CMPL

<Coliop|Coin> Mathematical Programming Language

Version 1.6.2

March 2012

Manual

M. Steglich, T. Schleiff

Table of content

1 About CMPL

CMPL (<Coliop|Coin> Mathematical Programming Language) is a mathematical programming language and a system for mathematical programming and optimization of linear optimization 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 the COIN-OR OSSolverService, GLPK, Gurobi, SCIP or CPLEX directly to solve the generated model instance. Because it is also possible to transform the mathematical problem into MPS, Free-MPS or OSiL files, 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).

CMPL is a COIN-OR project initiated by the Technical University of Applied Sciences Wildau and the Institute for Operations Research and Business Management at the Martin Luther University Halle-Wittenberg.

For further information please visit the CMPL website (www.coliop.org).

2 CMPL elements

2.1 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. Apart from the variable decision vector x all other components are constant.

$$c^T \cdot x \rightarrow max\,!$$
$$s.t.$$
$$A \cdot x \leq b$$
$$x \geq 0$$

A CMPL model consists of four sections, the `parameters` section, the `variables` section, the `objectives` section and the `constraints` section, which can be inserted several times and mixed in a different order. Each sector can contain one or more lines with user-defined expressions.

```
parameters:
    # definition of the parameters
variables:
    # definition of the variables
objectives:
    # definition of the objective(s)
constraints:
    # 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 contribution 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:

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

can be formulated in CMPL as follows:

```
parameters:
    c[] := ( 1, 2, 3 );
    b[] := ( 15, 20 );

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

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

objectives:
    c[]T * x[] -> max;

constraints:
    A[,] * x[] <= b[];
    x[] >= 0;
```

2.2 Keywords and other syntactic elements

Keywords

`parameters, variables, objectives, constraints`	section markers
`real, integer, binary`	types of variable
`real, integer, binary, string, set`	types only used for type casts
`max, min`	objective senses
`set, in, element, len, defset`	key words for sets
`max, min, count, format, type`	functions for parameter expressions
`sqrt, exp, ln, lg, ld, srand, rand, sin, cos, tan, acos, asin, atan, sinh, cosh, tanh, abs, ceil, floor, round`	mathematical functions that can be used for parameter expressions

`include`	include of CMPL file
`readcsv, readstdin`	data import from a CSV file or from user input
`error, echo`	error and user message
`sum`	summation
`continue, break, default, repeat`	key words for control structures

Arithmetic operators

`+ -`	signs for parameters or addition/subtraction
`^`	to the power of
`* /`	multiplication and division
`div mod`	integer division and remainder on division
`:=`	assignment operator

Condition operators

`= <= >=`	conditions for constraints, while-loops and if-then clauses
`== < > != <>`	additional conditions in while-loops and if-then clauses
`&& || !`	logical operations (and, or, not)

Other syntactic elements

`( )`	- arithmetical bracketing in constant expressions - lists for initialising vectors of constants - parameters for constant functions - increment in an algorithmic set
`[ ]`	- indexing of vectors - range specification in variable definitions
`{ }`	- control structures
`..`	- algorithmic set (e.g. range for indices or loop counters) - range specification in variable definitions
`,`	- element separation in an initialisation list for constant vectors and enumeration sets - separation of function parameters - separation of indices - separation of loop heads in a loop
`:`	- mark indicating beginning of sections - definition of variables - definition of parameter type - separation of loop header from loop body - separation of line names
`|`	- separation of alternative blocks in a control structure

;	- mark indicating end of a statement - every statement is to be closed by a semicolon
# /* */	- comment (up to end of line) - comment (between /* and */)

2.3 Objects

2.3.1 Parameters

A `parameters` section consists of parameter definitions and assignments to parameters. A parameter can only be defined within the `parameters` section using an assignment.

Note that a parameter can be used as a constant in a linear optimization model as coefficients in objectives and constraints. Otherwise parameters can be used like variables in programming languages. Parameters are usable in expressions, for instance in the calculation and definition of other parameters. A user can assign a value to a parameter and can then subsequently change the value with a new assignment.

A parameter is identified by name and, if necessary, by one or more indices. A parameter can be a scalar or an array of parameter values (e.g. vector, matrix or another multidimensional construct). A parameter is defined by an assignment with the assignment operator `:=`. The type of a parameter is defined by the assigned data. Possible types are `real`, `integer`, `binary`, `string` and `set`.

Usage:

```
name := scalarExpression;
name[index] := scalarExpression;
name[[set]] := arrayExpression;
```

name	Name of the parameter
index	A position in an array of parameters The index can be an integer or a string expression. For multidimensional arrays it is necessary to set the index for every dimension separated by commas.
scalarExpression	A scalar parameter or a single part of an array of parameters is assigned a single integer or real number, a single string, the scalar result of a mathematical function.
set	An optional set expression for the definition of the dimension of the array A set is a collection of distinct objects. Distinction can be made between enumeration sets, algorithmic sets and sets which are based on set operations like unions or intersections.

arrayExpression

A non-scalar expression consists of a list of *scalarExpressions* or *arrayExpression*. The elements of the list are separated by commas and imbedded in brackets.

The elements of the list can also be sets. But it is not possible to mix set and non-set expressions.

If an array contains only one element, then it is necessary to include an additional comma behind the element. Otherwise the expression is interpreted as an arithmetical bracket.

Examples:

`k := 10;`	parameter `k` with value 10
`k[1..5] := ( 0.5, 1, 2, 3.3, 5.5);` `k[] := ( 0.5, 1, 2, 3.3, 5.5);`	vector of parameters with 5 elements
`A[]:= (16, 45.4);`	definition of a vector with two integer values `a[1]`=16 and `a[2]`=45.
`a[, ] := (( 5.6, 7.7, 10.5 ),` `( 9.8, 4.2, 11.1 ));`	matrix with 2 rows and 3 columns
`b[] := ( 22 ,);`	definition of the vector `b` with only one element.
`b[] := ( 22 );`	causes an error: `array dimensions don't match`, since `(22)` is not interpreted as an array but as an assignment of a scalar expression.
`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;`	parameter `q` with value 3
`g[1..q] := ( 1, 2, 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`.

If a name is used for a defined parameter, different usages of this name with indices can only refer to parameters, but not to model variables (e.g. if `a[1]` is a parameter, then `a[2]`, `a` or `a[1,1]` can only be defined as parameter and not as model variables. `a` can also not be used as a local parameter like a loop counter).

A special kind of parameter is local parameters, which can only be defined within the head of a control structure. A local parameter is only valid in the body of the control structure and can be used like any other parameter. Only scalar parameters are permitted as local parameters. The main application of local parameters is loop counters iterated over a set.

2.3.2 Variables

The `variables` section is intended to declare the variables of a decision model, which are necessary for the definition of objectives and constraints in the decision model. A model variable is identified by name and, if necessary, by an index. A type must be specified. A model variable can be a scalar or a part of a vector, a matrix or another array of variables. A variable cannot be assigned a value.

Usage:

```
variables:
  name : type [[lowerBound]..[upperBound]];
  name[index] : type [[lowerBound]..[upperBound]];
  name[set] : type [[lowerBound]..[upperBound]];
```

name	name of model variable
type	type of model variable. Possible types are `real, integer, binary`.
[*lowerBound..upperBound*]	optional parameter for limits of model variable *lowerBound* and *upperBound* must be a real or integer expression. For the type `binary` it is not possible to specify bounds.

Examples:

`x: real;`	`x` is a real model variable with no ranges
`x: real[0..100];`	`x` is a real model variable, $0 \le x \le 100$
`x[1..5]: integer[10..20];`	vector with 5 elements, $10 \le x_n \le 20; n = 1(1)5$
`x[1..5,1..5,1..5]: real[0..];`	a three-dimensional array of real model variables with 125 elements identified by indices, $x_{i,j,k} \ge 0; i, j, k = 1(1)5$

`parameters:`
`  prod := set( "bike1", "bike2");`
`variables:`
`  x[prod]: real[0.. ];`
	defines a vector of non-negative real model variables based on the set `prod`

`y: binary;`	`x` is a binary model variable $y \in \{0,1\}$

Different indices may cause model variables to have different types. (e.g. the following is permissible: variables: x[1]: real; x[2]: integer;)

If a name is used for a model variable definition, different usages of this name with indices can only refer to model variables and not to parameters (e.g. if `x[1]` is a model variable, then `x[2]`, `x` or `x[1,1]` can only be defined as model variables).

2.3.3 Indices and sets

Sets are used for the definitions of arrays of parameters or model variables and for the iterations in loops. Indices are necessary to identify an element of an array like a vector or matrix of parameters or model variables. A set is a collection of distinct integer and string elements. Sets can be defined by an enumeration of elements or by algorithms within the `parameters` section. It is also possible to build sets using set operations like condition sets or unions or intersections of defined sets. A set can be stored in a scalar parameter or in an element of an array of parameters.

Usage of set definitions:

```
startNumber(in/decrementor)endNumber          #algorithmic set

[startNumber]..[endNumber]                    #algorithmic set

.integer.                                     #algorithmic set
.string.                                      #algorithmic set

set(listOfIntAndStrings)                      #enumeration set
```

startNumber`(`*in/decrementor*`)`*endNumber*	set of integers based on an algorithm The set starts at the *startNumber*, is changed by an *incrementer* or *decrementer* at every iteration and ends at the *endNumber.*
startNumber`..`*endNumber*	set of integers based on an algorithm The set starts at the *startNumber*, is changed by the number one at every iteration and ends at the *endNumber.* `startNumber` and `endNumber` are optional elements.
startNumber`..`	infinite set with all integers greater than or equal to *startNumber*
`..`*endNumber*	infinite set with all integers less than or equal to *endNumber*
`..`	infinite set with all integers and strings
`.integer.`	infinite set with all integers
`.string.`	infinite set with all strings
listOfIntAndStrings	elements of an enumeration set An enumeration set consists of one or more integer expressions or string expressions separated by commas and embedded in brackets, and is described by the key word `set`. It is possible to define an empty set using an empty array within the statement `set()`.

Examples:

`s:=..;`	`s` is assigned an infinite set of all integers and strings
`s:=..6;`	`s` is assigned $s \in (\ldots, 4, 5, 6)$
`s:=6..;`	`s` is assigned $s \in (6, 7, 8, \ldots)$
`s:=0..6;` `s:=0(1)6;`	`s` is assigned $s \in (0, 1, \ldots, 6)$
`s:=10(-2)4;`	`s` is assigned $s \in (10, 8, 6, 4)$
`prod := set( "bike1", "bike2");`	enumeration set of strings

`a:= set( 1, "a", 3, "b", 5, "c" );`
 	
`x[a]:=(10,20,30,40,50,60);`
 	
`echo x[1];`
`echo x["a"];`
`{i in a: echo x[i];}`
	enumeration set of strings and integers
	
	vector `x` identified by the set `a` is assigned an integer vector
	
	The following user messages are displayed:
	`10`
	`20`
	`10 20 30 40 50 60`

Usage of set operations and set construction:

```
set{ setIteration , condition: localParameter  };        #condition set

set1 + set2;                                             #union set
set1 * set2;                                             #intersection set
```

Usage set operations and set construction:

set1 **+** *set2*	union of *set1* and *set2*
set1 ***** *set2*	intersection of *set1* and *set2*
***set*{** *setIteration*, *condition*: *param* **}**	The local parameter *param* is to be used for the definition of an iteration over a set (defined by *setIteration*) and is to be evaluated in the condition *condition*. The result is a set of all elements which are in the iterated set and fulfil the condition.

Examples:

`s1 := set( "a","b","c","d");`
`s2 := set( "a","e","c","f");`
`s3 := s1 + s2;`
`s4 := s1 * s2;`
	`s3` is assigned `("a","b","c","d","e","f" )`
	`s4` is assigned `( "a", "c" )`

`s5 := set{i in 1..10, i mod 2 = 0: i};`	`s5` is assigned `(2,4,6,8,10)`
`s6 := set{i in s1, !(i element s2): i};`	`s6` is assigned `( "b", "d" )`

2.3.4 Line names

Line names are useful in huge models to provide a better overview of the model. In CMPL a line name can be defined by characters, numbers and the underscore character _ followed by a colon. Names that are used for parameters or model variables cannot be used for a line name. Within a control structure a line name can include the current value of local parameters. This is especially useful for local parameters which are used as a loop counter.

Usage:

```
lineName:

lineName$k$:

lineName$1$:
lineName$2$:

loopName { controlStructure }
```

lineName:	Defines a line name for a single row of the model. If more than one row is to be generated by CMPL, then the line names are extended by numbers in natural order.
$*k*$	k is replaced by the value of the local parameter k
1	1 is replaced by the number of the current line of the matrix.
2	In an implicit loop 2 is replaced by the specific value of the free index.
loopName{*controlStructure*}	Defines a line name subject to the following control structure. The values of loop counters in the control structure are appended automatically.

Examples:

