status of the model can be obtained by the methods `cmplStatus` and `cmplStatusText`

Cmpl.**knock()**

Description: Knocks on the door of the connected CMPLServer or CMPLGridScheduler and asks whether the model is finished - second step of solving a model on a CMPLServer asynchronously

Parameter: –

Return: – status of the model can be obtained by the methods `cmplStatus` and `cmplStatusText`

Cmpl.**retrieve()**

Description: Retrieves the `Cmpl` solution(s) if possible from the connected CMPLServer - last step of solving a model on a CMPLServer asynchronously

Parameter: -

Return: - status of the model and the solver can be obtained by the methods `cmplStatus`, `cmplStatusText`,`solverStatus` and `solverStatusText`

Cmpl.**cancel()**

Description: Cancels the `Cmpl` solving process on the connected CMPLServer

Parameter: -

Return: - status of the model can be obtained by the methods `cmplStatus` and `cmplStatusText`

Getter methods:

Cmpl.**cmplStatus()**

Description: Returns the CMPL related status of the `Cmpl` object

Return: `int`

```
CMPL_UNKNOWN = 0
CMPL_OK = 1
CMPL_WARNINGS = 2
CMPL_FAILED = 3
CMPLSERVER_OK = 6
CMPLSERVER_ERROR = 7
CMPLSERVER_BUSY = 8
CMPLSERVER_CLEANED = 9
CMPLSERVER_WARNING = 10
PROBLEM_RUNNING = 11
PROBLEM_FINISHED = 12
PROBLEM_CANCELED = 13
PROBLEM_NOTRUNNING = 14
CMPLGRID_SCHEDULER_UNKNOWN = 15
CMPLGRID_SCHEDULER_OK = 16
CMPLGRID_SCHEDULER_ERROR = 17
CMPLGRID_SCHEDULER_BUSY = 18
CMPLGRID_SCHEDULER_SOLVER_NOT_AVAILABLE = 19
CMPLGRID_SCHEDULER_WARNING = 20
CMPLGRID_SCHEDULER_PROBLEM_DELETED = 21
```

Cmpl.**cmplStatusText()**

Description: Returns the CMPL related status text of the `Cmpl` object

Return: `String`

```
CMPL_UNKNOWN
CMPL_OK
```

```
CMPL_WARNINGS
CMPL_FAILED
CMPLSERVER_OK
CMPLSERVER_ERROR
CMPLSERVER_BUSY
CMPLSERVER_CLEANED
CMPLSERVER_WARNING
PROBLEM_RUNNING
PROBLEM_FINISHED
PROBLEM_CANCELED
PROBLEM_NOTRUNNING
CMPLGRID_SCHEDULER_UNKNOWN
CMPLGRID_SCHEDULER_OK
CMPLGRID_SCHEDULER_ERROR
CMPLGRID_SCHEDULER_BUSY
CMPLGRID_SCHEDULER_SOLVER_NOT_AVAILABLE
CMPLGRID_SCHEDULER_WARNING
CMPLGRID_SCHEDULER_PROBLEM_DELETED
```

Cmpl.**solverStatus()**

Description: Returns the solver related status of the `Cmpl` object

Return: `int`

```
SOLVER_OK = 4
SOLVER_FAILED = 5
```

Cmpl.**solverStatusText()**

Description: Returns the solver related status text of the `Cmpl` object

Return: `String`

```
SOLVER_OK
SOLVER_FAILED
```

Cmpl.**jobId()**

Description: Returns the jobId of the `Cmpl` problem at the connected CMPLServer

Return: `String` string of the jobId

Cmpl.**output()**

Description: Returns the output of CMPL and the invoked solver.
Intended to use if an application needs to parse the output.

Return: `String` string of output of CMPL and the invoked solver

Examples:

`Cmpl m = new Cmpl("assignment.cmpl");` `m.solve();`	Solves the `Cmpl` object `m` locally
`Cmpl m = new Cmpl("assignment.cmpl");` `m.connect("http://127.0.0.1:8008");`	Solves the `Cmpl` object `m` remotely and syn-

`m.solve();`	chronously on the specified CMPLServer
`Cmpl m = new Cmpl("assignment.cmpl");` `m.connect("http://127.0.0.1:8008");` `m.send();` `m.knock();` `m.retrieve();`	Solves the `Cmpl` object `m` remotely and asynchronously on the specified CMPLServer
`ArrayList<Cmpl> models =` `  new ArrayList<Cmpl>();` `models.add(new Cmpl("m1.cmpl"));` `models.add(new Cmpl("m2.cmpl"));` `models.add(new Cmpl("m3.cmpl"));` `for (Cmpl c : models)` `  c.start();` `for (Cmpl c : models)` `  c.join();`	Starts all models in separate threads. Waits until the all solving threads are terminated.
`Cmpl m = new Cmpl("assignment.cmpl");` `m.solve();` `if (m.solverStatus() == Cmpl.SOLVER_OK)` `  m.solutionReport();`	Displays the optimal solution if the solver didn't fail.

4.4.3.4 Reading solutions

Setter methods:

Cmpl.**solutionReport()**

Description: Writes a standard solution report to stdOut

Parameter: -

Return: -

Cmpl.**saveSolution(***[solFileName]***)**

Description: Saves the solution(s) as `CmplSolutions` file

Parameter: `String` *solFileName* optional file name (default `<modelname>.csol`)

Return: -

Cmpl.**saveSolutionAscii(***[solFileName]***)**

Description: Saves the solution(s) as ASCII file

Parameter: `String` *solFileName* optional file name (default `<modelname>.sol`)

Return: -

Cmpl.**saveSolutionCsv(***[solFileName]***)**

Description: Saves the solution(s) as CSV file

Parameter: `String` *solFileName* optional file name (default `<modelname>.csv`)

Return: -

Getter methods:

Cmpl.**nrOfVariables()**

Description: Returns the number of variables of the generated and solved CMPL model

Return: `long` number of variables

Cmpl.**nrOfConstraints()**

Description: Returns the number of constraints of the generated and solved CMPL model

Return: `long` number of constraints

Cmpl.**objectiveName()**

Description: Returns the name of the objective function of the generated and solved CMPL model

Return: `String` objective name

Cmpl.**objectiveSense()**

Description: Returns the objective sense of the generated and solved CMPL model

Return: `String` objective sense

Cmpl.**nrOfSolutions()**

Description: Returns the number of solutions of the generated and solved CMPL model

Return: `int` number of solutions

Cmpl.**solver()**

Description: Returns the name of the invoked solver of the generated and solved CMPL model

Return: `String` invoked solver

Cmpl.**solverMessage()**

Description: Returns the message of the invoked solver of the generated and solved CMPL model

Return: `String` message of the invoked solver

Cmpl.**varDisplayOptions()**

Description: Returns a string with the display options for the variables of the generated and solved CMPL model

Return: `String` display options for the variables

Cmpl.**conDisplayOptions()**

Description: Returns a string with the display options for the constraints of the generated and solved CMPL model

Return: `String` display options for the constraints

Cmpl.**solution()**

Description: Returns the first (optimal) `CmplSolutions` object

Return: `CmplSolutions` first (optimal) solution

Cmpl.**solutionPool()**

Description: Returns a list of `CmplSolutions` objects

Return: `List of CmplSolutions` objects — list of `CmplSolution` object for solutions found

CmplSolutions.**status()**

Description: Returns a string with the status of the current solution provided by the invoked solver

Return: `String` solution status

CmplSolutions.**value()**

Description: Returns the value of the objective function of the current solution

Return: `double` objective function value

CmplSolutions.**idx()**

Description: Returns the index of the current solution

Return: `int` index of the current solution

CmplSolutions.**variables()**

Description: Returns a list of `CmplSolElement` objects for the variables of the current solution

Return: `ArrayList<CmplSolElement>` list of variables

CmplSolutions.**constraints()**

Description: Returns a list of `CmplSolElement` objects for the constraints of the current solution

Return:	`ArrayList<CmplSolElement>`	list of constraints

Cmpl.**getVarByName(***name*, *[solIdx]***)**

Description: Returns a `CmplSolElement` object or `CmplSolArray` of `CmplSolElement` objects for the variable or variable array with the specified name

Parameter:	`String name`	name of the variable or variable array
	`int` *solIdx*	optional solution index (default `0`)
Return:	`Object`	`CmplSolElement` for a single variable
		`CmplSolArray` for a variable array

Cmpl.**getConByName(***[solIdx]***)**

Description: Returns a `CmplSolElement` object or `CmplSolArray` of `CmplSolElement` objects for the constraint or constraint array with the specified name

Parameter:	`String name`	name of the constraint or constraint array
	`int` *solIdx*	optional solution index (default `0`)
Return:	`Object`	`CmplSolElement` for a single constraint
		`CmplSolArray` for a constraint array

`CmplSolElement`.**idx()**

Description: Index of the variable or constraint

Return:	`int`	index of the variable or constraint

`CmplSolElement`.**name()**

Description: Name of the variable or constraint

Return:	`String`	name of the variable or constraint

`CmplSolElement`.**type()**

Description: Type of the variable or constraint

Return:	`String`	type of the variable or constraint
		`C|I|B` for variables
		`L|E|G` for constraints

`CmplSolElement`.**activity()**

Description: Activity of the variable or constraint

Return:	*Object*	*Double|Long* Activity of the variable or constraint

`CmplSolElement.`**`lowerBound()`**

Description: Lower bound of the variable or constraint

Return: `double` lower bound of the variable or constraint

`CmplSolElement.`**`upperBound()`**

Description: Upper bound of the variable or constraint

Return: `double` upper bound of the variable or constraint

`CmplSolElement.`**`marginal()`**

Description: Marginal value (shadow prices or reduced costs) bound of the variable or constraint

Return: `double` marginal value of the variable or constraint

Examples:

```
Cmpl m = new Cmpl("assignment.cmpl");
...

m.solve();

System.out.printf("%s\n",m.solver());
System.out.printf("%s\n",m.solverMessage());
System.out.printf("%d\n",m.nrOfVariables());
System.out.printf("%d\n",m.nrOfConstraints());
System.out.printf("%s\n",m.varDisplayOptions());
System.out.printf("%s\n",m.conDisplayOptions());
System.out.printf("%s\n",m.objectiveName());
System.out.printf("%s\n",m.objectiveSense());
System.out.printf("%f\n",m.solution().value());
System.out.printf("%s\n",m.solution().status());
System.out.printf("%d\n",m.nrOfSolutions());
System.out.printf("%d\n",m.solution().idx());
```

Solves the example from subchapter 4.1 and displays some information about the generated and solved model

```
CBC

11
7
(all)
(all)
costs
min
29.000000
optimal
1
0
```

```
for (CmplSolElement v : m.solution().variables())
{
  System.out.printf("%8s %2s %2d %2.0f %2.0f%n",
     v.name(), v.type(),v.activity(),
     v.lowerBound(),v.upperBound());
}
```

