CMPL

Coliop Mathematical Programming Language

coliop.org

Version 1.4.4

March 2011

Manual

M. Steglich, T. Schleiff

Table of content

1 About CMPL

CMPL (Coliop Mathematical Programming Language) is a mathematical programming language for modelling linear programming (LP) problems or mixed integer programming (MIP) 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 transforms the mathematical problem into MPS, Free-MPS or OSiL files which can be used with certain solvers. CMPL is also a part of Coliop3 which is an IDE (Integrated Development Environment) intended to solve LP and MIP problems.

CMPL is an open source project licensed under GPL. It is written in C and is available for all relevant operating systems. CMPL and Coliop3 are projects of 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/Coliop3 website (www.coliop.org).

2 Syntactic elements

2.1 General structure of a CMPL program

The structure of a CMPL program 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.

$$\begin{aligned}
&c^T \cdot x \rightarrow max / min \\
&s.t. \\
&A \cdot x \leq b \\
&x \geq 0
\end{aligned}$$

A CMPL program 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 optimal production planning. 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.
For 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[1..dim(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 of parameter expression also used for type casts
`max, min`	objective senses
`set, in, element, len, defset`	key words for sets
`max, min, dim, def, 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 - separation of variables in a variable definition
`:`	- 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 defined within the `parameters` section using a 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 type can be specified but is not necessary. Possible types are `real, integer, binary, string` and `set`. 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 `:=`.

Usage:

```
name [: type] := scalarExpression;
name[index] [: type] := scalarExpression;

name[[set]] [: type] := non-scalarExpression;
```

name	name of the parameter
type	optional specification of the type of parameter Possible types are `real, integer, binary, string` and `set`. If the type is not defined, the type of the parameter is given by the expression on the right hand side.
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.

	The elements of an array can also be sets. But it is not possible to mix set and non-set expressions in one array.
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.
non-scalarExpression	A non-scalar expression consists of a list of *scalarExpressions*. 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.

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[]: integer := (16, 45.4);`	Definition of a matrix with two integer values `a[1]`=16 and `a[2]`=45. Since `A[]` is defined as an integer matrix the real value 45.4 is transformed to the rounded value 45.
`a[, ] := (( 5.6, 7.7, 10.5 ),` `          ( 9.8, 4.2, 11.1 ));`	matrix with 2 rows and 3 columns
`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`

If a name is used for a defined parameter, different usages of this name with indices can only refer to parameter, 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 <u>not</u> as model variables. `a` can also not be used as a local parameter like a loop counter).

A special kind of parameter are 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 are 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, $0 \leq x \leq 100$
`x[1..5]: integer[10..20];`	vector with 5 elements, $10 \leq x_n \leq 20 \, ; n = 1(1)5$
`x[1..5,1..5,1..5]: real[0..];`	a three-dimensional array of real model variables with 125 elements identified by indices, $x_{i,j,k} \geq 0 \, ; i, j, k = 1(1)5$

`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 <u>not</u> 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 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 imbedded 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 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 cmpl variables 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. It is also possible to include the current matrix line number using a substitution expression imbedded in `$ $`.

Usage:

```
lineName:

lineName$1$:
lineName$2$:

lineName$k$:

loopName { controlStructure }
```

`$k$`	`$k$` 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 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 named `profit`

`constraints:`
`restriction_$1$: A[,] * x[] <=b[];`
	generates 3 lines named `restriction_2`
	`restriction_3`
	`restriction_4`

`restriction_$2$: A[,] * x[] <=b[];`	generates 3 lines named `restriction_1` `restriction_2` `restriction_3`
`{ i:=1(1)3: restriction_$i$: A[,]*x[]<=b[]; }`	generates 3 lines named `restriction_1` `restriction_2` `restriction_3`
`restriction { i:=1(1)3: A[,]*x[]<=b[]; }`	generates 3 lines named `restriction_1` `restriction_2` `restriction_3`

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
dim(vector)            #returns the length of a vector

def(parameter|variable)      #returns 1 (true) or 0 (false) whether or not a
                             #parameter or model variable is defined