```
parameters:
      A[1..2,1..3] :=((1,2,3),(4,5,6));
      b[1..2] := (100,100);
      c[1..3] := (20,10,10);

variables:
      x[1..3]: real[0..];

objectives:
      profit: c[]T *x[] ->max;
```

generates a line `profit`

```
constraints:
      restriction: A[,] * x[] <=b[];
```

generates 2 lines `restriction_1` `restriction_2`

`{ i:=1(1)2:`
`  restriction_$i$: A[,]*x[]<=b[];`
`}`
	generates 2 lines	`restriction_1`
		`restriction_2`

`restriction { i:=1(1)3:`
`  A[,]*x[]<=b[];`
`  }`
	generates 2 lines	`restriction_1`
		`restriction_2`

`parameters:`
`  products:=set("P1", "P2", "P3");`
`  machines:=set("M1","M2");`
`  A[machines,products] :=((1,2,3),`
`  (4,5,6));`
`  b[machines] := (100,100);`
`  c[products] := (20,10,10);`
`variables:`
`  x[products]: real[0..];`
`objectives:`
`  profit: c[]T *x[] ->max;`
`constraints:`
`  capa_$2$: A[,] * x[] <=b[];`
	generates 3 lines	`profit`
		`capa_M1`
		`capa_M2`

2.4 CMPL header

A CMPL header is intended to define CMPL options, solver options and display options for the specific CMPL model. The elements of the CMPL header are not part of the CMPL model and are processed before the CMPL model is interpreted.

Usage CMPL header:

```
%arg optionName [optionValue]                       #CMPL options
%opt solverName solverOption [solverOptionValue]    #Solver options
%display var|con name[*] [name1[*]] ...             #Display options
```

optionName [*optionValue*]	All CMPL command line arguments excluding a new definition of the input file. Please see subchapter 9.1
solverName	In this version are only solver options for `cbc`, `glpk` and `gurobi` supported.
solverOption [*solverOptionValue*]	Please see to the solver specific parameters subchapter 12 Appendix
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.

Examples:

`%arg -solver glpk`	GLPK is used as the solver
`%arg -solutionAscii`	CMPL writes the optimization results in an ASCII file.

`%arg -solver cbc`
`%arg -solverUrl ↲`
` http://194.95.44.187:8080/ ↲`
` OSServer/services/OSSolverService`
	CBC is to be executed on a OSServer located at 194.95.44.187.

`%opt cbc threads 2`	If CBC is chosen as solver then 2 threads are executed.
`%opt glpk nopresol`	If GLPK is used then the presolver 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.

3 Expressions

3.1 Overview

Expressions are rules for computing a value during the run-time of a CMPL program. Therefore an expression generally cannot include a model variable. Exceptions to this include special functions whose value depends solely on the definition of a certain model variable. Expressions are a part of an assignment to a parameter or are usable within the echo function. Assignments to a parameter are only permitted within the `parameters` section or within a control structure. An expression can be a single number or string, a function or a set. Therefore only real, integer, binary, string or set expressions are possible in CMPL. An expression can contain the normal arithmetic operations.

3.2 Array functions

With the following functions a user may identify specific characteristics of an array or a single parameter or model variable.

Usage:

```
max(expressions)      #returns the numerically largest of a list of values
min(expressions)      #returns the numerically smallest of a list of values

count(parameter|variable|parameterArray|variableArray)
                      #returns the count of elements or 0 if the parameter
                      #or the variable does not exist
```

expressions	can be a list of numerical expressions separated by commas or can be a multi-dimensional array of parameters

Examples:

`a[]:= (1,2,5);`
`echo max( a[] );`
`echo min( a[] );`
	
	returns user message `5`
	returns user message `1`

`echo count( a[] );`	returns user message `3`

`echo count( a[1] );`
`echo count( a[5] );`
`echo count( a[] );`
	returns user message `1`
	returns user message `0`
	returns user message `3`

`b[,]:=((1,2,3,4), (2,3,4,5));`
`echo count(b[1,]);`
`echo count(b[,1]);`
`echo count(b[,]);`
`echo count(b[1,35]);`
	user messages:
	`4` - 4 elements in the first row
	`2` - 2 elements in the first column
	`8` - 4 x 2 elements in the entire matrix
	`0` - parameter does not exist

3.3 Mathematical functions

In CMPL there are the following mathematical functions which can be used in expressions. Excluding `div` and `mod` all these functions return a real value.

Usage:

```
p div q        #integer division
p mod q        #remainder on division
sqrt( x )      #sqrt function
exp( x )       #exp function
ln( x )        #natural logarithm
lg( x )        #common logarithm
ld( x )        #logarithm to the basis 2
srand( x )     #Initialisation of a pseudo-random number generator using the
               argument x. Returns the value of the argument x.
rand( x )      #returns an integer random number in the range 0<= rand <= x
sin( x )       #sine measured in radians
cos( x )       #cosine measured in radians
tan( x )       #tangent measured in radians
acos( x )      #arc cosine measured in radians
asin( x )      #arc sine measured in radians
atan( x )      #arc tangent measured in radians
sinh( x )      #hyperbolic sine
cosh( x )      #hyperbolic cosine
tanh( x )      #hyperbolic tangent
```

```
abs( x )       #absolute value
ceil( x )      #smallest integer value greater than or equal to a given value
floor( x )     #largest integer value less than or equal to a given value
round( x )     #simple round
```

p, q	integer expression
x	real or integer expression

Examples:

```
c[1] := sqrt( 36 );
c[2] := exp(10);
c[3] := ln(10);
c[4] := lg(10000);
c[5] := ld(8);
c[6] := rand(10);
c[7] := sin(2.5);
c[8] := cos(7.7);
c[9] := tan(10.1);
c[10] := acos(0.1);
c[11] := asin(0.4);
c[12] := atan(1.1);
c[13] := sinh(10);
c[14] := cosh(3);
c[15] := tanh(15);
c[16] := abs(-12.55);
c[17] := ceil(12.55);
c[18] := floor(-12.55);
c[19] := round(12.4);
c[20] := 35 div 4;
c[21] := 35 mod 4;
```

```
value is:
6.000000
22026.465795
2.302585
4.000000
3.000000
7.000000    (random number)
0.598472
0.153374
0.800789
1.470629
0.411517
0.832981
11013.232875
10.067662
1.000000
12.550000
13.000000
-13.000000
12.000000
8
3
```

3.4 Type casts

It is useful in some situations to change the type of an expression into another type. A set expression can only be converted to a string. A string can only be converted to a numerical type if it contains a valid numerical string. Every expression can be converted to a string.

Usage:

```
type(expression)          #type cast
```

type	Possible types are: `real, integer, binary, string`.
expression	expression

Examples:

	returns the user messages:

`a := 6.666;`
`echo integer(a);`
`echo binary(a);`
`a:=0;`
`echo binary(a);`
 	
`a := 6.6666;`
`echo string(a);`
	
	`7`
	`1`
	
	`0`
	
	`6.666600`

`b := 100;`
`echo real(b);`
`echo binary(b);`
`b := 0;`
`echo binary(b);`
`b:= 100;`
`echo string(b);`
	
	`100.000000`
	`1`
	
	`0`
	
	`100`

`c :=1;`
`echo real(c);`
`echo integer(c);`
`echo string(c);`
	
	`1.000000`
	`1`
	`1`

`e := "1.888";`
`echo real(e);`
`echo integer(e);`
`echo binary(e);`
`e := "";`
`echo binary(e);`
	
	`1.888000`
	`1`
	`1`
	
	`0`

3.5 String operations

Especially for displaying strings or numbers with the echo function there are string operations to concatenate and format strings.

Usage:

```
expression + expression                 #concat strings if one expression
                                        #has the type string

format( formatString, expression )      #converts a number into a
                                        #string using a format string

len( stringExpression )                 #length of a string

type( expression )                      #returns the type of the expression
                                        #as a string
```

expression	expression which is converted to string Cannot be a set expression. Such an expression must be converted to a string expression by a type cast
formatString	a string expression containing format parameters CMPL uses the format parameters of the programming language C++. For further information please consult a C++ manual.

Usage format parameters:

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

specifier	
d	integer
f	real
s	string
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 type(a)+ " " + a + " to string
" + format("%10.2f", a);
```

returns the user message

```
real 66.777770 to string 66.78
```

If you would like to display an entire set concatenating with a string, then you have to use a string cast of your set.

Example:

`s:= set( 7, "qwe", 6, "fe", 5, 8 );` `echo "set is " + string(s);`	returns the user message `set is set( 7, "qwe", 6, "fe", 5, 8 )`

3.6 Set functions

With the following functions a user can identify the specific characteristics of a set.

Usage:

```
len(set)              #count of the elements of the set - returns an integer

defset(array)         #returns the set of the first free index of the array

element element set   #returns 1 - if the element is an element of the set
                      #returns 0 - otherwise
```

array	array of parameters or model variables with at least one free index.
set	set expression
element	an integer or string that is to be checked

Examples:

`a:= set( 1, "a", 3, "b", 5, "c" );` `echo "length of the set: "+ len(a);`	returns the user message `length of the set: 6`

`A[,] := (  ( 1,2,3,4,5),`
`  (1,2,3,4,5,6,7));`
 	
`row := defset(A[,]);`
`col := defset(A[1,]);`
	
	`row` is assigned the set `1..2`
	`col` is assigned the set `1..5`

`a:= set( 1, "a", 3, "b", 5, "c" );`
`echo "a" element a;`
`echo 5 element a;`
`echo "bb" element a;`
	returns the user message `1`
	returns the user message `1`
	returns the user message `0`

4 Input and output operations

The CMPL input and output operations can be separated into message function, a function that reads the external data and the include statement that reads external CMPL code.

4.1 Error and user messages

Both kinds of message functions display a string as a message. In contrast to the echo function an error message terminates the CMPL program after displaying the message.

Usage:

```
error expression;          #error message - terminates the CMPL program

echo expression;           #user message
```

expression A message that is to be displayed. If the expression is not a string it will be automatically converted to string.

Examples:

`{a<0: error "negative value"; }`	If `a` is negative an error message is displayed and the CMPL program will be terminated.
`echo "constant definitions finished";`	A user message is displayed.

`{ i:=1(1)3: echo "value:" + i;}`	The following user messages are displayed:
	`value: 1`
	`value: 2`
	`value: 3`

4.2 Readcsv and readstdin

CMPL has two functions that enable a user to read external data. The function `readstdin` is designed to read a user's numerical input and assign it to a parameter. The function `readcsv` reads numerical data from a CSV file and assigns it to a vector or matrix of parameters.

Usage:

```
readstdin(message);            #returns a user numerical input

readcsv(fileName);             #reads numerical data from a csv file
                               #for assigning these data to an array
```

message string expression for the message that is to be displayed

fileName string expression for the file name of the CSV file (relative to the directory in which the current CMPL file resides)
In CMPL CSV files that use a comma or semicolon to separate values are permitted.

Example:

`a := readstdin("give me a number");`	reads a value from `stdin` to be used as value for `a`. Only recommended when using CMPL as a command line interpreter.

The following example uses three CSV files:

`1;2;3`	c.csv
`5.6;7.7;10.5` `9.8;4.2;11.1`	a.csv
`15;20`	b.csv

`parameters:`
`  c[] := readcsv("c.csv");`
`  b[] := readcsv("b.csv");`
`  A[,] := readcsv("a.csv");`
`variables:`
`  x[defset(c[])]: real[0..];`
`objectives:`
`  c[]T * x[]->max;`
`constraints:`
`  A[,] * x[] <= b[];`
	Using `readcsv` CMPL generates the following model:
	$1 \cdot x_1 + 2 \cdot x_2 + 3 \cdot x_3 \rightarrow max\,!$
	$s.t.$
	$5.6 \cdot x_1 + 7.7 \cdot x_2 + 10.5 \cdot x_3 \leq 15$
	$9.8 \cdot x_1 + 4.2 \cdot x_2 + 11.1 \cdot x_3 \leq 20$
	$x_j \geq 0\,;\, j = 1\,(1)\,3$

4.3 Include

Using the `include` directive it is possible to read external CMPL code in a CMPL program. The CMPL code in the external CMPL file can be used by several CMPL programs. This makes sense for sharing basic data in a couple of CMPL programs or for the multiple use of specific CMPL statements in several CMPL programs. The `include` directive can stand in any position in a CMPL file. The content of the included file is inserted at this position before parsing the CMPL code. Because `include` is not a statement it is not closed with a semicolon.

Usage:

```
include "fileName"          #include external CMPL code
```

fileName — file name of the CMPL file (relative to the directory in which the current CMPL file resides)

Note that *fileName* can only be a literal string value. It cannot be a string expression or a string parameter.

The following CMPL file "const-def.gen" 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) );`
	const-def.gen

```
parameters:
    include "const-def.cmpl"
variables:
    x[defset(c[])]: real[0..];

objectives:
    c[]T * x[] -> max;
constraints:
    A[,] * x[] <= b[];
```

Using the `include` statement CMPL generates the following model:

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

Using the keyword `include` - it is possible to include the CMPL expressions in file "const-def.cmpl" in another CMPL file.

5 Statements

As mentioned earlier, every CMPL program consists of at least one of the following sections: `parameters:`, `variables:`, `objectives:` and `constraints:`. Each section can be inserted several times and mixed in a different order. Every section can contain special statements.

Every statement finishes with a semicolon.

5.1 `parameters` and `variables` section

Statements in the `parameters` section are assignments to parameters. These assignments define parameters or reassign a new value to already defined parameters. Statements in the `variables` sections are definitions of model variables.

All the syntactic and semantic requirements are described in the chapters above.