Displays all information about variables and constraints of the optimal solution

Variables:

```
  x[1,1]  B  0  0  1
  x[1,2]  B  0  0  1
  x[1,3]  B  0  0  1
  x[1,4]  B  1  0  1
  x[2,1]  B  0  0  1
  x[2,3]  B  1  0  1
```

	x[2,4] B 0 0 1 x[3,1] B 1 0 1 x[3,2] B 0 0 1 x[3,3] B 0 0 1 x[3,4] B 0 0 1
for (CmplSolElement c:m.solution().constraints()) { System.out.printf("%8s %2s %2.0f %2.0f %2.0f %n", c.name(), c.type(),c.activity(), c.lowerBound(),c.upperBound()); }	Constraints: sos_m[1] E 1 1 1 sos_m[2] E 1 1 1 sos_m[3] E 1 1 1 sos_l[1] L 1 -Infinity 1 sos_l[2] L 0 -Infinity 1 sos_l[3] L 1 -Infinity 1 sos_l[4] L 1 -Infinity 1
CmplSolArray x = (CmplSolArray) m.getVarByName("x"); for(int[] tuple: (int[][]) combinations.values()) { System.out.printf("%5s %2d %n", x.get(tuple).name(), x.get(tuple).activity()); }	Direct access to the variable vector `x[]` by its name
Cmpl m = new Cmpl("assignment.cmpl"); ... m.setOption("%display nonZeros"); m.setOption("%arg -solver cplex"); m.setOption("%display solutionPool"); m.solve(); for (CmplSolution s : m.solutionPool()) { System.out.printf("Solution number: %d %n", (s.idx() + 1)); System.out.printf("Objective value: %f %n", s.value()); System.out.printf("Objective status: %s %n", s.status()); System.out.println("Variables:"); for (CmplSolElement v : s.variables()) { System.out.printf("%8s %2s %2d %2.0f %2.0f %n", v.name(), v.type(), v.activity(), v.lowerBound(), v.upperBound()); } System.out.println("Constraints:"); for (CmplSolElement c : s.constraints()) {	Solves the example from subchapter 4.1 and displays all information about variables and constraints of all solution found Solution number: 1 Objective value: 29.000000 Objective status: integer optimal solution Variables: x[1,4] B 1 0 1 x[2,3] B 1 0 1 x[3,1] B 1 0 1 Constraints: sos_m[1] E 1 1 1 sos_m[2] E 1 1 1

```
    System.out.printf("%8s %2s %2.0f %2.0f %2.0f
      %n", c.name(), c.type(), c.activity(),
      c.lowerBound(), c.upperBound());
  }
}
```

```
sos_m[3]  E  1  1  1
sos_l[1]  L  1 -Infinity  1
sos_l[3]  L  1 -Infinity  1
sos_l[4]  L  1 -Infinity  1
Solution number:  2
Objective value:  29.000000
Objective status:  integer
feasible solution
Variables:
  x[1,4]  B  1  0  1
 ...
```

4.4.3.5 Reading CMPL messages

Getter methods:

Cmpl.**cmplMessages()**

Description: Returns a list of `CmplMsg` objects that contain the CMPL messages

Return: `ArrayList<CmplMsg>` list of CMPL messages

CmplMsg.**type()**

Description: Returns the type of the messages

Return: `String` message type `warning|error`

CmplMsg.**module()**

Description: Returns the name of the CMPL module in that the error or warning occurs

Return: `String` CMPL module

CmplMsg.**location()**

Description: Returns the location where the error or warning occurs

Return: `String` location

CmplMsg.**description()**

Description: Returns the a description of the error or warning message

Return: `String` description of the error or warning

Examples:

```
model = Cmpl("diet.cmpl")
...
model.solve();
```

`if (model.cmplStatus()==Cmpl.CMPL_WARNINGS) {` `  for (CmplMsg m: model.cmplMessages()) {` `  System.out.printf("%s %s %s %s %s",` `  m.type(), m.module(), m.location(),` `  m.description());` `  }` `}`	If some warnings for the CMPL model `diet.cmpl` appear the messages will be shown.

4.4.4 CmplExceptions

jCMPL provides its own exception handling. If an error occurs either by using jCmpl classes or in the CMPL model a `CmplException` is raised by jCmpl automatically. This exception can be handled through using a try-catch block.

```
try {
  // do something
} catch (CmplException e) {
  System.out.println(e);
}
```

4.5 Examples

4.5.1 The diet problem

4.5.1.1 Problem description and CMPL model

In this subchapter the jCMPL and jCMPL formulation of the diet problem already discussed in subchapter 2.4.1.1 is dealt with.

The first step is to formulate the CMPL model `diet.cmpl` where the sets and parameters that are created in the pyCmpl script have to be specified in the CMPL header entry `%data`:

```
%data : NUTR set, FOOD set, costs[FOOD], vitamin[NUTR,FOOD], vitMin[NUTR]

var:
  x[FOOD]: integer[2..10];

obj:
  cost: costs^T * x->min;

con:
  vit: vitamin * x >= vitMin;
```

4.5.1.2 pyCMPL

The corresponding pyCMPL script `diet.py` is formulated as follows:

```
from pyCmpl import *

try:
  model = Cmpl("diet.cmpl")

  nutr = CmplSet("NUTR")
  nutr.setValues(["A", "B1", "B2", "C"])

  food = CmplSet("FOOD")
  food.setValues(["BEEF","CHK","FISH","HAM","MCH","MTL","SPG","TUR"])

  costs = CmplParameter("costs",food)
  costs.setValues([3.19, 2.59, 2.29, 2.89, 1.89, 1.99, 1.99, 2.49])

  vitmin = CmplParameter("vitMin",nutr)
  vitmin.setValues([ 700, 700, 700, 700])

  vitamin = CmplParameter("vitamin",nutr, food)
  vitamin.setValues ( [ [60,8,8,40,15,70,25,60] , [20,0,10,40,35,30,50,20] , \
                      [10,20,15,35,15,15,25,15] ,[15,20,10,10,15,15,15,10]])

  model.setSets(nutr,food)
  model.setParameters(costs,vitmin,vitamin)

  model.solve()
  model.solutionReport()

except CmplException as e:
  print(e.msg)
```

Executing this pyCMPL model by using the command:

```
python diet.py
```

leads to the following output created by pyCMPL's standard solution report:

```
-----------------------------------------------------------------------------------------------------
Problem             diet.cmpl
Nr. of variables    8
Nr. of constraints  4
Objective name      cost
Solver name         CBC
Display variables   (all)
Display constraints (all)
-----------------------------------------------------------------------------------------------------

Objective status    optimal
Objective value     101.14             (min!)
```

```
Variables
Name                 Type           Activity         LowerBound         UpperBound           Marginal
------------------------------------------------------------------------------------------------------
x[BEEF]                 I                  2               2.00              10.00                  -
x[CHK]                  I                  8               2.00              10.00                  -
x[FISH]                 I                  2               2.00              10.00                  -
x[HAM]                  I                  2               2.00              10.00                  -
x[MCH]                  I                 10               2.00              10.00                  -
x[MTL]                  I                 10               2.00              10.00                  -
x[SPG]                  I                 10               2.00              10.00                  -
x[TUR]                  I                  2               2.00              10.00                  -
------------------------------------------------------------------------------------------------------

Constraints
Name                 Type           Activity         LowerBound         UpperBound           Marginal
------------------------------------------------------------------------------------------------------
vit[A]                  G            1500.00             700.00                inf                  -
vit[B1]                 G            1330.00             700.00                inf                  -
vit[B2]                 G             860.00             700.00                inf                  -
vit[C]                  G             700.00             700.00                inf                  -
------------------------------------------------------------------------------------------------------
```

4.5.1.3 jCmpl

The corresponding jCMPL programme `diet.java` is formulated as follows:

```
import jCMPL.*;

public class Diet {

  public static void main(String[] args) throws CmplException {
    try {
      Cmpl model = new Cmpl("diet.cmpl");

      CmplSet nutr = new CmplSet("NUTR");
      String[] nutrLst = {"A", "B1", "B2", "C"};
      nutr.setValues(nutrLst);

      CmplSet food = new CmplSet("FOOD");
      String[] foodLst = {"BEEF", "CHK", "FISH", "HAM", "MCH",
                          "MTL", "SPG", "TUR"};
      food.setValues(foodLst);

      CmplParameter costs = new CmplParameter("costs", food);
      Double[] costVec = {3.19, 2.59, 2.29, 2.89, 1.89, 1.99, 1.99, 2.49};
      costs.setValues(costVec);

      CmplParameter vitmin = new CmplParameter("vitMin", nutr);
      int [] vitminVec = { 700,700,700,700};
      vitmin.setValues(vitminVec);

      CmplParameter vitamin = new CmplParameter("vitamin", nutr, food);
```

```
        int[][] vitMat = {      {60, 8, 8, 40, 15, 70, 25, 60},
                                {20, 0, 10, 40, 35, 30, 50, 20},
                                {10, 20, 15, 35, 15, 15, 25, 15},
                                {15, 20, 10, 10, 15, 15, 15, 10}};
        vitamin.setValues(vitMat);

        model.setSets(nutr, food);
        model.setParameters(costs, vitmin, vitamin);

        model.solve();
        model.solutionReport();

      } catch (CmplException e) {
        System.out.println(e);
      }
    }
}
```

Executing this jCMPL programme leads to the following output created by jCMPL's standard solution report:

```
------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------
Problem              diet.cmpl
Nr. of variables     8
Nr. of constraints   4
Objective name       cost
Solver name          CBC
Display variables    (all)
Display vonstraints  (all)
------------------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      101.14              (min!)

Variables
Name                 Type            Activity         LowerBound         UpperBound            Marginal
------------------------------------------------------------------------------------------------------------
x[BEEF]                 I                2               2.00              10.00                   -
x[CHK]                  I                8               2.00              10.00                   -
x[FISH]                 I                2               2.00              10.00                   -
x[HAM]                  I                2               2.00              10.00                   -
x[MCH]                  I               10               2.00              10.00                   -
x[MTL]                  I               10               2.00              10.00                   -
x[SPG]                  I               10               2.00              10.00                   -
x[TUR]                  I                2               2.00              10.00                   -
------------------------------------------------------------------------------------------------------------

Constraints
Name                 Type            Activity         LowerBound         UpperBound            Marginal
------------------------------------------------------------------------------------------------------------
line[A]                 G             1500.00             700.00           Infinity                   -
line[B1]                G             1330.00             700.00           Infinity                   -
line[B2]                G              860.00             700.00           Infinity                   -
line[C]                 G              700.00             700.00           Infinity                   -
------------------------------------------------------------------------------------------------------------
```

4.5.2 Transportation problem

4.5.2.1 Problem description and CMPL model

This subchapter discusses the pyCMPL formulation of the transportation problem from subchapter 2.4.1.7 .

The CMPL model `transportation.cmpl` can be formulated as follows:

```
%data : plants set, centers set,routes set[2], c[routes], s[plants], d[centers]

var:
  x[routes]: real[0..];
obj:
  costs: sum{ [i,j] in routes : c[i,j]*x[i,j] } ->min;
con:
  {i in plants : supplies[i]: sum{j in routes *> [i,*] : x[i,j]} = s[i];}
  {j in centers: demands[j]: sum{i in routes *> [*,j] : x[i,j]} <= d[j];}
```

4.5.2.2 pyCMPL

The corresponding pyCMPL script `transportation.py` is formulated as follows:

```
from pyCmpl import *

try:
  model = Cmpl("transportation.cmpl")

  routes = CmplSet("routes",2)
  routes.setValues([[1,1],[1,2],[1,4],[2,2],[2,3],[2,4],[3,1],[3,3]])

  plants = CmplSet("plants")
  plants.setValues(1,3)

  centers = CmplSet("centers")
  centers.setValues(1,4)

  costs = CmplParameter("c",routes)
  costs.setValues([3,2,6,5,2,3,2,4])

  s = CmplParameter("s",plants)
  s.setValues([5000,6000,2500])

  d = CmplParameter("d",centers)
  d.setValues([6000,4000,2000,2500])

  model.setSets(routes, plants, centers)
  model.setParameters(costs,s,d)
```

```
  model.setOutput(True)
  model.setOption("-display nonZeros")
  model.solve()

  if model.solverStatus == SOLVER_OK:
    model.solutionReport()
  else:
    print("Solver failed " + model.solver + " " + model.solverMessage)

except CmplException as e:
  print(e.msg)
```

Executing this pyCMPL model by using the command:

```
python transportation.py
```

leads to the following output of CMPL and CBC (enabled with `model.setOutput(True)`) and the standard solution report:

```
transportation> CMPL version: 2.0.0
transportation> Authors: Thomas Schleiff, Mike Steglich
transportation> Distributed under the GPLv3
transportation>
transportation> CMPL: Interpreting Cmpl code
transportation> CMPL: Writing model instance to Free-MPS file > /var/tmp/tmp.0.uR0ye9.mps
transportation> CMPL: Solving instance using CBC
transportation> /Applications/Cmpl2/bin/../Thirdparty/CBC/
transportation> Welcome to the CBC MILP Solver
transportation> Version: devel
transportation> Build Date: Feb 26 2021
transportation>
transportation> command line - /Applications/Cmpl2/bin/../Thirdparty/CBC/_cbc /var/tmp/tmp.0.uR0ye9.mps min solve
gsolu /var/tmp/tmp.0.uR0ye9.sol (default strategy 1)
transportation> At line 2 NAME transportation_cmpl__yxrxd1a6
transportation> At line 5 ROWS
transportation> At line 14 COLUMNS
transportation> At line 31 RHS
transportation> At line 36 BOUNDS
transportation> At line 45 ENDATA
transportation> Problem transportation_cmpl__yxrxd1a6 has 7 rows, 8 columns and 16 elements
transportation> Coin0008I transportation_cmpl__yxrxd1a6 read with 0 errors
transportation> Presolve 6 (-1) rows, 7 (-1) columns and 14 (-2) elements
transportation> Optimal - objective value 36500
transportation> After Postsolve, objective 36500, infeasibilities - dual 0 (0), primal 0 (0)
transportation> Optimal objective 36500 - 5 iterations time 0.002, Presolve 0.00
transportation> Total time (CPU seconds):       0.00   (Wallclock seconds):       0.00
transportation>
transportation>
transportation>
transportation> CMPL: Retrieving solution
transportation> CMPL: Writing solution to CmplSolution file > transportation_cmpl__yxrxd1a6.csol
transportation> CMPL: Writing CmplMessages to file > transportation_cmpl__yxrxd1a6.cmsg
-------------------------------------------------------------------------------------------------------
Problem              transportation.cmpl
Nr. of variables     8
Nr. of constraints   7
Objective name       costs
Solver name          CBC
```

```
Display variables    nonZeroVariables (all)
Display constraints  nonZeroConstraints (all)
-------------------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      36500.00            (min!)

Variables
Name                 Type            Activity         LowerBound         UpperBound            Marginal
-------------------------------------------------------------------------------------------------------------
x[1,1]                  C             2500.00               0.00                inf                0.00
x[1,2]                  C             2500.00               0.00                inf                0.00
x[2,2]                  C             1500.00               0.00                inf                0.00
x[2,3]                  C             2000.00               0.00                inf                0.00
x[2,4]                  C             2500.00               0.00                inf                0.00
x[3,1]                  C             2500.00               0.00                inf                0.00
-------------------------------------------------------------------------------------------------------------

Constraints
Name                 Type            Activity         LowerBound         UpperBound            Marginal
-------------------------------------------------------------------------------------------------------------
supplies[1]             E             5000.00            5000.00            5000.00                3.00
supplies[2]             E             6000.00            6000.00            6000.00                6.00
supplies[3]             E             2500.00            2500.00            2500.00                2.00
demands[1]              L             5000.00               -inf            6000.00                0.00
demands[2]              L             4000.00               -inf            4000.00               -1.00
demands[3]              L             2000.00               -inf            2000.00               -4.00
demands[4]              L             2500.00               -inf            2500.00               -3.00
-------------------------------------------------------------------------------------------------------------
```

4.5.2.3 jCMPL

The corresponding jCMPL script `transportation.java` is formulated as follows:

```
import jCMPL.*;

import java.util.ArrayList;

public class Transportation {

  public static void main(String[] args) throws CmplException {
    try {
      Cmpl model = new Cmpl("transportation.cmpl");

      CmplSet routes = new CmplSet("routes", 2);
      int[][] arcs = { {1, 1}, {1, 2}, {1, 4}, {2, 2}, {2, 3},
                       {2, 4}, {3, 1}, {3, 3}};
      routes.setValues(arcs);

      CmplSet plants = new CmplSet("plants");
      plants.setValues(1, 3);

      CmplSet centers = new CmplSet("centers");
      centers.setValues(1, 1, 4);
```

```
      CmplParameter costs = new CmplParameter("c", routes);
      Integer[] costArr = {3, 2, 6, 5, 2, 3, 2, 4};
      costs.setValues(costArr);

      CmplParameter s = new CmplParameter("s", plants);
      int[] sList = {5000,6000,2500};
      s.setValues(sList);

      CmplParameter d = new CmplParameter("d", centers);
      int[] dArr = {6000, 4000, 2000, 2500};
      d.setValues(dArr);

      model.setSets(routes, plants, centers);
      model.setParameters(costs, s, d);

      model.setOutput(true);
      model.setOption("-display nonZeros");
      model.solve();

      if (model.solverStatus() == Cmpl.SOLVER_OK) {
        model.solutionReport();
      } else {
        System.out.println("Solver failed " + model.solver() +
                           " " + model.solverMessage());
      }

    } catch (CmplException e) {
          System.out.println(e);

    }
  }
}
```

Executing this pyCMPL model by using the command:

```
pyCmpl transportation.py
```

leads to the following output of CMPL and CBC (enabled with `model.setOutput(True)`) and the standard solution report:

```
transportation> CMPL version: 2.0.0
transportation> Authors: Thomas Schleiff, Mike Steglich
transportation> Distributed under the GPLv3
transportation> CMPL: Interpreting Cmpl code
transportation> CMPL: Writing model instance to Free-MPS file > /var/tmp/tmp.0.117IGM.mps
transportation> CMPL: Solving instance using CBC
transportation> /Applications/Cmpl2/bin/../Thirdparty/CBC/
transportation> Welcome to the CBC MILP Solver
transportation> Version: devel
transportation> Build Date: Feb 26 2021
```

```
transportation> command line - /Applications/Cmpl2/bin/../Thirdparty/CBC/_cbc /var/tmp/tmp.0.117IGM.mps min solve
gsolu /var/tmp/tmp.0.117IGM.sol (default strategy 1)
transportation> At line 2 NAME transportation_cmpl__784488
transportation> At line 5 ROWS
transportation> At line 14 COLUMNS
transportation> At line 31 RHS
transportation> At line 36 BOUNDS
transportation> At line 45 ENDATA
transportation> Problem transportation_cmpl__784488 has 7 rows, 8 columns and 16 elements
transportation> Coin0008I transportation_cmpl__784488 read with 0 errors
transportation> Presolve 6 (-1) rows, 7 (-1) columns and 14 (-2) elements
transportation> Optimal - objective value 36500
transportation> After Postsolve, objective 36500, infeasibilities - dual 0 (0), primal 0 (0)
transportation> Optimal objective 36500 - 5 iterations time 0.002, Presolve 0.00
transportation> Total time (CPU seconds):       0.00   (Wallclock seconds):       0.00
transportation> CMPL: Retrieving solution
transportation> CMPL: Writing solution to CmplSolution file > transportation_cmpl__784488.csol
transportation> CMPL: Writing CmplMessages to file > transportation_cmpl__784488.cmsg
------------------------------------------------------------------------------------------------------------
Problem              transportation.cmpl
Nr. of variables     8
Nr. of constraints   7
Objective name       costs
Solver name          CBC
Display variables    nonZeroVariables (all)
Display vonstraints  nonZeroConstraints (all)
------------------------------------------------------------------------------------------------------------

Objective status     optimal
Objective value      36500.00             (min!)

Variables
Name                 Type            Activity         LowerBound         UpperBound            Marginal
------------------------------------------------------------------------------------------------------------
x[1,1]                  C             2500.00               0.00           Infinity                0.00
x[1,2]                  C             2500.00               0.00           Infinity                0.00
x[2,2]                  C             1500.00               0.00           Infinity                0.00
x[2,3]                  C             2000.00               0.00           Infinity                0.00
x[2,4]                  C             2500.00               0.00           Infinity                0.00
x[3,1]                  C             2500.00               0.00           Infinity                0.00
------------------------------------------------------------------------------------------------------------

Constraints
Name                 Type            Activity         LowerBound         UpperBound            Marginal
------------------------------------------------------------------------------------------------------------
supplies[1]             E             5000.00            5000.00            5000.00                3.00
supplies[2]             E             6000.00            6000.00            6000.00                6.00
supplies[3]             E             2500.00            2500.00            2500.00                2.00
demands[1]              L             5000.00          -Infinity            6000.00                0.00
demands[2]              L             4000.00          -Infinity            4000.00               -1.00
demands[3]              L             2000.00          -Infinity            2000.00               -4.00
demands[4]              L             2500.00          -Infinity            2500.00               -3.00
------------------------------------------------------------------------------------------------------------
```

4.5.3 The shortest path problem

4.5.3.1 Problem description and CMPL model

Consider an undirected network $G=(V,A)$ where V is a set of nodes and A is a set of directed edges joining pairs of nodes. The decision is to find the shortest path from a starting node s to a target node t. This problem can be formulated as an LP as follows (Hillier and Liebermann 2010, p. 383f.):

$$\sum_{(i,j)\in A} c_{ij} \cdot x_{ij} \rightarrow min!$$

$$s.t.$$

$$\sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ji} = \begin{cases} 1 & , if\ i=s \\ -1 & , if\ i=t \\ 0 & , otherwise \end{cases} ; \forall i \in V$$

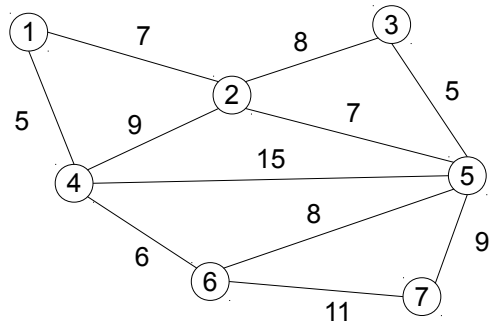
$$x_{ij} \geq 0 ; \forall (i,j) \in A$$

The decision variables are $x_{ij}; \forall \in A$ with $x_{ij}=1$ if the edge $i \rightarrow j$ is used. The parameters $c_{ij}; \forall \in A$ define the distance between the nodes i and j, but can also are interpreted as the time a vehicle takes to drive from node i to node j.