5.2 `objectives` and `constraints` section

In the `objectives` and `constraints` sections a user has to define the content of the decision model in linear terms. In general, an objective function of a linear optimization model has the form:

$$c_1 \cdot x_1 + c_2 \cdot x_2 + \ldots + c_n \cdot x_n \ \rightarrow \ max! \quad (\text{or } min!)$$

with the objective function coefficient c_j and model variables x_j . Constraints in general have the form:

$$\begin{aligned}
k_{11} \cdot x_1 + k_{12} \cdot x_2 + \ldots + k_{1n} \cdot x_n \ &\leq \ b_1 \\
k_{21} \cdot x_1 + k_{22} \cdot x_2 + \ldots + k_{2n} \cdot x_n \ &\leq \ b_2 \\
&\vdots \\
k_{m1} \cdot x_1 + k_{m2} \cdot x_2 + \ldots + k_{mn} \cdot x_n \ &\leq \ b_m
\end{aligned}$$

with constraint coefficients k_{ij} and model variables x_j .

An objective or constraint definition in CMPL must use exactly this form or a sum loop that expresses this form. A coefficient can be an arbitrary numerical expression, but the model variables cannot stand in expressions that are different from the general form formulated. The rule that model variables cannot stand in bracketed expressions serves to enforce this.

Please note, it is not permissible to put model variables in brackets!

The example (`a` and `b` are parameters, `x` and `y` model variables)

```
a*x + a*y + b*x + b*y
```

can be written alternatively (with parameters in brackets) as:

```
(a + b)*x + (a + b)*y
```

but not (with model variables in brackets) as:

```
a*(x + y) + b*(x + y)
```

For the definition of the objective sense in the `objectives` section the syntactic elements `->max` or `->min` are used. A line name is permitted and the definition of the objective function has to have a linear form.

Usage of an objective function:

```
objectives:
      [lineName:] linearTerm ->max|->min;
```

lineName	optional element description of objective
linearTerm	definition of linear objective function

The definition of a constraint has to consist of a linear definition of the use of the constraint and one or two relative comparisons. Line names are permitted.

Usage of a constraint:

```
constraints:
      [lineName:] linearTerm <=|>=|= linearTerm  [<=|>=|= linearTerm];
```

lineName	optional element description of objective
linearTerm	linear definition of the left-hand side or the right-hand side of a constraint

6 Control structure

6.1 Overview

A control structure is imbedded in `{ }` and defined by a header followed by a body separated off by `:`.

General usage of a control structure:

```
[controlName]|[sum|set] { controlHeader : controlBody }
```

A control structure can be started with an optional name for the control structure. In the `objectives` and in the `constraints` section this name is also used as the line name.

It is possible to define different kinds of control structures based on different headers, control statements and special syntactical elements. Thus the control structure can used for for loops, while loops, if-then-else clauses and switch clauses. Control structures can be used in all sections.

A control structure can be used for the definition of statements. In this case the control body contains one or more statements which are permissible in this section.

It is also possible to use control structures for `sum` and `set` as expressions. Then the body contains a single expression. A control structure as an expression cannot have a name because this place is taken by the keyword `sum` or `set`. Moreover a control structure as an expression cannot use control statements because the body is an expression and not a statement.

6.2 Control header

A control header consists of one or more control headers. Where there is more than one header, the headers must be separated by commas. Control headers can be divided into iteration headers, condition headers, local assignments and empty headers.

6.2.1 Iteration headers

Iteration headers define how many repeats are to be executed in the control body. Iteration headers are based on sets.

Usage:

```
localParam :=|in set          # iteration over a set
```

localParam	name of the local parameter
set	The defined local parameter iterates over the elements of the set and the body is executed for every element in the set.

Examples:

`s1 := set( "a","b","c","d");` `{k in s1:  ... }`	`k` is iterated over all elements of the set `s1`
`s2 := 1(1)10;` `{k in s2:  ... }`	`k` is iterated in the sequence $k \in \{1,2,\ldots,10\}$
`s3 := 2..6;` `{k := s3:  ... }`	`k` is iterated in the sequence $k \in \{2,3,\ldots,6\}$

6.2.2 Condition headers

A condition returns `1(True)` or `0(False)` subject to the result of a comparison or the properties of a parameter or a set. If the condition returns `1(True)` the body is executed once or else the body is skipped.

Comparison operators for parameters:

`=, ==`	equality
`<>, !=`	inequality
`<`	less than
`>`	greater than
`<=`	equal to or less than
`>=`	equal to or greater than

Comparison operators for sets:

`=`	equality
`==`	tests whether the iteration order of two sets is equal
`<>`	inequality
`!=`	tests whether the iteration order of two sets is not equal
`<`	subset or not equal
`>`	greater than
`<=`	subset or equal
`>=`	equal to or greater than

Logical operators:

`&&`	AND
`||`	OR
`!`	NOT

If a real or integer parameter is assigned 0, the condition returns `0 (false)`. Alternatively if the parameter is assigned `1` the condition returns `1 (true)`.

Examples:

`i:=1;`	
`j:=2;`	
`{i>j : … }`	condition is false
`{!(i>j) : … }`	condition is true
`{!i || j=2 :  … }`	condition is true
`{!i && j=2 : … }`	condition is false (`!i` is false, because `i` is not 0)

6.2.3 Local assignments

A local assignment as control header is useful if a user wishes to make several calculations in a local environment. Assigning expression to a parameter within the `constraints` section is generally not allowed with the exception of a local assignment within a control structure. The body will be executed once.

Usage:

```
localParam := expression            # assignment to a local parameter
```

localParam	Defines a local parameter with this name.
expression	Expression which is assigned to the local parameter.

Examples:

`constraints:` `  { k:=1 :  … }`	`k` is assigned `1` and used as local parameter within the control structure.

6.3 Alternative bodies

If a control header consists of at least one condition, it is possible to define alternative bodies. Structures like that make sense if a user wishes to combine a for loop with an if-then clause.

The first defined body after the headers is the main body of the control structure. Subsequent bodies must be separated by the syntactic element `|`. Alternative bodies are only executed if the main body is skipped.

Usage:

```
{ controlHeader: mainBody [ | condition1: alternativeBody1 ]
      [ | ... ] [ | default: alternativeDefaultBody ] }
```

controlHeader	header of the control structure including at least one condition The alternative bodies belong to last header of control header. This header cannot be an assignment of a local parameter, because in this case the main body is never skipped.
mainBody	main body of control structure
condition1	Will be evaluated if alternative body is executed.
alternativeBody1	The first alternative body with a condition that evaluates to true is executed. The remaining alternative bodies are skipped without checking the conditions.
alternativeDefaultBody	If no condition evaluates to true then the alternative default body is executed. If the control structure has no alternative default body, then no body is executed.

6.4 Control statements

It is possible to change or interrupt the execution of a control structure using the keywords `continue`, `break` and `repeat`. A `continue` stops the execution of the specified loop, jumps to the loop header and executes the next iteration. A `break` only interrupts the execution of the specified loop. The keyword `repeat` starts the execution again with the referenced header.

Every control statement references one control header. If no reference is given, it references the innermost header. Possible references are the name of the local parameter which is defined in this head, or the name of the control structure. The name of the control structure belongs to the first head in this control structure.

Usage:

```
continue [reference];
break [reference];
repeat [reference];
```

reference — a reference to a control header specified by a name or a local parameter

break *[reference]* — The execution of the body of the referenced head is cancelled. Remaining statements are skipped.

If the referenced header contains iteration over a set, the execution for the remaining elements of the set is skipped.

continue *[reference]* — The execution of the body of the referenced head is cancelled. Remaining statements are skipped.

If the referenced header contains iteration over a set, the execution is continued with the next element of the set. For other kinds of headers `continue` is equivalent to `break`.

repeat *[reference]* — The execution of the body of the referenced header is cancelled. Remaining statements are skipped.

The execution starts again with the referenced header. The expression in this header is to be evaluated again. If the header contains iteration over a set, the execution starts with the first element. If this header is an assignment to a local parameter, the assignment is executed again. If the header is a condition, the expression is to be checked prior to execution or skipping the body.

6.5 Specific control structures

6.5.1 For loop

A for loop is imbedded in `{ }` and defined by at least one iteration header followed by a loop body separated off by `:`. The loop body contains user-defined instructions which are repeatedly carried out. The number of repeats is based on the iteration header definition.

Usage:

```
{ iterationHeader [, iterationHeader1] [, ...]  : controlBody }
```

iterationHeader *iterationHeader1*	defined iteration headers
controlBody	CMPL statements that are executed in every iteration

Examples:

`{ i := 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` returns
	user messages `hours of product: p1 : 20`
	`hours of product: p2 : 55`
	`hours of product: p3 : 10`

`{i := 1(1)2:`
`  {j := 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`

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

`{i := 1(1)2, j := 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 := 1(1)5, j := 1(1)i:`
`  A[i,j] := i + j;`
`}`
	definition of a triangular matrix

6.5.2 If-then clause

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

Usage:

```
{ condition: thenBody [| default: elseBody ]}
```

condition	If the evaluated condition is true, the code within the body is executed.
thenBody	This body is executed if the *condition* is true.
elseBody	This body is executed if the *condition* is false.

Examples:

`{i := 1(1)5, j := 1(1)5:`
`  {i = j:  A[i,j] := 1; }`
`  {i != j: A[i,j] := 0; }`
`}`
`{i := 1(1)5, j := 1(1)5:`
`  {i = j:  A[i,j] := 1; |`
`  default: 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";`
`  | default: echo "i greater than 9";`
`}`
	example of an if-then-else clause
	returns user message `i greater than 9`

`sum{ i = j : 1 | default: 2 }`	conditional expression, evaluates to 1 if `i = j`, otherwise to 2

6.5.3 Switch clause

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

Usage:

{ *condition1*: *body1* *[***\|** *condition2*: *body2>]* *[***\| ...]** *[***\| default**: *defaultBody]***}**

If the first condition returns TRUE, only *body1* will be executed. Otherwise the next condition *condition1* will be verified. *body2* is executed if all of the previous conditions are not fulfilled. If no condition returns true, then the *defaultBody* is executed.

Example:

`i:=2;`
`{  i=1: echo "i equals 1";`
`  | i=2: echo "i equals 2";`
`  | i=3: echo "i equals 3";`
`  | default: echo "any other value";`
`}`
	example of a switch clause
	returns user message `i equals 2`

6.5.4 While loop

A while loop is imbedded in `{ }` and defined by a condition header followed by a loop body separated off by `:` and finished by the keyword `repeat`. The loop body contains user-defined instructions which are repeatedly carried out until the condition in the loop header is false.

Usage:

```
{ condition : statements repeat; }
```

condition	If the evaluated condition is true, the code within the body is executed. This repeats until the condition becomes false.
statements	one or more user-defined CMPL instructions To prevent an infinite loop the statements in the control body must have an impact on the *condition*.

Examples:

`i:=2;`
`{i<=4:`
`  A[i] := i;`
`  i := i+1;`
`  repeat;`
`}`
	while loop with a global parameter
	Can only be used in the `parameters` section, because the assignment to a global parameter is not permitted in other sections.
	defines `A[2] = 2`, `A[3] = 3` and `A[4] = 4`

`{a := 1, a < 5:`
`  echo a;`
`  a := a + 1;`
`  repeat;`
`}`
	while loop using a local parameter
	Can be used in all sections.
	returns user messages `1`
	`2`
	`3`
	`4`

`{a:=1:`
`  xx {:`
`  echo a;`
`  a := a + 1;`
`  {a>=4: break xx;}`
`  repeat;`
`  }`
`}`
	Alternative formulation:
	The outer control structure defines the local parameter `a`. This control structure is used as a loop with a defined name and an empty header. The name is necessary, because it is needed as reference for the `break` statement in the inner control structure. (Without this reference the `break` statement would refer to the condition a>=4)

6.6 Set and sum control structure as expression

Starting with the keyword `sum` or the keyword `set` a control structure returns an expression. Only expressions are permitted in the body of the control structure. Control statements are not allowed, because the body cannot contain a statement. It is possible to define alternative bodies.

Usage:

```
sum { controlHeader : bodyExpressions }
set { controlHeader : bodyExpressions }
```

controlHeader	header of the control structure The header of a sum or a set control structure is usually an iteration header, but all kinds of control header can be used.
bodyExpressions	user-defined expressions

A `sum` expression repeatedly summarises the user-defined expressions in the *bodyExpressions*. If the body is never executed, it evaluates to 0. A `set` expression returns a set subject to the *controlHeader* and the *bodyExpressions*. The element type included in *bodyExpressions* must be integer or string.

Examples:

`x[1..3] := (2, 4, 6);` `a := sum{i := 1(1)3 : x[i] };`	`a` is assigned `12`

`products:= set( "p1", "p2", "p3");`
`hours[products]:=(20,55,10);`
`totalHours:= sum{i in products: hours[i] };`
	`totalHours` is assigned `85`

`x[1..3,1..2]:=((1,2),(3,4),(5,6));` `b:= sum{i := 1(1)3, j := 1(1)2: x[i,j] };`	using `sum` with more than one control header `b` is assigned `21`.
`s:=set();` `d:= sum{i in s: i |default: -1 };`	sums up all elements in the set `s`. Since `s` is an empty set, `d` is assigned to the alternative default value `-1`.
`e:= set{i:= 1..10: i^2 };`	e is assigned the set `(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)`
`f:= set{i:= 1..100, round(sqrt(i))^2 = i: i };`	f is assigned the set `(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)`

The `sum` expression can also be used in linear terms for the definition of objectives and constraints. In this case the body of the control structure can contain model variables.