This CMPL model can be formulated as follows whilst the sets A an V and the parameters c_{ij}, t and s are defined in a pyCMPL script or jCMPL programme.

```
%data : A set[2], c[A], V set , s, t

par:
  { i in V: { i=s : rHs[i]:=1; |
       i=t : rHs[i]:=-1; |
       default: rHs[i]:=0;}  }

var:
  x[A] :real[0..];

obj:
  sum{ [i,j] in A: c[i,j]*x[i,j] } -> min;

con:
  { i in V: node[i]: sum{ j in (A *> [i,*]) : x[i,j]  } -
             sum{ j in (A *> [*,i]) : x[j,i]  }  = rHs[i];}
```

To describe the formulation of the shortest path problem in pyCMPL and jCMPL the simple example shown in the following figure is used where the weights on the arcs are interpreted as the time in minutes a vehicle needs to travel from a node i to a node j.



It is assumed that the starting node is node 1 and the target node is node 7.

4.5.3.2 pyCMPL

The corresponding pyCMPL script `shortest-path.py` is formulated as follows:

```
from pyCmpl import *

try:

  model = Cmpl("shortest-path.cmpl")

  routes = CmplSet("A",2)
  routes.setValues([ [1,2],[1,4],[2,1],[2,3],[2,4],[2,5],\
          [3,2],[3,5],[4,1],[4,2],[4,5],[4,6],\
          [5,2],[5,3],[5,4],[5,6],[5,7],\
          [6,4],[6,5],[6,7],[7,5],[7,6] ])

  nodes = CmplSet("V")
  nodes.setValues(1,7)

  c = CmplParameter("c", routes)
  c.setValues([7,5,7,8,9,7,8,5,5,9,15,6,7,5,15,8,9,6,8,11,9,11])

  sNode = CmplParameter("s")
  sNode.setValues(1)

  tNode = CmplParameter("t")
  tNode.setValues(7)

  model.setSets(routes, nodes)
  model.setParameters(c,sNode,tNode)

  model.solve()
  print("Objective Value: ", model.solution.value)
```

```
  for v in model.solution.variables:
    if v.activity>0:
      print(v.name , " " , v.activity)
except CmplException as e:
  print(e.msg)
```

Executing this pyCMPL script through using the command:

```
python shortes-path.py
```

leads to the following output of the pyCMPL script:

```
Objective Value:  22.0
x[1,4]   1.0
x[4,6]   1.0
x[6,7]   1.0
```

The optimal route is 1→4→6→7 with a travelling time of 22 minutes.

4.5.3.3 jCMPL

The corresponding jCMPL programme `shortest-path.java` is formulated as follows:

```
import jCMPL.*;

public class ShortestPath {
    public static void main(String[] args) throws CmplException {

        try {
            Cmpl m = new Cmpl("shortest-path.cmpl");

            CmplSet routes = new CmplSet("A",2);
            int[][] arcs = {{1,2},{1,4},{2,1},{2,3},{2,4},{2,5},
           {3,2},{3,5},{4,1},{4,2},{4,5},{4,6},
           {5,2},{5,3},{5,4},{5,6},{5,7},
           {6,4},{6,5},{6,7},{7,5},{7,6}};
            routes.setValues(arcs);

            CmplSet nodes = new CmplSet("V");
            nodes.setValues(1,7);

            CmplParameter c = new CmplParameter("c", routes);
            Integer[] cArr = {7,5,7,8,9,7,8,5,5,9,15,6,7,5,15,8,9,6,8,11,9,11};
            c.setValues(cArr);

            CmplParameter sNode = new CmplParameter("s");
            sNode.setValues(1);

            CmplParameter tNode = new CmplParameter("t");
```

```
            tNode.setValues(7);

            m.setSets(routes, nodes);
            m.setParameters(c, sNode, tNode);

            m.setOption("-display nonZeros");

            //start CmplServer first with cmplServer -start
            //model.connect("http://127.0.0.1:8008");

            m.solve();

            if (m.solverStatus() == Cmpl.SOLVER_OK) {
                System.out.println("Objective value     :" +
                    m.solution().value() );

                for (CmplSolElement v : m.solution().variables()) {
                    System.out.println( v.name() + " " + v.activity() );
                }

            } else {
                System.out.println("Solver failed " + m.solver() + " " +
                    m.solverMessage());
            }
        } catch (CmplException e) {
            System.out.println(e);
        }
    }
}
```

Executing this jCMPL programme leads to the following output of the pyCMPL script:

```
Objective value     :22.0
x[1,4] 1.0
x[4,6] 1.0
x[6,7] 1.0
```

As in pyCMPL the optimal route is 1→4→6→7 with a travelling time of 22 minutes.

4.5.4 Solving randomized shortest path problems in parallel

4.5.4.1 Problem description

For the last example it was shown that the optimal route travelling from node 1 to node 7 is 1→4→6→7. This solution is based on the assumption that the travelling times between nodes are certain. This example

describes how a randomized shortest path problem can be solved where subproblems describing random situations are solved in own threads in parallel.

4.5.4.2 pyCMPL

Assuming that the starting node is node 1 and the target node is node 7 the corresponding pyCMPL script `shortest-path-threads.py` is formulated as follows:

```
1  from pyCmpl import *
2  import random
3
4  try:
5    routes = CmplSet("A",2)
6    routes.setValues([ [1,2],[1,4],[2,1],[2,3],[2,4],[2,5],\
7            [3,2],[3,5],[4,1],[4,2],[4,5],[4,6],\
8            [5,2],[5,3],[5,4],[5,6],[5,7],\
9            [6,4],[6,5],[6,7],[7,5],[7,6] ])
10
11   nodes = CmplSet("V")
12   nodes.setValues(1,7)
13
14   cList = [7,5,7,8,9,7,8,5,5,9,15,6,7,5,15,8,9,6,8,11,9,11]
15
16   sNode = CmplParameter("s")
17   sNode.setValues(1)
18
19   tNode = CmplParameter("t")
20   tNode.setValues(7)
21
22   models= []
23   randC = []
24   for i in range(5):
25     models.append(Cmpl("shortest-path.cmpl"))
26     models[i].setSets(routes, nodes)
27
28     tmpC =[]
29     for m in cList:
30       tmpC.append( m + random.randint(-40,40)/10)
31
32     randC.append(CmplParameter("c", routes))
33     randC[i].setValues(tmpC)
34
35     models[i].setParameters(randC[i],sNode,tNode)
36
37   for m in models:
38     m.start()
```

```
39
40    for m in models:
41      m.join()
42
43    i = 0
44    for m in models:
45      print("problem : " , i , " needed time " , m.solution.value)
46
47      for v in m.solution.variables:
48        if v.activity>0:
49          print(v.name , " " , v.activity)
50      i = i + 1
51
52  except CmplException as e:
53    print(e.msg)
54  except:
56    print("Unexpected error:", sys.exc_info()[0])
57
```

This script uses the same sets and parameters as before but for each of the five models instantiated in line 25 a new parameter array c is created whilst the original array c is changed by random numbers in line 30. In line 38 all of the models are starting and in line 41 the pyCmpl script is waiting for the termination of all of the models.

Executing this pyCMPL script through using the command:

```
python shortes-path-threads.py
```

can lead to the following output of the pyCMPL script, but every new run will show different results because of the random numbers.

```
problem :  0  needed time  18.2
x[1,4]   1.0
x[4,6]   1.0
x[6,7]   1.0
problem :  1  needed time  17.5
x[1,4]   1.0
x[4,6]   1.0
x[6,7]   1.0
problem :  2  needed time  20.2
x[1,2]   1.0
x[2,5]   1.0
x[5,7]   1.0
problem :  3  needed time  14.6
x[1,2]   1.0
x[2,5]   1.0
x[5,7]   1.0
problem :  4  needed time  19.1
x[1,4]   1.0
```

```
x[4,6]   1.0
x[6,7]   1.0
```

Depending on the uncertain traffic situations two different routes between the nodes 1→7 can be optimal: 1→4→6→7 and 1→2→5→7.

4.5.4.3 jCMPL

Assuming that the staring node is node 1 and the target node is node 7 the corresponding jCMPL programme `shortest-path.java` is formulated as follows:

```
1  import java.util.ArrayList;
2  import jCMPL.*;
3
4  public class ShortestPathThreads {
5      public static void main(String[] args) throws CmplException {
6
7          try {
8               CmplSet routes = new CmplSet("A",2);
9               int[][] arcs = {{1,2},{1,4},{2,1},{2,3},{2,4},{2,5},
10                              {3,2},{3,5},{4,1},{4,2},{4,5},{4,6},
11                              {5,2},{5,3},{5,4},{5,6},{5,7},
12                              {6,4},{6,5},{6,7},{7,5},{7,6}};
13              routes.setValues(arcs);
14
15              CmplSet nodes = new CmplSet("V");
16              nodes.setValues(1,7);
17
18              Integer[] cArr = {7,5,7,8,9,7,8,5,5,9,15,6,7,5,15,8,
19                                9,6,8,11,9,11};
20
21              CmplParameter sNode = new CmplParameter("s");
22              sNode.setValues(1);
23
24              CmplParameter tNode = new CmplParameter("t");
25              tNode.setValues(7);
26
27              ArrayList<Cmpl> models = new ArrayList<Cmpl>();
28              ArrayList<CmplParameter> randC = new ArrayList<CmplParameter>();
29
30              for (int i = 0; i  < 5; i++) {
31                models.add(new Cmpl("shortest-path.cmpl") );
32                models.get(i).setSets(routes, nodes);
33                randC.add(new CmplParameter("c", routes) );
34
35                ArrayList<Double> tmpC = new ArrayList<Double>();
```

```
36                for (Integer cArr1 : cArr) {
37                  tmpC.add(Double.valueOf(cArr1) +
38                       Double.valueOf( -40 +  (Math.random() *  40) )/10);
39              }
40              randC.get(i).setValues(tmpC);
41              models.get(i).setParameters(randC.get(i), sNode, tNode);
42              models.get(i).setOption("-display nonZeros");
43          }
44
45          for (Cmpl c : models) {
46            c.start();
47          }
48
49          for (Cmpl c : models) {
50            c.join();
51          }
52
53          int i = 1;
54          for (Cmpl c : models) {
56            System.out.println("model : " + String.valueOf(i) +
57                 " needed time : " + c.solution().value());
58
59            for (CmplSolElement v : c.solution().variables()) {
60              System.out.println(v.name() + " " + v.activity());
61            }
62            i++;
63          }
64        } catch (CmplException | InterruptedException e) {
65            System.out.println(e);
66        }
67    }
68 }
```

This programme uses the same sets and parameters as before but for each of the five models instantiated in line 31 a new parameter array c is created whilst the original array c is changed by random numbers in lines 37 and 38. In line 46 all of the models are starting and in line 50 the programme is waiting for the termination of all of the models.

Executing this jCMPL programme can lead to the following output, but every new run will show different results because of the random numbers.

```
model : 1 needed time : 17.0396
x[1,4] 1.0
x[4,6] 1.0
x[6,7] 1.0
model : 2 needed time : 15.6087
```

```
x[1,4] 1.0
x[4,6] 1.0
x[6,7] 1.0
model : 3 needed time : 13.0639
x[1,4] 1.0
x[4,6] 1.0
x[6,7] 1.0
model : 4 needed time : 11.4603
x[1,4] 1.0
x[4,6] 1.0
x[6,7] 1.0
model : 5 needed time : 14.3389
x[1,2] 1.0
x[2,5] 1.0
x[5,7] 1.0
```

Depending on the uncertain traffic situations two different routes between 1→7 the nodes can be optimal: 1→4→6→7 and 1→2→5→7 .

4.5.5 Column generation for a cutting stock problem

4.5.5.1 Problem description and CMPL model

The following pyCMPL script and the corresponding jCMPL programme including the example are based on the AMPL formulation of a column generator for a cutting stock problem and is taken from (Fourer et.al. 2003, p. 304ff). In this cutting stock problem long raw rolls of paper have to be cut up into combinations of smaller widths that have to meet given orders and the objective is to minimize the waste.

In the example, the raw width is 110" and the demands for particular widths are given in the following table:

orders (demand)	withs
48	20"
35	45"
24	50"
10	55"
8	75"

Fourer, Gay & Kernigham use the Gilmore-Gomory procedure to define cutting patterns by involving two linear programmes.

The first model is a cutting optimisation model that finds the minimum number of raw rolls with a given set of possible cutting patterns subject to fulfilling the orders for the particular widths. This problem can be formulated as in the CMPL file `cut.cmpl` as follows:

```
%data : rollWidth,widths set,patterns set,orders[widths], nbr[widths, patterns]
```

```
var:
   cut[patterns]: integer[0..];

obj:
  number: sum{ j in patterns: cut[j] }->min;

con:
   {i in widths: fill[i]:
       sum{ j in patterns : nbr[i,j] * cut[j] } >= orders[i];
   }
```

The parameter `rollWidth` defines the width of the raw rolls, the set `widths` defines the widths to be cut, the set `patterns` the set of the patterns, the parameter `orders` the number of orders per width and the parameters `nbr[i,j]` the number of rolls of width `i` in pattern `j`. The variables are the `cut[j]` and they define how many cuts of a pattern `j` are to be produced.

The second model is the pattern generation model that is indented to identify a new pattern that can be used in the cutting optimisation.

```
%data :  widths set, price[widths], rollWidth

var:
  use[widths]: integer[0..];
  reducedCosts : real;

obj:
  sum{ i in widths: price[i] * use[i]} -> max;

con:
  sum{ i in widths : i * use[i] } <= rollWidth;
```

This model in the CMPL file `cut-pattern.cmpl` requires as specified in the `%data` entry the set `widths`, the parameter `rollWidth` and a parameter vector `price`, that contains the marginals of the constraints `fill` of a solved `cut.cmpl` problem with a relaxation of the integer variables `cut[j]`.

It is a knapsack problem that "fills" a knapsack (here a raw roll with a given width `rollWidth`) with the most valuable things (here the desired widths via the variables `use[i]`) where the value of a width `i` is specified by the `price[i]`.

4.5.5.2 pyCMPL

The relationship between these two CMPL models and the entire cutting optimisation procedure is controlled by the following pyCMPL script `cut.py`

```
1   from pyCmpl import *
2   import math
```

```
3
4   try:
5     cuttingOpt = Cmpl("cut.cmpl")
6     patternGen = Cmpl("cut-pattern.cmpl")
7
8     cuttingOpt.setOption("-no-remodel")
9     cuttingOpt.setOption("-solver cplex")
10    patternGen.setOption("-solver cplex")
11
12    r = CmplParameter("rollWidth")
13    r.setValues(110)
14
15    w = CmplSet("widths")
16    w.setValues([ 20, 45, 50, 55, 75])
17
18    o = CmplParameter("orders",w)
19    o.setValues([ 48, 35, 24, 10, 8 ])
20
21    nPat=w.len
22    p = CmplSet("patterns")
23    p.setValues(1,nPat)
24
25    nbr = []
26    for i in range(nPat):
27      nbr.append( [ 0 for j in range(nPat) ] )
28
29    for i in w.values:
30      pos = w.values.index(i)
31      nbr[pos][pos] = int(math.floor( r.value / i ))
32
33    n = CmplParameter("nbr", w, p)
34    n.setValues(nbr)
35
36    price = []
37    for i in range(w.len):
38      price.append(0)
39
40    pr = CmplParameter("price", w)
41    pr.setValues(price)
42
43    cuttingOpt.setSets(w,p)
44    cuttingOpt.setParameters(r, o, n)
45
46    patternGen.setSets(w)
47    patternGen.setParameters(r,pr)
48
```

```
49    ri = cuttingOpt.setOption("-int-relax")
50
51    while True:
52      cuttingOpt.solve()
53
54      for i in w.values:
55        pos = w.values.index(i)
56        price[pos] = cuttingOpt.fill[i].marginal
57
58      pr.setValues(price)
59
60      patternGen.solve()
61
62      if (1-patternGen.solution.value) < -0.00001:
63        nPat = nPat + 1
64        p.setValues(1,nPat)
65        for i in w.values:
66          pos = w.values.index(i)
67          nbr[pos].append(patternGen.use[i].activity)
68        n.setValues(nbr)
69      else:
70        break
71
72    cuttingOpt.delOption(ri)
73
74    cuttingOpt.solve()
75
76    print("Objective value: " , cuttingOpt.solution.value , "\n")
77    print("Pattern:")
78
79    vStr="   | "
80    for j in p.values:
81      vStr+= " %d " % j
82    print(vStr)
83
84    print("----------------------------")
85    for i in range(len(w.values)):
86      vStr="%2d | " % w.values[i]
87      for j in p.values:
88        vStr += " %d " % nbr[i][j-1]
89      print(vStr)
90    print("\n")
91
92    for j in p.values:
93      if cuttingOpt.cut[j].activity>0:
94        print("%2d pieces of pattern: %d" % (cuttingOpt.cut[j].activity, j))
```

```
95          for i in range(len(w.values)):
96            print("   width ", w.values[i] , " - " , nbr[i][j-1])
97
98  except CmplException as e:
99    print(e.msg)
```

In the lines 9 and 10, Cplex is chosen as solver for both models instantiated in the lines 5 and 6. The option `-no-remodel` is needed to prevent some unwanted effects caused by CMPL-internal transformations. In the next lines 12-19 the parameters `rollWidth` and `orders` and the set `widths` are created and the corresponding data are assigned. The lines 25-34 are intended to create an initial set of patterns whilst the matrix `nbr` contains only one pattern per width, where the diagonal elements are equal to the maximal possible number of rolls of the particular width. After creating the vector `price` with null values in the lines 36-41 all relevant sets and parameters are committed to both `Cmpl` objects (lines 43-47).

In the next lines the Gilmore-Gomory procedure is performed.

1. Solving the cutting optimisation problem `cut.cmpl` with an integer relaxation (line 49 and 52).
2. Assigning the shadow prices `cuttingOpt.fill[i].marginal` to the corresponding elements `price[i]` for each pattern (lines 54-56).
3. Solving the pattern generation model `cut-pattern.cmpl` (line 60).
4. If (1 – optimal objective value) is approximately < 0 (line 62)

   then add a new pattern using the activities `patternGen.use[i].activity` for all elements in `widths` (lines 65-67) and jump to step 1,

   else

   Solve the final cutting optimisation problem `cut.cmpl` as integer programme (lines 72 and 74)

After finding the final solution the next lines (lines 76-99) are intended to provide some information about the final integer solution.

Executing this pyCMPL model through using the command:

```
python cut.py
```

leads to the following output of the pyCMPL script:

```
Objective value:  47.0

Pattern:
   |  1  2  3  4  5  6  7  8
---------------------------
20 |  5  0  0  0  0  1  1  3
45 |  0  2  0  0  0  0  2  0
50 |  0  0  2  0  0  0  0  1
55 |  0  0  0  2  0  0  0  0
75 |  0  0  0  0  1  1  0  0
```

```
 8 pieces of pattern: 3
   width  20  -  0
   width  45  -  0
   width  50  -  2
   width  55  -  0
   width  75  -  0
 5 pieces of pattern: 4
   width  20  -  0
   width  45  -  0
   width  50  -  0
   width  55  -  2
   width  75  -  0
 8 pieces of pattern: 6
   width  20  -  1
   width  45  -  0
   width  50  -  0
   width  55  -  0
   width  75  -  1
18 pieces of pattern: 7
   width  20  -  1
   width  45  -  2
   width  50  -  0
   width  55  -  0
   width  75  -  0
 8 pieces of pattern: 8
   width  20  -  3
   width  45  -  0
   width  50  -  1
   width  55  -  0
   width  75  -  0
```