Examples:

```
parameters:
      a[1..2,1..3] :=((1,2,3),(4,5,6));
      b[1..2] := (100,100);
      c[1..3] := (20,10,10);
```

```
variables:
      x[1..3]: real[0..];
objectives:
      sum{j:=1..3: c[j] *x[j]}->max;
constraints:
      { i:=1..2:
          sum{j:=1..3: a[i,j] * x[j]}<= b[i];
      }
```

objective definition using a sum

$$20 \cdot x_1 + 10 \cdot x_2 + 10 \cdot x_3 \rightarrow max!$$

constraints definition using a sum

$$1 \cdot x_1 + 2 \cdot x_2 + 3 \cdot x_3 \leq 100$$
$$4 \cdot x_1 + 5 \cdot x_2 + 6 \cdot x_3 \leq 100$$

7 Matrix-Vector notations

CMPL allows users to define objectives and constraints in a matrix-vector notation (e.g. matrix vector multiplication). CMPL generates all required rows and columns automatically by implicit loops.

Implicit loops are formed by matrices and vectors, which are defined by the use of free indices. A free index is an index which is not specified by a position in an array. It can be specified by an entire set or without any specification. But the separating commas between indices must in any case be specified. A multidimensional array with one free index is always treated as a column vector, regardless of where the free index stands. A column vector can be transposed to a row vector with `T`. A multidimensional array with two free indices is always treated as a matrix. The first free index is the row, the second the column.

Implicit loops are only possible in the `objectives` section and the `constraints` section.

Usage:

```
vector[[set]]                    #column vector
vector[[set]]T                   #transpose of column vector  - row vector

matrix[index, [set]]             #column vector
matrix[[set], index]             #also column vector

matrix[index, [set]]T            #transpose of column vector  - row vector
matrix[[set], index]T            #transpose of column vector  - row vector

matrix[[set1], [set2]]           #matrix
```

vector, matrix	name of a vector or matrix
index	a certain index value
[set]	optional specification of a set for the free index

Examples:

`x[]`	vector with free index across the entire defined area
`x[2..5]`	vector with free index in the range 2 – 5

`A[,]`	matrix with two free indices
`A[1,]`	matrix with one fixed and one free index; this is a column vector.
`A[,1]`	matrix with one fixed and one free index; this is also a column vector.

The most important ways to define objectives and constraints with implicit loops are vector-vector multiplication and matrix-vector multiplication. A vector-vector multiplication defines a row of the model (e.g. an objective or one constraint). A matrix-vector multiplication can be used for the formulation of more than one row of the model.

Usage of multiplication using implicit loops :

```
paramVector[set]T * varVector[set]  #vector-vector multiplication
varVector[set]T * paramVector[set]  #vector-vector multiplication

paramMatrix[set1],[set2] * varVector[set2]
                                    #matrix-vector multiplication
varVector[set1]T * paramMatrix[set1],[set2]
                                    #matrix-vector multiplication
```

paramVector	name of a vector of parameters
varVector	name of a vector of model variables
paramMatrix	name of a matrix of parameters
`T`	syntactic element for transposing a vector

Examples:

```
parameters:
      a[1..2,1..3] :=(  (1,2,3),
                        (4,5,6));
      b[1..2] := (100,100);
      c[1..3] := (20,10,10);
variables:
      x[1..3]: real[0..];
objectives:
      c[]T * x[] ->max;

constraints:
      a[, ] * x[] <=b[];
```

objective definition using implicit loops

$$20 \cdot x_1 + 10 \cdot x_2 + 10 \cdot x_3 \rightarrow max\ !$$

constraint definition using implicit loops

$$1 \cdot x_1 + 2 \cdot x_2 + 3 \cdot x_3 \leq 100$$
$$4 \cdot x_1 + 5 \cdot x_2 + 6 \cdot x_3 \leq 100$$

Aside from vector-vector multiplication and matrix-vector multiplication vector subtractions or additions are also useful for the definition of constraints. The addition or subtraction of a variable vector adds new columns to the constraints. The addition or subtraction of a constant vector changes the right side of the constraints.

Usage of additions or subtractions using implicit loops:

```
  linearTerms + varVector[[set]]          #variable vector addition
  linearTerms - varVector[[set]]          #variable vector subtraction

  linearTerms + paramVector[[set]]        #parameter vector addition
  linearTerms - paramVector[[set]]        #parameter vector subtraction
```

linearTerms other linear terms in an objective or constraint

Examples:

<table>
<tr><td>

```
parameters:
      a[1..2,1..3] :=(  (1,2,3),
                        (4,5,6));
      b[1..2] := (100,100);
      d[1..2] := (10,10);
      c[1..3] := (20,10,10);
variables:
      x[1..3]: real[0..];
objectives:
      c[]T * x[] ->max;
constraints:
      a[, ] * x[] + d[] <=b[];
```

</td><td>constraints definition using implicit loops
$1 \cdot x_1 + 2 \cdot x_2 + 3 \cdot x_3 \leq 90$
$4 \cdot x_1 + 5 \cdot x_2 + 6 \cdot x_3 \leq 90$
equivalent to
<code>a[,] * x[] <=b[] - d[];</code></td></tr>
<tr><td><code>0 <= x[1..3]+y[1..3]+z[2]<= b[1..3];</code></td><td>implicit loops for a column vector</td></tr>
<tr><td><code>0 <= x[1] + y[1] + z[2] <= b[1];</code>
<code>0 <= x[2] + y[2] + z[2] <= b[2];</code>
<code>0 <= x[3] + y[3] + z[2] <= b[3];</code></td><td>equivalent formulation</td></tr>
<tr><td>

```
parameters:
      a[1..2,1..3] :=(  (1,2,3),
                        (4,5,6));
      b[1..2] := (100,100);
      d[1..2] := (10,10);
      c[1..3] := (20,10,10);
variables:
      x[1..3]: real[0..];
      z[1..2]: real[0..];
```

</td><td></td></tr>
<tr><td>

```
objectives:
      c[]T * x[] ->max;
```

</td><td></td></tr>
<tr><td>

```
constraints:
      a[,] * x[] + z[] <=b[];
```

</td><td>constraints definition using implicit loops
$1 \cdot x_1 + 2 \cdot x_2 + 3 \cdot x_3 + z_1 \leq 90$
$4 \cdot x_1 + 5 \cdot x_2 + 6 \cdot x_3 + z_2 \leq 90$</td></tr>
</table>

8 Automatic model reformulations

8.1 Overview

CMPL includes two types of automatic code generation which release the user from additional modelling work. CMPL automatically optimizes the generated model by means of matrix reductions. The second type of automatic code reformulations is the equivalent transformation of variable products.

8.2 Matrix reductions

Matrix reductions are subject to constraints of a specific form.

a) If a constraint contains only one variable or only one of the variables with a coefficient not equal to 0, then the constraint is taken as a lower or upper bound.

For the following summation (`x[]` is a variable vector)

```
sum{i:=1(1)2: (i-1) * x[i]} <= 10;
```

no matrix line is generated; rather `x[2]` has an upper bound of 10.

b) If there is a constraint in the coefficients of all variables proportional to another constraint, only the more strongly limiting constraint is retained.

Only the second of the two constraints (`x[]` is a variable vector)

```
2*x[1] + 3*x[2] <= 20;
10*x[1] + 15*x[2] <= 50;
```

is used in generating a model line.

8.3 Equivalent transformations of Variable Products

A product of variables cannot be a part of an LP or MIP model, because such a variable product is a non-linear term. But if one factor of the product is an integer variable then it is possible to formulate an equivalent transformation using a set of specific linear inequations. [cf. Rogge/Steglich (2007)]

The automatic generation of an equivalent transformation of a variable product is a unique characteristic of CMPL.

8.3.1 Variable Products with at least one binary variable

A product of variables with at least one binary variable can be transformed equivalently in a system of linear inequations as follows [Rogge/Steglich (2007), p. 25ff.] :

$w := u \cdot v,\ u \le u \le \bar{u}\ \ (u \text{ real or integer}), v \in \{0,1\}$
is equivalent to

u real or integer, $v \in \{0,1\}$ and
$\underline{u} \cdot v \le w \le \bar{u} \cdot v$
$\underline{u} \cdot (1-v) \le u - w \le \bar{u} \cdot (1-v)$

CMPL is able to perform these transformations automatically. For the following given variables

```
variables:  x: binary;
            y: real[YU..YO];
```

each occurrence of the term `x*y` in the CMPL model description is replaced by an implicit newly-defined variable `x_y`, and the following additional statements are generated automatically:

```
constraints:
      min(YU, 0) <= x_y <= max(YO, 0);
      {YU < 0: x_y - YU*x >= 0; }
      {YO > 0: x_y - YO*x <= 0; }
      y - x_y + YU*x >= YU;
      y - x_y + YO*x <= YO;
```

8.3.2 Variable Product with at least one integer variable

Also for products of variables with at least one integer variable it is possible to formulate an equivalent system of linear inequation [Rogge/Steglich (2007), p. 28ff.] :

$w := u \cdot v,$
$\underline{u} \le u \le \bar{u},\ (u \text{ real or integer, if } u \text{ integer then } \underline{v} - \bar{v} \le \underline{u} - \bar{u}),$
$\underline{v} \le v \le \bar{v}\ (v \text{ integer})$
is equivalent to

u real or integer and
$v = \underline{v} + \sum_{j=0}^{d} 2^j \cdot y_j,\ v \le \bar{v},\ \text{with } d = \lceil ld(\bar{v} - \underline{v} + 1) \rceil - 1$
$w = u \cdot \underline{v} + \sum_{j=0}^{d} 2^j \cdot w_j$
$\underline{u} \cdot y_j \le w_j \le \bar{u} \cdot y_j$
$\underline{u} \cdot (1 - y_j) \le u - w_j \le \bar{u} \cdot (1 - y_j)$

$y_j \in \{0,1\},\ j = 0(1)d$

CMPL is able to perform these transformations automatically as described above. For the following given variables

```
variables:  x: integer[XU..XO];
            y: real[YU..YO];
```

each occurrence of the term `x*y` in the CMPL model description is replaced by an implicit newly-defined variable `x_y`, and the following additional statements are generated automatically (here `d` stands for the number of binary positions needed for `XO-XU+1`):

```
variables:
      _x[1..d]: binary;
      _x_y[1..d]: real;

constraints:
      min(XU*YU,XU*YO,XO*YU,XO*YO) <= x_y <= max(XU*YU,XU*YO,XO*YU,XO*YO);

      x = XU + sum{i=1(1)d: (2^(i-1))*_x[i]};
      x_y = XU*y + sum{i=1(1)d: (2^(i-1))*_x_y[i]};

      {i = 1(1)d:
            min(YU, 0) <= _x_y[i] <= max(YO, 0);
            {YU < 0: _x_y[i] - YU*_x[i] >= 0; }
            {YO > 0: _x_y[i] - YO*_x[i] <= 0; }
            y - _x_y[i] + YU*_x[i] >= YU;
            y - _x_y[i] + YO*_x[i] <= YO;
      }
```

9 CMPL as command line tool

9.1 Usage

The CMPL command line tool can be used in two modes. Using the solver mode, an LP or MIP can be formulated, solved and analysed. In this mode, OSSolverService, GLPK or Gurobi is invoked. In the model mode it is possible to transform the mathematical problem into MPS, Free-MPS or OSiL files that can be used by certain alternative LP or MIP solvers.

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

Usage: cmpl [options] <cmplFile>

Model mode:

-i <cmplFile>	: input file
-m [<File>]	: export model in MPS format in a file or stdout
-x [<File>]	: export model in OSiL XML format in a file or stdout
-noOutput	: no generating of a MPS or OSiL file

Solver mode:

-solver <solver>	: name of the solver you want to use possible options: glpk, clp, cbc, symphony, gurobi, scip
-solverUrl <url>	: URL of the solver service w/o a defined remote solver service, a local solver is used
-solutionCsv	: optimization results in CSV format A file <cmplFileName>.csv will be created.
-solutionAscii	: optimization results in ASCII format A file <cmplFileName>.sol will be created.
-silent	: optimization results are not displayed
-obj <objName>	: name of the objective function
-objSense <max/min>	: objective sense
-maxDecimals <x>	: maximal number of decimals in the solution report (max 12)
-zeroPrecision <x>	: Precision of zero values in the solution report (default 1e-9)
-ignoreZeros	: display only variables and constraints with non-zero values in the solution report
-dontRemoveTmpFiles	: Don't remove temporary files (mps,osil,osrl,osol,gsol)

General options:

-e [<File>]	: output for error messages and warnings -e simple output to stderr (default) -e<File> output in MprL XML format to file
-silent	: suppresses CMPL and solver messages (assuming -e <problemName>.mprl)
-matrix [<File>]	: Writes the generated matrix in a file or on stdout.
-l [<File>]	: output for replacements for products of variables
-s [<File>]	: short statistic info
-p [<File>]	: output for protocol
-gn	: no matrix reductions
-gf	: Generated constraints for products of variables are included at the original position of the product.
-cd	: no warning at multiple parameter definition
-ca	: no warning for deprecated '=' assignment expression
-ci <x>	: mode for integer expressions (0 - 3), (default 1) If the result of an integer operation is outside the range of a long integer then the type of result will change from integer to real. This flag defines the integer range check behaviour. -ci 0 no range check -ci 1 default, range check with a type change if necessary

-ci 2 range check with error message if necessary
-ci 3 Each numerical operation returns a real result

-ff : format option: generate free MPS

-f% <format> : format option for MPS or OSiL files (C++ style - default %f)

-h : get this help

-v : version

Examples - solver mode:

`cmpl test.cmpl`	solves the problem `test.cmpl` locally with the default solver and displays a standard solution report
`cmpl -solver glpk test.cmpl`	solves the problem `test.cmpl` locally using GLPK and displays a standard solution report
`cmpl -solverUrl ↲ http://194.95.44.187:8080/ ↲ OSServer/services/OSSolverService ↲ test.cmpl`	solves the problem `test.cmpl` remotely with the defined web service and displays a standard solution report
`cmpl -solutionCsv test.cmpl`	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.

Examples - model mode:

`cmpl -i test.cmpl -m test.mps`	reads the file `test.cmpl` and generates the MPS-file `test.mps`.
`cmpl -ff -i test.cmpl -m test.mps`	reads the file `test.cmpl` and generates the Free-MPS-file `test.mps`.
`cmpl -i test.cmpl -x test.osil`	reads the file `test.cmpl` and generates the OSiL-file `test.osil`.

9.2 Input and output file formats

9.2.1 Overview

CMPL uses several ASCII files for the communication with the user and other programs such as solvers.

CMPL	input file for CMPL - syntax as described above
MPS	output file for the generated model in MPS format Can be used with most solvers. This format is very restrictive and therefore not recommended.

Free-MPS	output file for the generated model in Free-MPS format Can be used with most solvers.
OSiL	output file for the generated model in OSiL format The OSiL XML schema is developed by the COIN-OR community (COmputational INfrastructure for Operations Research - open source for the operations research community). Can be used with solvers which are supported by the COIN-OR Optimization Services (OS) Framework.
OSoL	OSoL is an XML-based language for representing options that get passed to an optimization solver or a hosted optimization solver Web service. CMPL creates an OsoL file automatically if CBC is chosen and some CBC options are defined in the CMPL header.
OSrL	Optimization Services result Language OSrL (result) is a format for optimization results, if a COIN-OS solver is used.
GLPK plain text (result) format	GLPK can write the results of an optimization in the form of a plain text file. Only used if GLPK or Gurobi is chosen as solver.
CPLEX solution format	an XML based solution file format
SCIP solution format	a plain text file format for SCIP solutions
MPrL	output file for the status of the results or errors of a CMPL model XML file in accordance with the MPrL schema

9.2.2 CMPL

A CMPL file is an ASCII file that includes the user-defined CMPL code with a syntax as described in this manual.

The example

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

can be formulated in CMPL as follows:

```
parameters:
    c[] := ( 1, 2, 3 );
    b[] := ( 15, 20 );

    A[,] := (( 5.6, 7.7, 10.5 ),
             ( 9.8, 4.2, 11.1 ));
variables:
    x[defset(c[])]: real[0..];

objectives:
    profit: c[]T * x[] -> max;
```

```
constraints:
    machine: A[,] * x[] <= b[];
```

9.2.3 MPS

An MPS (Mathematical Programming System) file is a ASCII file for presenting linear programming (LP) and mixed integer programming problems.

MPS is an old format and was the de facto standard for most LP solvers. MPS is column-oriented and is set up for punch cards with defined positions for fields. Owing to these requirements the length of column or row names and the length of a data field are restricted. MPS is very restrictive and therefore not recommended. For more information please see http://en.wikipedia.org/wiki/MPS_(format).

The MPS file for the CMP example given in the section above is generated as follows:

```
* CMPL - MPS - Export
NAME          test.cmpl
* OBJNAME profit
* OBJSENSE max
ROWS
 N  profit
 L  machine_1
 L  machine_2
COLUMNS
    x[1]      profit               1   machine_1      5.600000
    x[1]      machine_2      9.800000
    x[2]      profit               2   machine_1      7.700000
    x[2]      machine_2      4.200000
    x[3]      profit               3   machine_1     10.500000
    x[3]      machine_2     11.100000
RHS
    RHS       machine_1            15   machine_2            20
RANGES
BOUNDS
 PL BOUND     x[1]
 PL BOUND     x[2]
 PL BOUND     x[3]
ENDATA
```

9.2.4 Free - MPS

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.cmpl
* OBJNAME profit
* OBJSENSE max
ROWS
 N profit
 L machine_1
 L machine_2
COLUMNS
 x[1] profit 1 machine_1 5.600000
 x[1] machine_2 9.800000
 x[2] profit 2 machine_1 7.700000
 x[2] machine_2 4.200000
 x[3] profit 3 machine_1 10.500000
 x[3] machine_2 11.100000
RHS
 RHS machine_1 15 machine_2 20
RANGES
BOUNDS
 PL BOUND x[1]
 PL BOUND x[2]
 PL BOUND x[3]
ENDATA
```

9.2.5 OSiL

OSiL is an XML-based format which can be used for presenting linear programming (LP) and mixed integer programming problems. The OSiL XML schema was developed by the COIN-OR community (COmputational INfrastructure for Operations Research - open source for the operations research community). The format makes it very easy to save and present a model and so is particularly suitable for defining an interface to several solvers. An OSiL file can be used with solvers which are supported by the COIN-OR Optimization Services (OS) Framework. [cf. Gassmann, Ma, Martin, Sheng, 2011, p. 38ff.]

The OSiL file for the given CMP example is generated as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<osil xmlns="os.optimizationservices.org"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="os.-
optimizationservices.org http://www.optimizationservices.org/schemas/2.0/OS-
iL.xsd">
      <instanceHeader>
            <name>test.cmpl</name>
            <description>generated by CMPL</description>
      </instanceHeader>
      <instanceData>
            <variables numberOfVariables="3">
                  <var name="x[1]"  type="C" lb="0"/>
```

```
            <var name="x[2]"  type="C" lb="0"/>
            <var name="x[3]"  type="C" lb="0"/>
        </variables>
        <objectives numberOfObjectives="1">
            <obj name="profit" maxOrMin="max" numberOfObjCoef="3">
                <coef idx="0">1</coef>
                <coef idx="1">2</coef>
                <coef idx="2">3</coef>
            </obj>
        </objectives>
        <constraints numberOfConstraints="2">
            <con name="machine_1" ub="15"/>
            <con name="machine_2" ub="20"/>
        </constraints>
        <linearConstraintCoefficients numberOfValues="6">
            <start>
                <el>0</el>
                <el>2</el>
                <el>4</el>
                <el>6</el>
            </start>
            <rowIdx>
                <el>0</el>
                <el>1</el>
                <el>0</el>
                <el>1</el>
                <el>0</el>
                <el>1</el>
            </rowIdx>
            <value>
                <el>5.600000</el>
                <el>9.800000</el>
                <el>7.700000</el>
                <el>4.200000</el>
                <el>10.500000</el>
                <el>11.100000</el>
            </value>
        </linearConstraintCoefficients>
    </instanceData>
```

9.2.6 OSoL

"OSoL is an XML-based language for representing options that get passed to an optimization solver or a hosted optimization solver Web service. For more information please visit the project website of COIN-OR OS project." [Gassmann, Ma, Martin, Sheng, 2011, p. 41ff.]

The following OSoL-file describes a couple of parameters for the CBC solver.

```
<?xml version="1.0" encoding="UTF-8"?>
<osol xmlns="os.optimizationservices.org"
     xmlns:xs="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="os.optimizationservices.org http://www.optimization-
services.org/schemas/OSoL.xsd">
       <optimization>
          <solverOptions numberOfSolverOptions="1">
          <solverOption name="threads"  solver="cbc"  value="2"  />
       </solverOptions>
   </optimization>
</osol>
```

9.2.7 MPrL

MPrL 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. MPrL is intended for communication with other software that uses CMPL for modelling linear optimization problems.

An MPrL 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 `<mplResult>` section consists of one or more messages about specific lines in the CMPL model.

After the transformation of the given CMPL model, CMPL will finish without errors. The general status is represented in the following MPrL file.

```
<?xml version="1.0" encoding="UTF-8"?>
<mprl xmlns="www.coliop.org" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="www.coliop.org
http://www.coliop.org/schemas/Mprl.xsd">
    <general>
      <generalStatus>normal</generalStatus>
      <mplName>CMPL</mplName>
      <instanceName>test.cmpl</instanceName>
      <message>cmpl finished normal</message>
    </general>
</mprl>
```

If a semicolon is not set in line 26, CMPL will finish with errors that are represented in the following MPrL file.

```
<?xml version="1.0" encoding="UTF-8"?>
<mprl xmlns="www.coliop.org" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="www.coliop.org
http://www.coliop.org/schemas/Mprl.xsd">
    <general>
      <generalStatus>error</generalStatus>
      <mplName>CMPL</mplName>
      <instanceName>test.cmpl</instanceName>
```

```
      <message>cmpl finished with errors</message>
    </general>
    <mplResult numberOfMessages="1">
      <mplmessage type ="error" file="test.cmpl" line="26" description="syntax
           error"/>
    </mplResult>
</mprl>
```

The MPrL schema is defined as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
    <xsd:element name="mprl">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element ref="general" minOccurs="1" maxOccurs="1"/>
          <xsd:element ref="mplResult" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="general">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element ref="generalStatus" minOccurs="1" maxOccurs="1"/>
          <xsd:element name="instanceName" type="xsd:string" minOccurs="1"
                maxOccurs="1"/>
          <xsd:element name="mplName" type="xsd:string" minOccurs="1"
                maxOccurs="1"/>
          <xsd:element name="message" type="xsd:string" minOccurs="0"
                maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="generalStatus">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="error"/>
          <xsd:enumeration value="warning"/>
          <xsd:enumeration value="normal"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
    <xsd:element name="mplResult">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element ref="mplMessage" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="numberOfMessages" type="xsd:nonNegativeInteger"
          use="required"/>
      </xsd:complexType>
```

```
    </xsd:element>
    <xsd:element name="mplMessage">
      <xsd:complexType>
        <xsd:attribute name="type" use="required">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                <xsd:enumeration value="error"/>
                <xsd:enumeration value="warning"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
        <xsd:attribute name="file" type="xsd:string" use="required"/>
        <xsd:attribute name="line" type="xsd:nonNegativeInteger"
            use="required"/>
        <xsd:attribute name="description" type="xsd:string" use="required"/>
      </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

9.2.8 OSrL

"OSrL is an XML-based language for representing the solution of large-scale optimization problems including linear programs, mixed-integer programs, quadratic programs, and very general nonlinear programs." [Gassmann, Ma, Martin, Sheng, 2011, p. 39ff.]

The following OSrL-file contains the results of the given problem.

```
<?xml version="1.0" encoding="UTF-8"?><?xml-stylesheet type="text/xsl"
href="http://www.coin-or.org/OS/stylesheets/OSrL.xslt"?>
<osrl xmlns="os.optimizationservices.org"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="os.-
optimizationservices.org http://www.optimizationservices.org/schemas/2.0/OS-
rL.xsd" >
<general>
    <generalStatus type="normal">
    </generalStatus>
    <serviceName>

    Optimization Services Solver
    Main Authors: Horand Gassmann, Jun Ma, and Kipp Martin
    Distributed under the Eclipse Public License
    OS Version: 2.4.4
    Build Date: Feb 26 2012
    SVN Version: 4456

    </serviceName>
    <instanceName>test.cmpl</instanceName>
    <solverInvoked>COIN-OR clp</solverInvoked>
</general>
```

```
<job>
    <timingInformation numberOfTimes="1">
    <time type="elapsedTime" unit="second" category="total">
    5.699999999999998e-4</time>
    </timingInformation>
</job>
<optimization numberOfSolutions="1" numberOfVariables="3"
    numberOfConstraints="2" numberOfObjectives="1">
    <solution targetObjectiveIdx="-1">
      <status type="optimal">
      </status>
      <variables numberOfOtherVariableResults="1">
        <values numberOfVar="3">
          <var idx="0">0</var>
          <var idx="1">0</var>
          <var idx="2">1.4285714285714284</var>
        </values>
        <basisStatus>
          <basic numberOfEl="1"><el>2</el></basic>
          <atLower numberOfEl="2"><el>0</el><el>1</el></atLower>
        </basisStatus>
        <other numberOfVar="3" name="reduced costs"
          description="the variable reduced costs">
          <var idx="0">-.5999999999999996</var>
          <var idx="1">-.20000000000000012</var>
          <var idx="2">-0</var>
        </other>
      </variables>
      <objectives >
        <values numberOfObj="1">
          <obj idx="-1">4.285714285714285</obj>
        </values>
      </objectives>
      <constraints >
        <dualValues numberOfCon="2">
          <con idx="0">.2857142857142857</con>
          <con idx="1">-0</con>
        </dualValues>
        <basisStatus>
          <basic numberOfEl="1"><el>1</el></basic>
          <atLower numberOfEl="1"><el>0</el></atLower>
        </basisStatus>
      </constraints>
    </solution>
</optimization>
</osrl>
```

9.2.9 GLPK plain text (result) format

GLPK is able to write the solution of an optimization in the form of a plain text file in a specific structure. Please see for details [GLPK, 2011, p. 105ff.] Despite that Gurobi does not support this format CMPL uses this file format also for Gurobi because the interaction with Gurobi is established by a python script that is able to write the Gurobi optimization results in the GLPK plain text (result) format.

The following GLPK result file describes the solution of the example.

```
3 3
2 2 4.28571428571429
1 4.28571428571429 0
3 15 0.285714285714286
1 15.8571428571429 0
2 0 -0.6
2 0 -0.2
1 1.42857142857143 0
```

9.2.10 CPLEX solution file format

"CPLEX … writes solution files, formatted in XML, for all problem types, for all application programming interfaces (APIs). ... The XML solution file format makes it possible for you to display and view these solution files in most browsers as well as to pass the solution to XML-aware applications." [CPLEX manual → File formats supported by CPLEX → SOL file format: solution files]

CMPL is able to read the CMPL solution file.