4.5.5.3 jCMPL

The relationship between these `cut-pattern.cmpl` and `cut.cmpl` and the entire cutting optimisation procedure is controlled by the following jCMPL programme `CuttingStock.java`.

```
1   import jCMPL.*;
2   import java.util.ArrayList;
3
4   public class CuttingStock {
5
6       public static void main(String[] args) throws CmplException {
7
8           try {
9               Cmpl cuttingOpt = new Cmpl("cut.cmpl");
```

```
10              Cmpl patternGen = new Cmpl("cut-pattern.cmpl");
11              cuttingOpt.setOption("-no-remodel");
12              cuttingOpt.setOption("-solver cplex");
13              patternGen.setOption("-solver cplex");
14
15              CmplParameter r = new CmplParameter("rollWidth");
16              r.setValues(110);
17
18              CmplSet w = new CmplSet("widths");
19              int[] wVals = {20, 45, 50, 55, 75};
20              w.setValues(wVals);
21
22              CmplParameter o = new CmplParameter("orders", w);
23              int[] oVals = {48, 35, 24, 10, 8};
24              o.setValues(oVals);
25
26              int nPat = w.len();
27
28              CmplSet p = new CmplSet("patterns");
29              p.setValues(1, nPat);
30
31              ArrayList<ArrayList<Long>> nbr = new ArrayList<>();
32
33              for (int i = 0; i < nPat; i++) {
34                  ArrayList<Long> nbrRow = new ArrayList<>();
35                  for (int j = 0; j < nPat; j++) {
36                      if (i == j) {
37                          Double nr = Math.floor(((Integer) r.value()) /
38                                      ((int[]) w.values())[i]);
39                          nbrRow.add(nr.longValue());
40                      } else {
41                          nbrRow.add(Long.valueOf(0));
42                      }
43                  }
44                  nbr.add(nbrRow);
45              }
46
47              CmplParameter n = new CmplParameter("nbr", w, p);
48              n.setValues(nbr);
49
50              Double[] price = new Double[w.len()];
51              for (int i = 0; i < price.length; i++) {
52                  price[i] = 0.0;
53              }
54
55              CmplParameter pr = new CmplParameter("price", w);
```

```
56              pr.setValues(price);
57
58              cuttingOpt.setSets(w, p);
59              cuttingOpt.setParameters(r, o, n);
60
61              patternGen.setSets(w);
62              patternGen.setParameters(r, pr);
63
64              int ri = cuttingOpt.setOption("-int-relax");
65
66              while (true) {
67                  cuttingOpt.solve();
68
69                  CmplSolArray fill =
70                       (CmplSolArray) cuttingOpt.getConByName("fill");
71
72                  int pos = 0;
73                  for (int with : (int[]) w.values()) {
74                      price[pos] = fill.get(with).marginal();
75                      pos++;
76                  }
77
78                  pr.setValues(price);
79
80                  patternGen.solve();
81                  CmplSolArray use =
82                     (CmplSolArray) patternGen.getVarByName("use");
83
84                  if (1 - patternGen.solution().value() < -0.00001) {
85                      nPat++;
86                      p.setValues(1, nPat);
87                      for (int i = 0; i < w.len(); i++) {
88                          ArrayList<Long> tmpList = nbr.get(i);
89                          tmpList.add((Long) use.get(w.get(i)).activity());
90                          nbr.set(i, tmpList);
91                      }
92                      n.setValues(nbr);
93                  } else {
94                      break;
95                  }
96              }
97              cuttingOpt.delOption(ri);
98
99              cuttingOpt.solve();
100             CmplSolArray cut =
101                  (CmplSolArray) cuttingOpt.getVarByName("cut");
```

```
102
103             System.out.printf("Objective value: %4.2f%n%n",
104                               cuttingOpt.solution().value());
105
106             System.out.printf("Pattern:\n");
107             System.out.printf("   | ");
108             for (int j : (ArrayList<Integer>) p.values()) {
109                 System.out.printf(" %d ", j);
110             }
111             System.out.printf("\n----------------------------\n");
112             for (int i = 0; i < w.len(); i++) {
113                 System.out.printf("%2d | ", w.get(i));
114                 for (int j : (ArrayList<Integer>) p.values()) {
115                     System.out.printf(" %d ", nbr.get(i).get(j - 1));
116                 }
117                 System.out.printf("\n");
118             }
119             System.out.printf("\n");
120             for (int j : (ArrayList<Integer>) p.values()) {
121                 if ((Long) cut.get(j).activity() > 0) {
122                     System.out.printf("%2d pieces of pattern: %d %n",
123                             (Long) cut.get(j).activity(), j);
124                     for (int i = 0; i < w.len(); i++) {
125                         System.out.printf("\twidth %d - %d%n",
126                             w.get(i), nbr.get(i).get(j - 1));
127                     }
128                 }
129             }
130
131         } catch (CmplException e) {
132             System.err.println(e);
133         }
134     }
135 }
```

In the lines 12 and 13, Cplex is chosen as solver for both models instantiated in the lines 9 and 10. The option `-no-remodel` is needed to prevent some unwanted effects caused by CMPL-internal transformations. In the next lines 15-24 the parameters `rollWidth` and `orders` and the set `widths` are created and the corresponding data are assigned. The lines 28-45 are intended to create an initial set of patterns whilst the matrix `nbr` contains only one pattern per width, where the diagonal elements are equal to the maximal possible number of rolls of the particular width. After creating the vector `price` with null values in the lines 55-56 all relevant sets and parameters are committed to both `Cmpl` objects (lines 58-62).

In the next lines the Gilmore-Gomory procedure is performed.

5. Solving the cutting optimisation problem `cut.cmpl` with an integer relaxation (line 64 and 67).

6. Assigning the shadow prices `cuttingOpt.fill[i].marginal` to the corresponding elements `price[i]` for each pattern (lines 73-76).
7. Solving the pattern generation model `cut-pattern.cmpl` (line 80).
8. If (1 – optimal objective value) is approximately < 0 (line 84)
   then add a new pattern using the activities `patternGen.use[i].activity` for all elements in `widths` (lines 87-91) and jump to step 1.

   else

   Solve the final cutting optimisation problem `cut.cmpl` as integer programme (line 97 and 99)

After finding the final solution the next lines (lines 100-135) are intended to provide some information about the final integer solution.

Executing this jCMPL model leads to the following output:

```
Objective value: 47.00

Pattern:
   |  1  2  3  4  5  6  7  8
---------------------------
20 |  5  0  0  0  0  1  1  3
45 |  0  2  0  0  0  0  2  0
50 |  0  0  2  0  0  0  0  1
55 |  0  0  0  2  0  0  0  0
75 |  0  0  0  0  1  1  0  0

 8 pieces of pattern: 3
        width 20 - 0
        width 45 - 0
        width 50 - 2
        width 55 - 0
        width 75 - 0
 5 pieces of pattern: 4
        width 20 - 0
        width 45 - 0
        width 50 - 0
        width 55 - 2
        width 75 - 0
 8 pieces of pattern: 6
        width 20 - 1
        width 45 - 0
        width 50 - 0
        width 55 - 0
        width 75 - 1
18 pieces of pattern: 7
        width 20 - 1
```

```
        width 45 - 2
        width 50 - 0
        width 55 - 0
        width 75 - 0
8 pieces of pattern: 8
        width 20 - 3
        width 45 - 0
        width 50 - 1
        width 55 - 0
        width 75 - 0
```

5 Authors and Contact

- **CMPL**

  Thomas Schleiff - Halle(Saale), Germany

  Mike Steglich - Technical University of Applied Sciences Wildau, Germany - mike.steglich@th-wildau.de

- **Coliop, pyCMPL and CMPLServer**

  Mike Steglich

- **jCMPL**

  Mike Steglich

  Bernhard Knie - Technical University of Applied Sciences Wildau, Germany

- **Contact:**

  c/o Mike Steglich

  Professor of Business Administration, Quantitative Methods and Management Accounting

  Technical University of Applied Sciences Wildau
  Faculty of Business, Administration and Law
  Hochschulring 1
  15745 Wildau (Germany)

  Tel.: +493375 / 508-365
  Fax.: +493375 / 508-566

  mike.steglich@th-wildau.de

- **Support via mailing list**

Please use GitHub to get support, to post bugs or to communicate wishes.

https://github.com/MikeSteglich/Cmpl2/issues.

6 Appendix

6.1 Selected CBC parameters

The CBC parameters are taken (mostly unchanged) from the CBC command line help. Only the CBC parameters that are useful in a CMPL context are described afterwards.

Usage CBC parameters:

```
%opt cbc solverOption[=solverOptionValue]
```

Double parameters:

dualB(ound) *doubleValue*

Initially algorithm acts as if no gap between bounds exceeds this value

Range of values is 1e-20 to 1e+12, default 1e+10

dualT(olerance) *doubleValue*

For an optimal solution no dual infeasibility may exceed this value
Range of values is 1e-20 to 1e+12, default 1e-07

objective(Scale) *doubleValue*

Scale factor to apply to objective

Range of values is -1e+20 to 1e+20, default 1

primalT(olerance) *doubleValue*

For an optimal solution no primal infeasibility may exceed this value

Range of values is 1e-20 to 1e+12, default 1e-07

primalW(eight) *doubleValue*

Initially algorithm acts as if it costs this much to be infeasible

Range of values is 1e-20 to 1e+20, default 1e+10

rhs(Scale) *doubleValue*

Scale factor to apply to rhs and bounds

Range of values is -1e+20 to 1e+20, default 1

Branch and Cut double parameters:

allow(ableGap) *doubleValue*

Stop when gap between best possible and best less than this

Range of values is 0 to 1e+20, default 0

artif(icialCost) *doubleValue*

Costs >= these are treated as artificials in feasibility pump 0.0 off - otherwise variables with costs >= these are treated as artificials and fixed to lower bound in feasibility pump

Range of values is 0 to 1.79769e+308, default 0

cuto(ff) *doubleValue*

All solutions must be better than this value (in a minimization sense).

This is also set by code whenever it obtains a solution and is set to value of objective for solution minus cutoff increment.

Range of values is -1e+60 to 1e+60, default 1e+50

fix(OnDj) *doubleValue*

Try heuristic based on fixing variables with reduced costs greater than this

If this is set integer variables with reduced costs greater than this will be fixed before branch and bound - use with extreme caution!

Range of values is -1e+20 to 1e+20, default -1

fraction(forBAB) *doubleValue*

Fraction in feasibility pump

After a pass in feasibility pump, variables which have not moved about are fixed and if the pre-processed model is small enough a few nodes of branch and bound are done on reduced problem. Small problem has to be less than this fraction of original.

Range of values is 1e-05 to 1.1, default 0.5

inc(rement) *doubleValue*

A valid solution must be at least this much better than last integer solution

Whenever a solution is found the bound on solutions is set to solution (in a minimization sense) plus this. If it is not set then the code will try and work one out.