The following CPLEX result file describes the solution of the example.

```
<?xml version = "1.0" encoding="UTF-8" standalone="yes"?>
<CPLEXSolution version="1.2">
 <header
   problemName="test.mps"
   objectiveValue="4.28571428571429"
   solutionTypeValue="1"
   solutionTypeString="basic"
   solutionStatusValue="1"
   solutionStatusString="optimal"
   solutionMethodString="dual"
   primalFeasible="1"
   dualFeasible="1"
   simplexIterations="1"
   writeLevel="1"/>
 <quality
   epRHS="1e-06"
   epOpt="1e-06"
   maxPrimalInfeas="0"
   maxDualInfeas="0"
   maxPrimalResidual="1.99840144432528e-15"
   maxDualResidual="0"
   maxX="1.42857142857143"
```

```
   maxPi="0.285714285714286"
   maxSlack="4.14285714285715"
   maxRedCost="0.6"
   kappa="3.83642857142857"/>
 <linearConstraints>
  <constraint name="machine_1" index="0" status="LL" slack="0"
    dual="0.285714285714286"/>
  <constraint name="machine_2" index="1" status="BS" slack="4.14285714285715"
    dual="-0"/>
 </linearConstraints>
 <variables>
  <variable name="x[1]" index="0" status="LL" value="0" reducedCost="-0.6"/>
  <variable name="x[2]" index="1" status="LL" value="0" reducedCost="-0.2"/>
  <variable name="x[3]" index="2" status="BS" value="1.42857142857143"
    reducedCost="-0"/>
 </variables>
</CPLEXSolution>
```

9.2.11 SCIP solution file format

SCIP writes the solution of the variables in the form of a plain text file in a specific structure. Because SCIP is only used for MIPs the result file format does not contain information about reduced costs or shadow prices.

The following SCIP result file describes the solution of the example if the variables defined as integers.

```
solution status: optimal solution found
objective value:                                 3
x[3]                                             1   (obj:3)
```

9.3 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 installation routine installs a customized version of the COIN-OR OSSolverService (including the COIN-OR solvers CLP, CBC and Symphony) and GLPK. OSSolverService is the default optimization environment. If you have installed Gurobi, CPLEX or SCIP then you can also use these solvers directly.

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

9.3.1 COIN-OR OSSolverService

"The objective of Optimization Services (OS) is to provide a general framework consisting of a set of standards for representing optimization instances, results, solver options, and communication between clients and solvers in a distributed environment using Web Services. … A command line executable OSSolverService for

reading problem instances (OSiL format, AMPL nl format, MPS format) and calling a solver either locally or on a remote server." [Gassmann, Ma, Martin, Sheng, 2011, p. 4.]

For more information please visit https://projects.coin-or.org/OS.

The CMPL distribution contains a customized version of OSSolverService including the COIN-OR solvers CLP, CBC and Symphony. OSSolverService is the default solver environment for CMPL. It is possible but not necessary to specify which solver is to be used. The default solver for LPs is CLP and for MIPs CBC.

OSSolverService can be used in two modes:

```
cmpl <problem>.cmpl                  #Solves the problem locally
                                     #with Clp (LP) or CBC (MIP)
cmpl <problem>.cmpl ↲
     -solverUrl http://<domain>:8080/OSServer/services/OSSolverService

                                     #Solves the problem using
                                     #OSServer located at <domain>
                                     #with Clp (LP) or CBC (MIP)
```

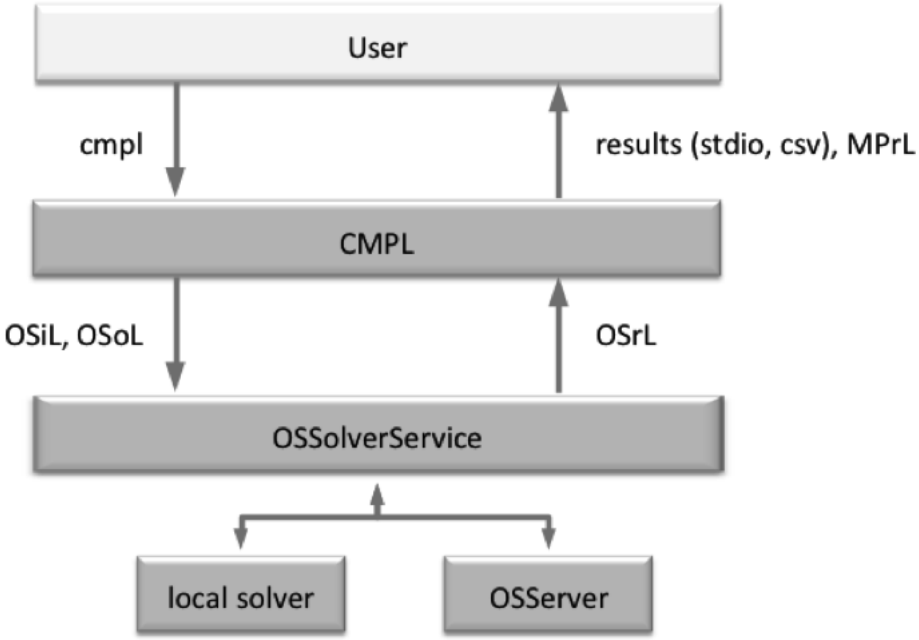
For more information about the OSServer please visit the COIN-OS website: https://projects.coin-or.org/OS.

It is possible to use most of the CBC solver options within the CMPL header. Please see for a list of useful CBC parameters in Appendix 12.1.

Usage of CBC parameters within the CMPL header:

```
%opt cbc solverOption [solverOptionValue]
```

CMPL interacts with the OSSolverService directly as shown in the picture below.



The user has to formulate the CMPL model. CMPL transforms the CMPL model into an OSiL file. If some CBC solver parameters are defined in the CMPL header, then CMPL generates an OSoL file.

After generating the OSiL file (and if needed the OSoL file) CMPL executes OSSolverService directly as an external process. OSSolverService executes either a local solver or a remote solver. CMPL waits for the OSSolverService process. After finishing OSSolverService - and if no error occurs - the result file (OSrL) is to be read by CMPL. After that CMPL creates a standard report or exports the solver results in an ASCII or CSV file.

9.3.2 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, 2011, p. 212.]. For more information please visit the GLPK project website: http://www.gnu.org/software/glpk.

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:

```
%arg -solver glpk
```

Most of the GLPK solver options can be used by defining solver options within the CMPL header. Please see Appendix 12.2 for a list of useful GLPK parameters.

Usage of GLPK parameters within the CMPL header:

```
%opt glpk solverOption [solverOptionValue]
```

9.3.3 Gurobi

"The Gurobi Optimizer is a state-of-the-art solver for linear programming (LP), quadratic programming (QP) and mixed-integer programming (MIP including MILP and MIQP). It was designed from the ground up to exploit modern multi-core processors. For solving LP and QP models, the Gurobi Optimizer includes high-performance implementations of the primal simplex method, the dual simplex method, and a parallel barrier solver. For MILP and MIQP models, the Gurobi Optimizer incorporates the latest methods including cutting planes and powerful solution heuristics." [www.gurobi.com]

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:

```
%arg -solver gurobi
```

All Gurobi parameters (excluding NodefileDir, LogFile and ResultFile) described in the Gurobi manual can be used in the CMPL header. Please see: http://www.gurobi.com/doc/46/refman/node591.html

Usage of Gurobi parameters within the CMPL header:

```
%opt gurobi solverOption [solverOptionValue]
```

9.3.4 SCIP

SCIP is a project of the Konrad-Zuse-Zentrum fuer 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]

SCIP can be used only for mixed integer programming (MIP) problems. If SCIP is chosen as solver and the problem is an LP then CLP 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 `ScipFileName` in the file `<cmplhome>/bin/cmpl.opt`.

Examples:

`ScipFileName = /Applications/Scip/scip`	The binary scip is located in the folder /Applications/Scip
`ScipFileName = /Program Files/Scip/scip.exe`	Example for a Windows system. Please keep in mind to use a slash as a path separator.

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:

```
%arg -solver scip
```

All SCIP parameters described in the SCIP Doxygen Documentation can be used in the CMPL header.
Please see: http://scip.zib.de/doc/html/PARAMETERS.html

Usage SCIP parameters within the CMPL header:

```
%opt scip solverOption solverOptionValue
```

Please keep in mind, that in contrast to the SCIP Doxygen Documentation you do not have to use = as assignment operator between the *solverOption* and the *solverOptionValue*.

Examples:

`%opt scip branching/scorefunc p`	CMPL solver parameter description for `the parameter branching score function` which is described in the SCIP Doxygen Documentation as follows:
	`# branching score function ('s'um, 'p'roduct)`
	`# [type: char, range: {sp}, default: p]`
	`branching/scorefunc = p`

`%opt scip lp/checkfeas TRUE`	`# should LP solutions be checked, resolving LP when numerical troubles occur?`
	`# [type: bool, range: {TRUE,FALSE}, default: TRUE]`
	`lp/checkfeas = TRUE`

`%opt scip lp/fastmip 1`	`# which FASTMIP setting of LP solver should be used? 0: off, 1: low`
	`# [type: int, range: [0,1], default: 1]`
	`lp/fastmip = 1`

9.3.5 CPLEX

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

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

Example:

`CplexFileName = /Applications/IBM/ILOG/ ↲`
`CPLEX_Studio_Academic124/cplex/ ↲`
`bin/x86-64_darwin9_gcc4.0/cplex`
	The binary scip is located in the specified folder

Please note that for Windows installations you also have to use slashes as a path separators (instead of the usual backslashes). 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:

```
%arg -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 solverOption solverOptionValue
```

You have to use the parameters for the Interactive Optimizer. The names of sub-parameters of hierarchical parameters are to be separated by slashes.

Examples:

`%opt cplex threads 2`	Sets the default number of parallel threads that will be invoked.
`%opt cplex mip/limits/aggforcut 4`	Limits the number of constraints that can be aggregated for generating flow cover and mixed integer rounding (MIR) cuts to 4.
`%opt cplex ↲ simplex/tolerances/optimality ↲ 1e-8`	Sets the reduced-cost tolerance for optimality to 1e-8.

9.3.6 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:

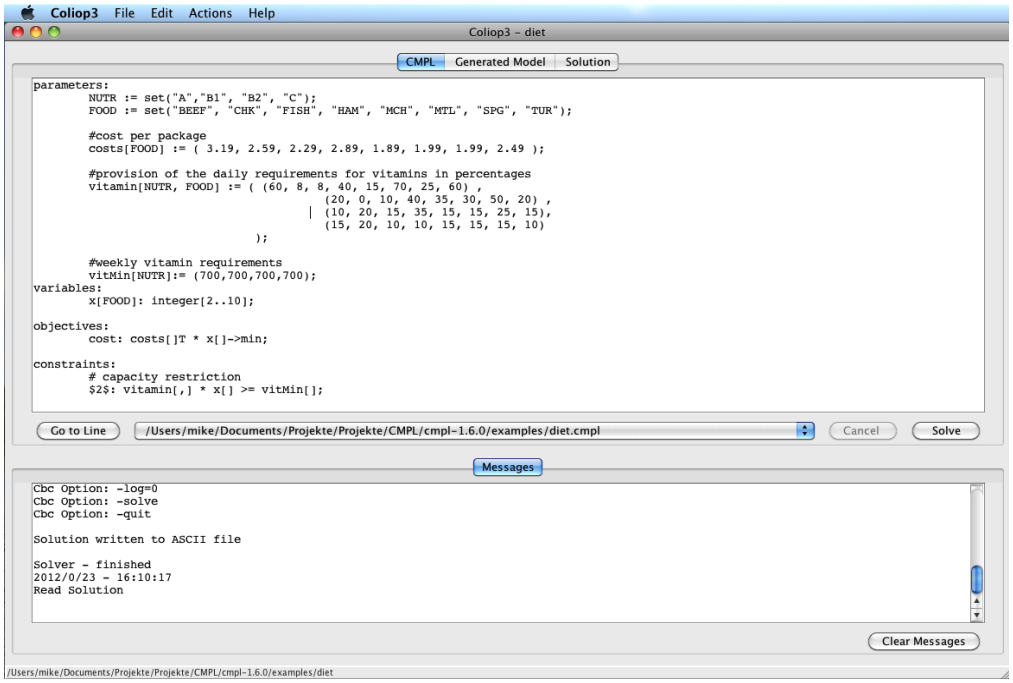
```
cmpl -m <problemname>.mps <problemname>.cmpl          #MPS export
cmpl -ff -m <problemname>.mps <problemname>.cmpl      #Free-MPS export
cmpl -x <problemname>.osil <problemname>.cmpl         #OSiL export
```

9.4 Using CMPL with Coliop

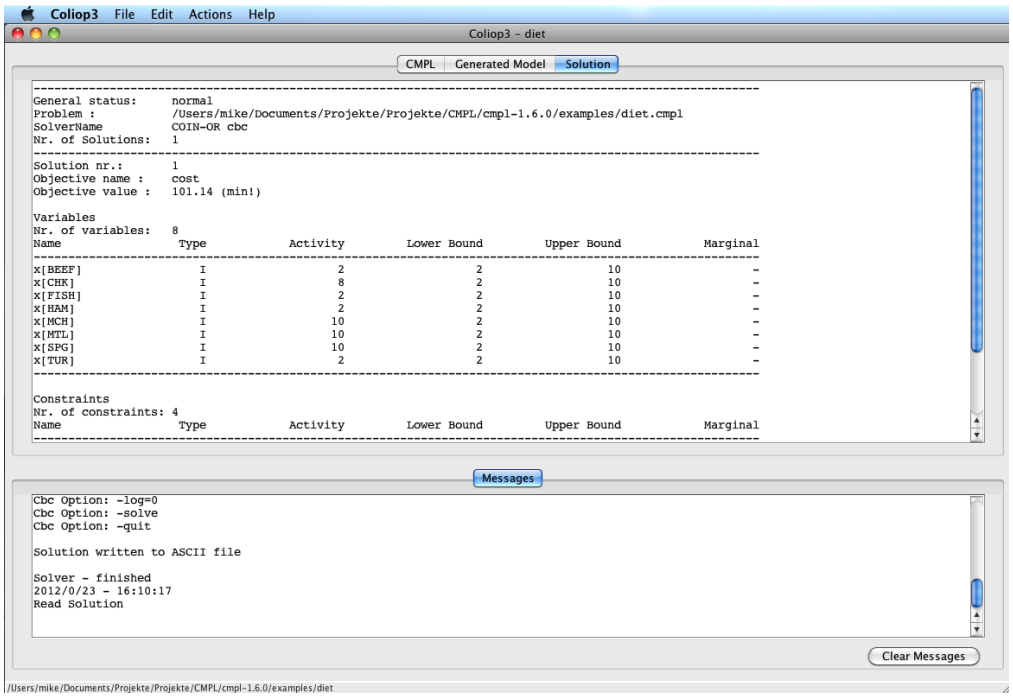
Coliop is an (simple) IDE (Integrated Development Environment) for CMPL intended to solve linear programming (LP) problems and mixed integer programming (MIP) problems. Coliop is a project of the Technical University of Applied Sciences Wildau and the Institute for Operations Research and Business Management at

the Martin Luther University Halle-Wittenberg. Coliop is an open source project licensed under GPL. It is written in Java and is as an integral part of the CMPL distribution available for most of the relevant operating systems (OS X, Linux and Windows). Because Coliop is written in Java it is necessary to install the java runtime environment. Please visit to install java: http://java.com/de/download/index.jsp.

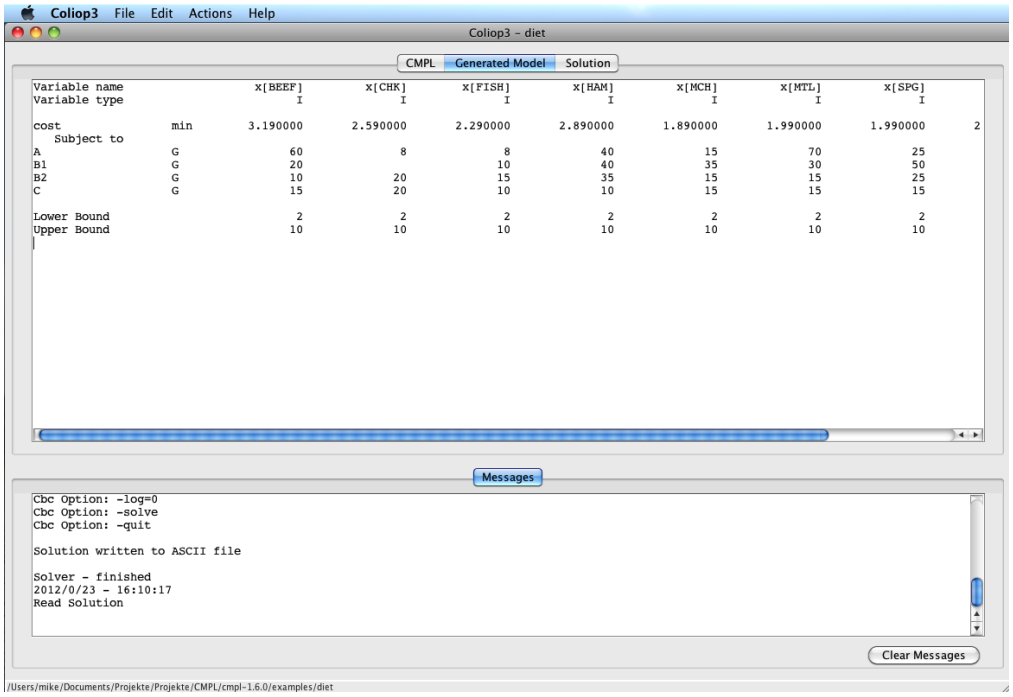
The first working step is to create or to open a CMPL model. By pushing the button <Solve> the model can be solved.



If a syntax error occurs then a user can analyse it by checking the CMPL messages. If CMPL is executed w/o any problems the standard solution report appears.



It is also possible to check the generated model matrix in the tab <Generated Model>.



10 Examples

10.1 Selected decision problems

10.1.1 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` is formulated as follows:

```
parameters:
      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);
variables:
      x[FOOD]: integer[2..10];

objectives:
      cost: costs[]T * x[]->min;

constraints:
      # capacity restriction
      $2$: vitamin[,] * x[] >= vitMin[];
```

CMPL command:

```
cmpl diet.cmpl
```

Solution:

```
-----------------------------------------------------------------------------------------------------
Problem              diet.cmpl
Nr. of variables     8
Nr. of constraints   4
Status               optimal
```

```
Solver name          COIN-OR cbc
Objective name       cost
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
-----------------------------------------------------------------------------------------------------------
A                       G                1500                700           Infinity                   -
B1                      G                1330                700           Infinity                   -
B2                      G                 860                700           Infinity                   -
C                       G                 700                700           Infinity                   -
-----------------------------------------------------------------------------------------------------------
```

10.1.2 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:

```
parameters:
      price[] := (500, 600, 450 );
      costs[] := (425, 520, 400);

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

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

      #profit contribution per unit
      {j:=defset(price[]):  c[j] := price[j]-costs[j]; }

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

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

objectives:
      profit: c[]T  * x[] ->max;

constraints:
      res: a[,] * x[] <= b[];
      0<=x[]<=xMax[];
```

CMPL command:

```
cmpl production-mix.cmpl -solver glpk
```

Solution:

```
-----------------------------------------------------------------------------------------------------------
Problem              production-mix.cmpl
Nr. of variables     3
Nr. of constraints   2
Status               normal
Solver name          GLPK
Objective name       profit
Objective value      6448.28 (max!)
-----------------------------------------------------------------------------------------------------------

Variables
Name                 Type            Activity         Lower bound         Upper bound            Marginal
-----------------------------------------------------------------------------------------------------------
x[1]                    C             34.4828                   0                 250                   0
x[2]                    C             48.2759                   0                 240                   0
x[3]                    C                   0                   0                 250                 -14
-----------------------------------------------------------------------------------------------------------
```

```
Constraints
Name                 Type            Activity         Lower bound         Upper bound            Marginal
------------------------------------------------------------------------------------------------------------
res_1                   L                1000           -Infinity                1000             3.10345
res_2                   L                1000           -Infinity                1000             3.34483
------------------------------------------------------------------------------------------------------------
```

10.1.3 Production mix including thresholds and step-wise fixed costs

This model calculates the production mix that maximizes 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.

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:

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

$$s.t.$$

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

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

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

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

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

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

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

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

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

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

```
parameters:
      price[] := (500, 600, 450 );
      costs[] := (425, 520, 400);

      #machine hours required per unit
      a[,] := ((8, 15, 12), (15, 10, 8));
      #upper bounds of the machines
      b[] := (1000, 1000);

      #profit contribution per unit
      {j:=defset(price[]):  c[j] := price[j]-costs[j]; }

      #upper bound of a product
      xMax[] := (250, 240, 250 );
      xMin[] := (45, 45, 45 );

      #fixed setup costs
      FC[] := ( 500, 400, 500);
variables:
      {j:=1(1)dim(c[]): x[j]: integer[0..xMax[j]]; }
      y[defset(price[])] : binary;

objectives:
      profit: c[]T  * x[] - FC[]T * y[] ->max;

constraints:
      res: a[,] * x[] <= b[];
      {j in defset(price[]): xMin[j] * y[j] <= x[j] <= xMax[j] * y[j]; }
```

CMPL command:

```
cmpl production-mix-fixed-costs.cmpl
```

Solution:

```
-------------------------------------------------------------------------------------------------------------
Problem              production-mix-fixed-costs.cmpl
Nr. of variables     6
Nr. of constraints   8
Status               optimal
Solver name          COIN-OR cbc
Objective name       profit
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           -Infinity                1000                 -
res_2                   L                 660           -Infinity                1000                 -
line_4                  L                   0           -Infinity                   0                 -
line_5                  L                   0           -Infinity                   0                 -
line_6                  L                 -21           -Infinity                   0                 -
line_7                  L                -174           -Infinity                   0                 -
line_8                  L                   0           -Infinity                   0                 -
line_9                  L                   0           -Infinity                   0                 -
-----------------------------------------------------------------------------------------------------
```

10.1.4 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	weight [pounds]	price [€/box]
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.cmpl:

```
parameters:
      boxes := 1(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-basic.cmpl` can be formulated as follows:

```
include "knapsack-data.cmpl"
variables:
      x[boxes] : binary;
objectives:
      sales: p[]T * x[] ->max;
constraints:
      weight: w[]T * x[] <= maxWeight;
```

CMPL command:

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

Solution:

```
-----------------------------------------------------------------------------------------------------------
Problem              knapsack-max-basic.cmpl
Nr. of variables     10
Nr. of constraints   1
Status               optimal
Solver name          COIN-OR cbc
Objective name       sales
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                   0                   0                   1                   -
x[4]                    B                   1                   0                   1                   -
x[5]                    B                   0                   0                   1                   -
x[6]                    B                   1                   0                   1                   -
x[7]                    B                   0                   0                   1                   -
x[8]                    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                  60           -Infinity                  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:

```
include "knapsack-data.cmpl"
variables:
      x[boxes] : binary;
objectives:
      weight: w[]T * x[] ->min;
constraints:
      sales: p[]T * x[] >= minSales;
```

CMPL command:

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

Solution:

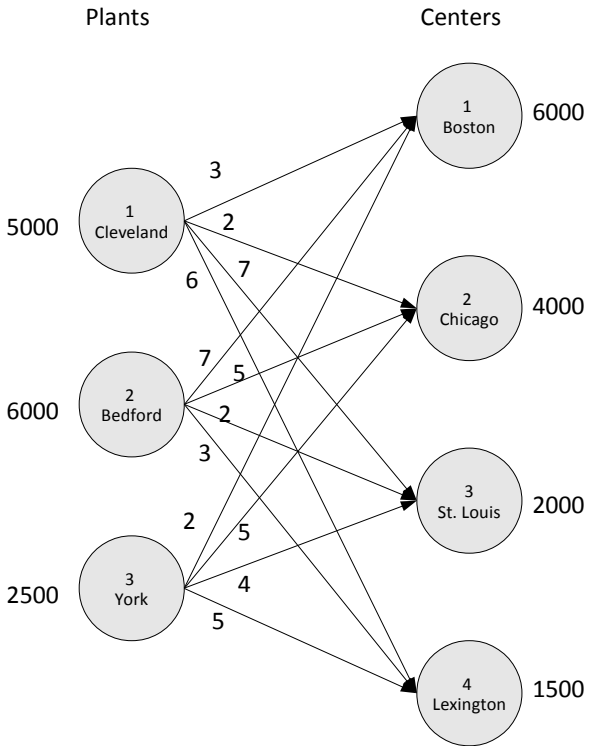
```
---------------------------------------------------------------------------------------------------------
Problem              knapsack-min-basic.cmpl
Nr. of variables     10
Nr. of constraints   1
Status               optimal
Solver name          COIN-OR cbc
Objective name       weight
Objective 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[3]                    B                   0                   0                   1                   -
x[4]                    B                   1                   0                   1                   -
x[5]                    B                   0                   0                   1                   -
x[6]                    B                   0                   0                   1                   -
x[7]                    B                   0                   0                   1                   -
x[8]                    B                   0                   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            Infinity                   -
---------------------------------------------------------------------------------------------------------
```

10.1.5 Transportation problem

A transportation 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/Sweeney/Williams/Martin: An Introduction to Management Science - Quantitative Approaches to Decision Making, 13th ed., South-Western, Cengage Learning 2008, p. 261ff.

This problem involves the transportation of a product from three plants to four distribution centers. 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 firm distributes its generators through four regional distribution centers 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 center. The objective is to minimize the transportation costs.