Range of values is -1e+20 to 1e+20, default 1e-05

inf(easibilityWeight) *doubleValue*

Each integer infeasibility is expected to cost this much

Range of values is 0 to 1e+20, default 0

integerT(olerance) *doubleValue*

For an optimal solution no integer variable may be this away from an integer value

Range of values is 1e-20 to 0.5, default 1e-06

preT(olerance) *doubleValue*

Tolerance to use in presolve

Range of values is 1e-20 to 1e+12, default 1e-08

pumpC(utoff) *doubleValue*

Fake cutoff for use in feasibility pump

0.0 off - otherwise add a constraint forcing objective below this value in feasibility pump

Range of values is -1.79769e+308 to 1.79769e+308, default 0

pumpI(ncrement) *doubleValue*

Fake increment for use in feasibility pump

0.0 off - otherwise use as absolute increment to cut off when solution found in feasibility pump

Range of values is -1.79769e+308 to 1.79769e+308, default 0

ratio(Gap) *doubleValue*

If the gap between best solution and best possible solution is less than this fraction of the objective value at the root node then the search will terminate.

Range of values is 0 to 1e+20, default 0

reallyO(bjectiveScale) *doubleValue*

Scale factor to apply to objective in place

Range of values is -1e+20 to 1e+20, default 1

sec(onds) *doubleValue*

maximum seconds

After this many seconds coin solver will act as if maximum nodes had been reached.

Range of values is -1 to 1e+12, default 1e+08

tighten(Factor) *doubleValue*

Tighten bounds using this times largest activity at continuous solution

Range of values is 0.001 to 1e+20, default -1

Integer parameters:

idiot(Crash) *integerValue*

This is a type of 'crash' which works well on some homogeneous problems. It works best on problems with unit elements and rhs but will do something to any model. It should only be used before primal. It can be set to -1 when the code decides for itself whether to use it, 0 to switch off or n > 0 to do n passes.

Range of values is -1 to 99999999, default -1

maxF(actor) *integerValue*

Maximum number of iterations between refactorizations

Range of values is 1 to 999999, default 200

maxIt(erations) *integerValue*

Maximum number of iterations before stopping

Range of values is 0 to 2147483647, default 2147483647

passP(resolve) *integerValue*

How many passes in presolve

Range of values is -200 to 100, default 5

pO(ptions) *integerValue*

If this is > 0 then presolve will give more information and branch and cut will give statistics

Range of values is 0 to 2147483647, default 0

slp(Value) *integerValue*

Number of slp passes before primal

If you are solving a quadratic problem using primal then it may be helpful to do some sequential Lps to get a good approximate solution.

Range of values is -1 to 50000, default -1

slog(Level) *integerValue*

Level of detail in (LP) Solver output

Range of values is -1 to 63, default 1

subs(titution) *integerValue*

How long a column to substitute for in presolve

Normally Presolve gets rid of 'free' variables when there are no more than 3 variables in column. If you increase this the number of rows may decrease but number of elements may increase.

Range of values is 0 to 10000, default 3

Branch and Cut integer parameters:

cutD(epth) *integerValue*

Depth in tree at which to do cuts

Cut generators may be - off, on only at root, on if they look possible and on. If they are done every node then that is that, but it may be worth doing them every so often. The ori-

ginal method was every so many nodes but it is more logical to do it whenever depth in tree is a multiple of K. This option does that and defaults to -1 (off -> code decides).

Range of values is -1 to 999999, default -1

cutL(ength) *integerValue*

Length of a cut

At present this only applies to Gomory cuts. -1 (default) leaves as is. Any value >0 says that all cuts <= this length can be generated both at root node and in tree. 0 says to use some dynamic lengths. If value >=10,000,000 then the length in tree is value%10000000 - so 10000100 means unlimited length at root and 100 in tree.

Range of values is -1 to 2147483647, default -1

dense(Threshold) *integerValue*

Whether to use dense factorization

Range of values is -1 to 10000, default -1

depth(MiniBab) *integerValue*

Depth at which to try mini BAB

Rather a complicated parameter but can be useful. -1 means off for large problems but on as if -12 for problems where rows+columns<500, -2 means use Cplex if it is linked in. Otherwise if negative then go into depth first complete search fast branch and bound when depth>= -value-2 (so -3 will use this at depth>=1). This mode is only switched on after 500 nodes. If you really want to switch it off for small problems then set this to -999. If >=0 the value doesn't matter very much. The code will do approximately 100 nodes of fast branch and bound every now and then at depth>=5. The actual logic is too twisted to describe here.

Range of values is -2147483647 to 2147483647, default -1

diveO(pt) *integerValue*

Diving options

If >2 && <8 then modify diving options

–3 only at root and if no solution,

–4 only at root and if this heuristic has not got solution,

–5 only at depth <4,

–6 decay, 7 run up to 2 times

if solution found 4 otherwise.

Range of values is -1 to 200000, default 3

hOp(tions) *integerValue*

Heuristic options

1 says stop heuristic immediately allowable gap reached. Others are for feasibility pump - 2 says do exact number of passes given, 4 only applies if initial cutoff given and says relax after 50 passes, while 8 will adapt cutoff rhs after first solution if it looks as if code is stalling.

Range of values is -9999999 to 9999999, default 0

hot(StartMaxIts) *integerValue*

Maximum iterations on hot start

Range of values is 0 to 2147483647, default 100

log(Level) *integerValue*

Level of detail in Coin branch and Cut output

If 0 then there should be no output in normal circumstances. 1 is probably the best value for most uses, while 2 and 3 give more information.

Range of values is -63 to 63, default 1

maxN(odes) *integerValue*

Maximum number of nodes to do

Range of values is -1 to 2147483647, default 2147483647

maxS(olutions) *integerValue*

Maximum number of solutions to get

You may want to stop after (say) two solutions or an hour. This is checked every node in tree, so it is possible to get more solutions from heuristics.

Range of values is 1 to 2147483647, default -1

passC(uts) *integerValue*

Number of cut passes at root node

The default is 100 passes if less than 500 columns, 100 passes (but stop if drop small if less than 5000 columns, 20 otherwise

Range of values is -9999999 to 9999999, default -1

passF(easibilityPump) *integerValue*

How many passes in feasibility pump

This fine tunes Feasibility Pump by doing more or fewer passes.

Range of values is 0 to 10000, default 30

passT(reeCuts) *integerValue*

Number of cut passes in tree

Range of values is -9999999 to 9999999, default 1

small(Factorization) *integerValue*

Whether to use small factorization

If processed problem <= this use small factorization

Range of values is -1 to 10000, default -1

strong(Branching) *integerValue*

Number of variables to look at in strong branching

Range of values is 0 to 999999, default 5

thread(s) *integerValue*

Number of threads to try and use

To use multiple threads, set threads to number wanted. It may be better to use one or two more than number of cpus available. If 100+n then n threads and search is repeatable (maybe be somewhat slower), if 200+n use threads for root cuts, 400+n threads used in sub-trees.

Range of values is -100 to 100000, default 0

trust(PseudoCosts) *integerValue*

Number of branches before we trust pseudocosts

Range of values is -3 to 2000000, default 5

Keyword parameters:

bscale *option*

Whether to scale in barrier (and ordering speed)

Possible options: off on off1 on1 off2 on2, default off

chol(esky) *option*

Which cholesky algorithm

Possible options: native dense fudge(Long_dummy) wssmp_dummy

crash *option*

Whether to create basis for problem

If crash is set on and there is an all slack basis then Clp will flip or put structural variables into basis with the aim of getting dual feasible. On the whole dual seems to be better without it and there are alternative types of 'crash' for primal e.g. 'idiot' or 'sprint'.

Possible options: off on so(low_halim) ha(lim_solow(JJF mods)), dfeault off

cross(over) *option*

Whether to get a basic solution after barrier

Interior point algorithms do not obtain a basic solution (and the feasibility criterion is a bit suspect (JJF)). This option will crossover to a basic solution suitable for ranging or branch and cut. With the current state of quadratic it may be a good idea to switch off crossover for quadratic (and maybe presolve as well) - the option maybe does this.

Possible options: on off maybe presolve, default on

dualP(ivot) *option*

Dual pivot choice algorithm

Possible options: auto(matic) dant(zig) partial steep(est), default auto(matic)

fact(orization) *option*

Which factorization to use

Possible options: normal dense simple osl, default normal

gamma((Delta)) *option*

Whether to regularize barrier

Possible options: off on gamma delta onstrong gammastrong deltastrong, default off

KKT *option*

Whether to use KKT factorization

Possible options: off on, default off

perturb(ation) *option*

Whether to perturb problem

Possible options: on off, default on

presolve *option*

Presolve analyzes the model to find such things as redundant equations, equations which fix some variables, equations which can be transformed into bounds etc etc. For the initial solve of any problem this is worth doing unless you know that it will have no effect. on will normally do 5 passes while using 'more' will do 10. If the problem is very large you may need to write the original to file using 'file'.

Possible options for presolve are: on off more file, default on

primalP(ivot) *option*

Primal pivot choice algorithm

Possible options: auto(matic) exa(ct) dant(zig) part(ial) steep(est) change sprint, default auto(matic)

scal(ing) *option*

Whether to scale problem

Possible options: off equi(librium) geo(metric) auto(matic) dynamic rows(only), default auto(matic)

spars(eFactor) *option*

Whether factorization treated as sparse

Possible options: on off, default on

timeM(ode) *option*

Whether to use CPU or elapsed time

cpu uses CPU time for stopping, while elapsed uses elapsed time. (On Windows, elapsed time is always used).

Possible options: cpu elapsed, default cpu

vector *option*

If this parameter is set to on ClpPackedMatrix uses extra column copy in odd format.

Possible options: off on, default off

Branch and Cut keyword parameters:

clique(Cuts) *option*

Whether to use Clique cuts

Possible options: off on root ifmove forceOn onglobal, default ifmove

combine(Solutions) *option*

Whether to use combine solution heuristic

This switches on a heuristic which does branch and cut on the problem given by just using variables which have appeared in one or more solutions. It obviously only tries after two or more solutions. See Rounding for meaning of on,both,before

Possible options: off on both before, default on

combine2(Solutions) *option*

Whether to use crossover solution heuristic

This switches on a heuristic which does branch and cut on the problem given by fixing variables which have same value in two or more solutions. It obviously only tries after two or more solutions. See Rounding for meaning of on,both,before

Possible options: off on both before, default off

cost(Strategy) *option*

How to use costs as priorities

This orders the variables in order of their absolute costs - with largest cost ones being branched on first. This primitive strategy can be surprsingly effective. The column order option is obviously not on costs but easy to code here.

Possible options: off pri(orities) column(Order?) 01f(irst?) 01l(ast?) length(?), default off

cuts(OnOff) *option*

Switches all cuts on or off

This can be used to switch on or off all cuts (apart from Reduce and Split). Then you can do individual ones off or on See branchAndCut for information on options.