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

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

$$s.t.$$

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

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

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

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 as follows:

```
%arg -ignoreZeros
parameters:
      plants  := 1(1)3;
      centers := 1(1)4;

      s[plants]  := (5000,6000,2500);
      d[centers] := (6000,4000,2000,1500);

      c[plants,centers] := ( (3,2,7,6), (7,5,2,3),(2,5,4,5) );

variables:
      x[plants,centers]: real[0..];

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

constraints:
      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];}
```

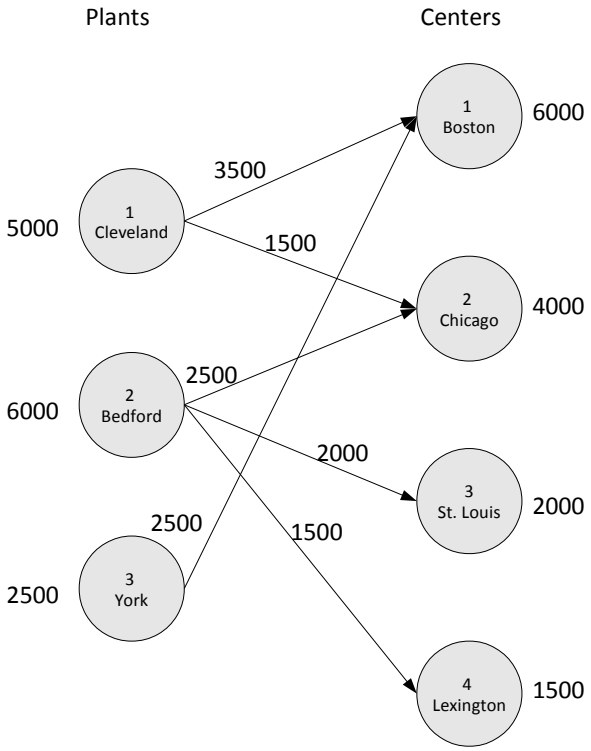
CMPL command:

```
cmpl  transportation.cmpl
```

Solution:

```
-------------------------------------------------------------------------------------------------------------
Problem              transportation.cmpl
Nr. of variables     12
Nr. of constraints   7
Status               optimal
Solver name          COIN-OR clp
Objective name       costs
Objective value      39500 (min!)
-------------------------------------------------------------------------------------------------------------

Nonzero variables
Name                 Type            Activity        Lower bound        Upper bound            Marginal
-------------------------------------------------------------------------------------------------------------
x[1,1]                  C                3500                  0           Infinity                   0
x[1,2]                  C                1500                  0           Infinity                   0
x[2,2]                  C                2500                  0           Infinity                   0
x[2,3]                  C                2000                  0           Infinity                   0
x[2,4]                  C                1500                  0           Infinity                   0
x[3,1]                  C                2500                  0           Infinity                   0
-------------------------------------------------------------------------------------------------------------

Nonzero constraints
Name                 Type            Activity        Lower bound        Upper bound            Marginal
-------------------------------------------------------------------------------------------------------------
```

supplies_1	L	5000	-Infinity	5000	-3
supplies_2	L	6000	-Infinity	6000	-
supplies_3	L	2500	-Infinity	2500	-4
demands_1	E	6000	6000	6000	6
demands_2	E	4000	4000	4000	5
demands_3	E	2000	2000	2000	2
demands_4	E	1500	1500	1500	3



10.1.6 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

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

The general model can be formulated as follows:

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

$$s.t.$$

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

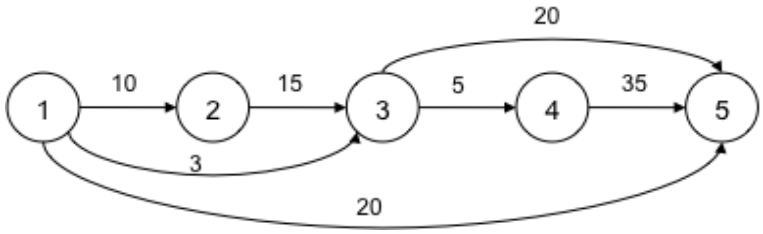
$$\sum_{h=1}^{n} x_{hj}=1 \quad ;j=1(1)n$$

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

Because of the product $x_{hj}\cdot x_{ik}$ in the objective function the model is not a linear model. But it is possible to use a set of inequations to make an equivalent transformation of such multiplications of variables. This transformation is implemented in CMPL and the set of inequations will be generated automatically.

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.



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:

```
%arg -solver glpk
%arg -ignoreZeros
%display var x[*
%display con sos*

parameters:
      n:=5;
      M:=1000;
      d[,]:= (    ( M, 1, 2, 3, 4),
                  ( 2, M, 1, 2, 3),
                  ( 3, 1, M, 1, 2),
                  ( 2, 3, 1, M, 1),
                  ( 5, 3, 1, 1, M) );
```

```
      t[,]:= (    ( 0, 10, 10, 0, 20),
                  ( 0, 0,  15, 0, 0 ),
                  ( 0, 0,  0,  5, 20),
                  ( 0, 0,  0,  0, 35),
                  ( 0, 0,  0,  0, 0 ) );
variables:
      x[1..n,1..n]: binary;
      #dummy variables to store the products x_hj * x_ik
      w[1..n,1..n,1..n,1..n]: real[0..1];

objectives:
      costs: sum{  h:=1(1)n, i:=1(1)n, j:=1(1)n, k:=1(1)n :
                  t[h,i]*d[j,k]*w[h,j,i,k] } ->min;
constraints:
      { h:=1(1)n, i:=1(1)n, j:=1(1)n, k:=1(1)n:
                  { t[h,i] = 0:  w[h,j,i,k] = 0; |
                    # definition of the products x_hj * x_ik
                    default: w[h,j,i,k] = x[h,j] * x[i,k]; }
      }
      sos1 { h:=1(1)n: sum{ j:=1(1)n: x[h,j] } = 1; }
      sos2 { j:=1(1)n: sum{ h:=1(1)n: x[h,j] } = 1; }
```

CMPL command:

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

Solution:

```
-----------------------------------------------------------------------------------------------------------
Problem              quadratic-assignment.cmpl
Nr. of variables     375
Nr. of constraints   710
Status               normal
Solver name          GLPK
Objective name       costs
Objective value      155 (min!)
-----------------------------------------------------------------------------------------------------------

Nonzero variables (x[*)
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                   -
-----------------------------------------------------------------------------------------------------------

Nonzero constraints (sos*)
Name                 Type            Activity         Lower bound         Upper bound            Marginal
-----------------------------------------------------------------------------------------------------------
sos1_1                  E                   1                   1                   1                   -
sos1_2                  E                   1                   1                   1                   -
sos1_3                  E                   1                   1                   1                   -
sos1_4                  E                   1                   1                   1                   -
sos1_5                  E                   1                   1                   1                   -
sos2_1                  E                   1                   1                   1                   -
```

```
sos2_2                E                1                1                1                -
sos2_3                E                1                1                1                -
sos2_4                E                1                1                1                -
sos2_5                E                1                1                1                -
--------------------------------------------------------------------------------------------------------
```

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

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

10.1.7 Generic travelling salesman problem

The 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.

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

```
%arg -solver cbc
%arg -cd
%arg -ignoreZeros
%opt cbc threads 2
%display var x*

 parameters:
      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);
          }
```

```
      }

variables:
      x[cities,cities]: binary;
      u[cities]: real[0..];

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

constraints:
      sos_i {j in cities:  sum{i in cities: x[i,j]}=1; }
      sos_j {i in cities:  sum{j in cities: x[i,j]}=1; }

      noSubs {i:=2..nrOfCities, j:=2..nrOfCities, 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
Status               optimal
Solver name          COIN-OR cbc
Objective name       distance
Objective value      321.319 (min!)
-------------------------------------------------------------------------------------------------------------

Nonzero variables (x*)
Name                 Type            Activity         Lower bound         Upper bound             Marginal
-------------------------------------------------------------------------------------------------------------
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                    -
-------------------------------------------------------------------------------------------------------------
```

The tour is optimal as follows:

1→4→10→3→6→9→5→8→7→2→1

10.2 Using CMPL as a pre-solver

CMPL is not only intended to generate models in the MPS or OSIL format. CMPL can also be used as a pre-solver or simple solver. In this way it is possible to find a preliminary solution of a problem as a basis for the model which is to be generated.

10.2.1 Solving the knapsack problem

The knapsack problem is a very simple problem that does not necessarily have to be solved by an MIP solver. CMPL can be used as a simple solver for knapsack problems to approximate the optimal solution.

The idea of the following models is to evaluate each item using the relation between the value per item 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 section10.1.4a CMPL model to maximize the total sales relative to capacity can be formulated as follows.

Model 1: maximize the total sales `knapsack-max-presolved.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]; }
```

CMPL command:

```
cmpl knapsack-max-presolved.cmpl -noOutput -cd
```

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 not identical to the optimal solution on page 62 but good enough as an approximate solution.

Model 2: minimize the total weight `knapsack-min-presolved.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]; }
```

CMPL command:

```
cmpl knapsack-min-presolved.cmpl -noOutput -cd
```

Solution:

```
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 10.1.4.

10.2.2 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. [cf. Hillier/Liebermann (2010), p. 554f.]

A CMPL program to find the maximum of $f(x)=12\cdot x-3\cdot x^4-2\cdot x^6$ can be formulated as follows (`bisection.cmpl`):

```
parameters:
      #distance epsilon
      e:=0.02;
      #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): " + format("%10.4f",fd) + " xl: " +
                  format("%6.4f",xl) +
                  " xo: " + format("%6.4f",xo) + " xn: " + format("%6.4f",xn) +
                  " f(xn): " + format("%6.4f",fx);
            repeat;
      }
      echo "Optimal solution found";
      x:=xn;
      echo "x: "+ format("%2.3f",x);
      echo "function value: " + (12 * x -3 * x^4 - 2 * x^6);
```

CMPL command:

```
cmpl bisection.cmpl -noOutput -cd
```

Solution:

```
f'(xn):   -12.0000 xl: 0.0000 xo: 1.0000 xn: 0.5000 f(xn): 5.7812
f'(xn):    10.1250 xl: 0.5000 xo: 1.0000 xn: 0.7500 f(xn): 7.6948
f'(xn):     4.0898 xl: 0.7500 xo: 1.0000 xn: 0.8750 f(xn): 7.8439
f'(xn):    -2.1940 xl: 0.7500 xo: 0.8750 xn: 0.8125 f(xn): 7.8672
f'(xn):     1.3144 xl: 0.8125 xo: 0.8750 xn: 0.8438 f(xn): 7.8829
f'(xn):    -0.3397 xl: 0.8125 xo: 0.8438 xn: 0.8281 f(xn): 7.8815
f'(xn):     0.5113 xl: 0.8281 xo: 0.8438 xn: 0.8359 f(xn): 7.8839
Optimal solution found
x: 0.836
function value: 7.883868
```

10.3 Several selected CMPL applications

10.3.1 Calculating the Fibonacci sequence

By definition, the first Fibonacci sequence starts with the numbers 0 and 1, and each remaining number is the sum of the previous two.

$$a_{n+1}=a_n+a_{n-1} \quad ; \quad a_1=0, a_2=1, n\in\mathbb{N}$$

CMPL code to calculate the Fibonacci sequence (`fibonacci.cmpl`):

```
%arg -noOutput

parameters:
      # initialing the first elements
      F[1..2] := (0, 1);
      # Calculating the Fibonacci sequence until the 10th elelement
```

```
      {i:=3(1)10: F[i] := F[i-2] + F[i-1]; }
      echo "The Fibonacci sequence for the first 10 elements";
      {i:=1(1)10: echo "element " + i + ": " + F[i]; }
```

CMPL command:

```
cmpl fibonacci.cmpl
```

Calculated sequence:

```
The Fibonacci sequence for the first 10 elements
element 1: 0
element 2: 1
element 3: 1
element 4: 2
element 5: 3
element 6: 5
element 7: 8
element 8: 13
element 9: 21
element 10: 34
```

10.3.2 Calculating primes

A prime is defined as a natural number that has exactly two distinct natural number divisors: 1 and itself.

CMPL code to calculate the sequence of primes (`primes.cmpl`):

```
%arg -noOutput
parameters:
      # Initialing the first element
      P[1] := 2;
      # Calculating a prime sequence in the range 3 until 10
      {i := 3(1)10:
            #Test whether number is prime
            t := 1;
            {j := defset(P[]), t != 0:
                  t := i mod P[j];
            }
            # If number is prime, save then as prime number
            {t != 0:
                  P[count(P[]) + 1] := i;
            }
      }
      echo "The prime sequence in the range 3 until 10";
      {i:=defset(P[]): echo "element " + i + ": " + P[i]; }
```

CMPL command:

```
cmpl primes.cmpl
```

Calculated sequence:

```
The prime sequence in the range 3 until 10
element 1: 2
element 2: 3
element 3: 5
element 4: 7
```

11 Authors and Contact

Thomas Schleiff - Halle(Saale), Germany

Mike Steglich - Technical University of Applied Sciences Wildau, Germany - mike.steglich@th-wildau.de

- 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
  Bahnhofstraße
  D-15745 Wildau

  Tel.: +493375 / 508-365
  Fax.: +493375 / 508-566

  mike.steglich@th-wildau.de

- Support via mailing list

  Please use our CMPL mailing list hosted at COIN-OR http://list.coin-or.org/mailman/listinfo/Cmpl to get a direct support, to post bugs or to communicate wishes.

12 Appendix

12.1 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 original 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!. Doh 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

12.2 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

- Anderson/Sweeney/Williams/Martin: An Introduction to Management Science - Quantitative Approaches to Decision Making, 13th ed., South-Western, Cengage Learning 2008.
- Fourer/Gay/Kernigham: AMPL, 2nd ed., Thomson 2003.
- Gassmann/Ma/Martin/Sheng: Optimization Services 2.4 User's Manual, 2011.
- GLPK: GNU Linear Programming Kit Reference Manual for GLPK Version 4.45, 2010.
- Hillier/Lieberman: Introduction to Operations Research, 9th ed., McGraw-Hill Higher Education 2010.
- Rogge/Steglich: Betriebswirtschaftliche Entscheidungsmodelle zur Verfahrenswahl sowie Auflagen- und Lagerpolitiken, in: Diskussionsbeiträge zu Wirtschaftsinformatik und Operations Research 10/2007, Martin-Luther-Universität Halle-Wittenberg 2007.