Possible options: off on root ifmove forceOn, default on

Dins *option*

This switches on Distance induced neighborhood Search. See Rounding for meaning of on,both,before

Possible options: off on both before often, default off

DivingS(ome) *option*

This switches on a random diving heuristic at various times. C - Coefficient, F - Fractional, G - Guided, L - LineSearch, P - PseudoCost, V - VectorLength. You may prefer to use individual on/off See Rounding for meaning of on,both,before

Possible options: off on both before, default off

DivingC(oefficient) *option*

Whether to try DiveCoefficient

Possible options: off on both before, default on

DivingF(ractional) *option*

Whether to try DiveFractional

Possible options: off on both before, default off

DivingG(uided) *option*

Whether to try DiveGuided

Possible options: off on both before, default off

DivingL(ineSearch) *option*

Whether to try DiveLineSearch

Possible options: off on both before, default off

DivingP(seudoCost) *option*

Whether to try DivePseudoCost

Possible options: off on both before, default off

DivingV(ectorLength) *option*

Whether to try DiveVectorLength

Possible options: off on both before, default off

feas(ibilityPump) *option*

This switches on feasibility pump heuristic at root. This is due to Fischetti, Lodi and Glover and uses a sequence of Lps to try and get an integer feasible solution. Some fine tuning is available by passFeasibilityPump and also pumpTune. See Rounding for meaning of on,both,before

Possible options: off on both before, default on

flow(CoverCuts) *option*

This switches on flow cover cuts (either at root or in entire tree)

See branchAndCut for information on options.

Possible options: off on root ifmove forceOn onglobal, default ifmove

gomory(Cuts) *option*

Whether to use Gomory cuts

The original cuts - beware of imitations! Having gone out of favor, they are now more fashionable as LP solvers are more robust and they interact well with other cuts. They will almost always give cuts (although in this executable they are limited as to number of variables in cut). However the cuts may be dense so it is worth experimenting (Long allows any length). See branchAndCut for information on options.

Possible options: off on root ifmove forceOn onglobal forceandglobal forceLongOn long, default ifmove

greedy(Heuristic) *option*

Whether to use a greedy heuristic

Switches on a greedy heuristic which will try and obtain a solution. It may just fix a percentage of variables and then try a small branch and cut run. See Rounding for meaning of on,both,before

Possible options: off on both before, default on

heur(isticsOnOff) *option*

Switches most heuristics on or off

Possible options: off on, default on

knapsack(Cuts) *option*

This switches on knapsack cuts (either at root or in entire tree)

Possible options: off on root ifmove forceOn onglobal forceandglobal, default ifmove

lift(AndProjectCuts) *option*

Whether to use Lift and Project cuts

Possible options: off on root ifmove forceOn, default off

local(TreeSearch) *option*

This switches on a local search algorithm when a solution is found. This is from Fischetti and Lodi and is not really a heuristic although it can be used as one. When used from Coin solve it has limited functionality. It is not switched on when heuristics are switched on.

Possible options: off on, default off

mixed(IntegerRoundingCuts) *option*

This switches on mixed integer rounding cuts (either at root or in entire tree) See branchAndCut for information on options.

Possible options: off on root ifmove forceOn onglobal, default ifmove

naive(Heuristics) *option*

Really silly stuff e.g. fix all integers with costs to zero!. Do option does heuristic before preprocessing

Possible options: off on both before, default off

node(Strategy) *option*

What strategy to use to select nodes

Normally before a solution the code will choose node with fewest infeasibilities. You can choose depth as the criterion. You can also say if up or down branch must be done first (the up down choice will carry on after solution). Default has now been changed to hybrid which is breadth first on small depth nodes then fewest.

Possible options: hybrid fewest depth upfewest downfewest updepth downdepth, default fewest

pivotAndC(omplement) *option*

Whether to try Pivot and Complement heuristic

Possible options: off on both before, default off

pivotAndF(ix) *option*

Whether to try Pivot and Fix heuristic

Possible options: off on both before, default off

preprocess *option*

This tries to reduce size of model in a similar way to presolve and it also tries to strengthen the model - this can be very useful and is worth trying. Save option saves on file presolved.mps. equal will turn <= cliques into ==. sos will create sos sets if all 0-1 in sets (well one extra is allowed) and no overlaps. trysos is same but allows any number extra. equalall will turn all valid inequalities into equalities with integer slacks.

Possible options: off on save equal sos trysos equalall strategy aggregate forcesos, default sos

probing(Cuts) *option*

This switches on probing cuts (either at root or in entire tree) See branchAndCut for information on options. but strong options do more probing

Possible options: off on root ifmove forceOn onglobal forceonglobal forceOnBut forceOnStrong forceOnButStrong strongRoot, default forceOnStrong

rand(omizedRounding) *option*

Whether to try randomized rounding heuristic

Possible options: off on both before, default off

reduce(AndSplitCuts) *option*

This switches on reduce and split cuts (either at root or in entire tree) See branchAndCut for information on options.

Possible options: off on root ifmove forceOn, default off

residual(CapacityCuts) *option*

Residual capacity cuts. See branchAndCut for information on options.

Possible options: off on root ifmove forceOn, default off

Rens *option*

This switches on Relaxation enforced neighborhood Search. on just does 50 nodes 200 or 1000 does that many nodes. Doh option does heuristic before preprocessing

Possible options: off on both before 200 1000 10000 dj djbefore, default off

Rins *option*

This switches on Relaxed induced neighborhood Search. Doh option does heuristic before preprocessing

Possible options: off on both before often, default on

round(ingHeuristic) *option*

This switches on a simple (but effective) rounding heuristic at each node of tree. On means do in solve i.e. after preprocessing, Before means do if doHeuristics used, off otherwise, and both means do if doHeuristics and in solve.

Possible options: off on both before, default on

two(MirCuts) *option*

This switches on two phase mixed integer rounding cuts (either at root or in entire tree) See branchAndCut for information on options.

Possible options: off on root ifmove forceOn onglobal forceandglobal forceLongOn, default root

Vnd(VariableNeighborhoodSearch) *option*

Whether to try Variable Neighborhood Search

Possible options: off on both before intree, default off

Actions:

barr(ier)	Solve using primal dual predictor corrector algorithm
dualS(implex)	Do dual simplex algorithm
either(Simplex)	Do dual or primal simplex algorithm
initialS	Solve to continuous
	This just solves the problem to continuous - without adding any cuts
outDup	takes duplicate rows etc out of integer model
primalS	Do primal simplex algorithm
reallyS	Scales model in place
stat	Print some statistics
tightLP	Poor person's preSolve for now

Branch and Cut actions:

branch	Do Branch and Cut

6.2 Selected GLPK parameters

The following parameters are taken from the GLPK command line help.

Only the GLPK parameters that are useful in a CMPL context are described afterwards.

Usage GLPK parameters:

```
%opt glpk solverOption[=solverOptionValue]
```

General options:

simplex	use simplex method (default)
interior	use interior point method (LP only)
scale	scale problem (default)
noscale	do not scale problem
ranges *filename*	write sensitivity analysis report to filename in printable format (simplex only)
tmlim *nnn*	limit solution time to nnn seconds
memlim *nnn*	limit available memory to nnn megabytes
wlp *filename*	write problem to filename in CPLEX LP format
wglp *filename*	write problem to filename in GLPK format
wcnf *filename*	write problem to filename in DIMACS CNF-SAT format
log *filename*	write copy of terminal output to filename

LP basis factorization options:

luf	LU + Forrest-Tomlin update (faster, less stable; default)
cbg	LU + Schur complement + Bartels-Golub update (slower, more stable)
cgr	LU + Schur complement + Givens rotation update (slower, more stable)

Options specific to simplex solver:

primal	use primal simplex (default)
dual	use dual simplex
std	use standard initial basis of all slacks
adv	use advanced initial basis (default)
bib	use Bixby's initial basis

steep	use steepest edge technique (default)
nosteep	use standard "textbook" pricing
relax	use Harris' two-pass ratio test (default)
norelax	use standard "textbook" ratio test
presol	use presolver (default; assumes scale and adv)
nopresol	do not use presolver
exact	use simplex method based on exact arithmetic
xcheck	check final basis using exact arithmetic

Options specific to interior-point solver:

nord	use natural (original) ordering
qmd	use quotient minimum degree ordering
amd	use approximate minimum degree ordering (default)
symamd	use approximate minimum degree ordering

Options specific to MIP solver:

nomip	consider all integer variables as continuous (allows solving MIP as pure LP)
first	branch on first integer variable
last	branch on last integer variable
mostf	branch on most fractional variable
drtom	branch using heuristic by Driebeck and Tomlin (default)
pcost	branch using hybrid pseudocost heuristic (may be useful for hard instances)
dfs	backtrack using depth first search
bfs	backtrack using breadth first search
bestp	backtrack using the best projection heuristic
bestb	backtrack using node with best local bound (default)
intopt	use MIP presolver (default)
nointopt	do not use MIP presolver
binarize	replace general integer variables by binary ones (assumes intopt)
fpump	apply feasibility pump heuristic
gomory	generate Gomory's mixed integer cuts
mir	generate MIR (mixed integer rounding) cuts
cover	generate mixed cover cuts

clique	generate clique cuts
cuts	generate all cuts above
mipgap *tol*	set relative mip gap tolerance to tol
minisat	translate integer feasibility problem to CNF-SAT and solve it with MiniSat solver
objbnd *bound*	add inequality obj <= bound (minimization) or obj >= bound (maximization) to integer feasibility problem (assumes minisat)

References

- Achterberg, T. 2009. *SCIP - solving constraint integer programs.* Mathematical Programming Computation Volume 1 Number 1. 1–41.
- Coulouris, G.F.; J. Dollimore, T. Kindberg, G. Blai. 2012. *Distributed Systems : Concepts and Design*, 5th ed., Addison-Wesley.
- Fourer, R., D. M. Gay, B. W. Kernighan. 2003. *AMPL: A Modeling Language for Mathematical Programming*, 2nd ed. Duxbury Press, Pacific Grove, CA.
- Anderson, D. R., D. J. Sweeney, Th. A. Williams, K. Martin. 2011. *An Introduction to Management Science : Quantitative Approaches to Decision Making.* 13th ed.. South-Western.
- Fourer, R, J. Ma, R. K. Martin. 2010. *optimisation Services: A Framework for Distributed optimisation.* Operations Research 58(6). 1624-1636.
- GLPK. 2014. *GNU Linear Programming Kit Reference Manual for GLPK Version 4.54.*
- Hillier, F. S., G. J. Lieberman. 2010. *Introduction to Operations Research.* 9th ed.. McGraw-Hill Higher Education.
- Foster, I., C. Kesselman (editors). 2004. *The Grid2: 2nd Edition: Blueprint for a New Computing Infrastructure*, Kindle ed., Morgan Kaufmann Publishers Inc.
- Kshemkalyani, A.D., M. Singhal, M. 2008. *Distributed Computing – Principles, Algorithms, and Systems*, Kindle ed., Cambridge University Press.
- St. Laurent, S., J. Johnston, E. Dumbill. 2001. *Programming Web Services with XML-RPC*, 1st ed., O'Reilly.