def(array[ [,[, ..]] ])      #returns the length of the first free index
```

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

variable	a scalar parameter or model variable or a multidimensional array of parameters or model variables
array[*[,[, ..]]*]	array of parameters or variables with at least one free index

Examples:

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

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

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

3.3 Mathematical functions

In CMPL there are the following mathematical functions which can be used in expressions. With the exception of `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:

	value is:
`c[1] := sqrt( 36 );`	6.000000
`c[2] := exp(10);`	22026.465795
`c[3] := ln(10);`	2.302585
`c[4] := lg(10000);`	4.000000
`c[5] := ld(8);`	3.000000
`c[6] := rand(10);`	3.000000 (random number)
`c[7] := sin(2.5);`	0.598472
`c[8] := cos(7.7);`	0.153374
`c[9] := tan(10.1);`	0.800789
`c[10] := acos(0.1);`	1.470629
`c[11] := asin(0.4);`	0.411517
`c[12] := atan(1.1);`	0.832981
`c[13] := sinh(10);`	11013.232875
`c[14] := cosh(3);`	10.067662
`c[15] := tanh(15);`	1.000000
`c[16] := abs(12.55);`	12.000000
`c[17] := ceil(12.55);`	13.000000
`c[18] := floor(-12.55);`	-13.000000
`c[19] := round(12.4);`	12.000000
`c[20] := 35 div 4;`	8
`c[21] := 35 mod 4;`	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: real:= 6.666;`	
`echo integer(a);`	7

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

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

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

`e: string := "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 strin 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	
`ld`	integer
`lf`	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 `ld`: 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 `lf`: 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("%10.2lf", 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[1..dim(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 ),`
	`const-def.gen`

`       ( 9.8,4.2,11.1) );`	

`parameters:`
`    include "const-def.gen"`
`variables:`
`    x[1..dim(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.gen" 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:

$$\begin{array}{l} k_{11}\cdot x_1+k_{12}\cdot x_2+\ldots+k_{1\mathrm{n}}\cdot x_n \;\leq\; b_1 \\ k_{21}\cdot x_1+k_{22}\cdot x_2+\ldots+k_{2\mathrm{n}}\cdot x_n \;\leq\; b_2 \\ \vdots \\ k_{ml}\cdot x_1+k_{m2}\cdot x_2+\ldots+k_{mn}\cdot x_n \;\leq\; b_m \end{array}$$

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 use of 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 are 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; }`
`}`
	definition of the identity matrix with combined loops and two if-then clauses

`{i := 1(1)5, j := 1(1)5:`
`      {i = j:   A[i,j] := 1; |`
`       default: A[i,j] := 0; }`
`}`
	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 then 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;`	objective definition using a sum $20 \cdot x_1 + 10 \cdot x_2 + 10 \cdot x_3 \rightarrow max\,!$

`constraints:`
`      { i:=1..2:`
`          sum{j:=1..3: a[i,j] * x[j]}<= b[i];`
`      }`
	constraints definition using a sum
	$1 \cdot x_1 + 2 \cdot x_2 + 3 \cdot x_3 \leq 100$
	$4 \cdot x_1 + 5 \cdot x_2 + 6 \cdot x_3 \leq 100$

6.7 Implicit loops

As mentioned above it is possible to define objectives and constraints using control structures as loops. The syntax of these control structures is easy to understand and to use, but it follows the idea of programming languages. For a formulation of objectives and constraints in a more mathematical way it is simpler to use implicit loops. Implicit loops allow users to define objectives and constraints in a mathematical notation (e.g. matrix vector multiplication). All mathematical requirements are applied for implicit loops. Implicit loops are only possible in the `objectives` section and the `constraints` section.

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.

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;`	objective definition using implicit loops $20 \cdot x_1 + 10 \cdot x_2 + 10 \cdot x_3 \rightarrow max\,!$
`constraints:` `      a[, ] * x[] <=b[];`	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:

`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[];`
	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
	`a[, ] * x[]  <=b[] - d[];`

`0 <= x[1..3]+y[1..3]+z[2]<= b[1..3];`	implicit loops for a column vector.

`0 <= x[1] + y[1] + z[2] <= b[1];`
`0 <= x[2] + y[2] + z[2] <= b[2];`
`0 <= x[3] + y[3] + z[2] <= b[3];`
	equivalent formulation

`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..];`	

`objectives:` `      c[]T * x[] ->max;`	
`constraints:` `      a[,] * x[] + z[] <=b[];`	constraints definition using implicit loops $1\cdot x_1+2\cdot x_2+3\cdot x_3 \quad +z_1 \qquad \leq 90$ $4\cdot x_1+5\cdot x_2+6\cdot x_3 \qquad +z_2 \quad \leq 90$

7 Automatic code generating

7.1 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 generation is the equivalent transformation of variable products. If a CMPL program includes a variable product with at least one integer factor, CMPL will transform this non-linear form equivalent in a set of linear inequations.

7.2 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.

7.3 Equivalent transformations of Variable Products

In general a product of variables like $x \cdot y$ cannot be a part of an LP or MIP model, because such a variable product is a non-linear term. But it is possible to formulate an equivalent transformation using a set of specific inequations. The automatic generation of an equivalent transformation of a variable product is a special capability characteristic of CMPL.

7.3.1 Variable Products with at least one binary variable

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;
```

7.3.2 Variable Product with at least one integer variable

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 `n` stands for the number of binary positions needed for `XO-XU+1`):

```
variables:
      _x[1..n]: binary;
      _x_y[1..n]: 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)n: (2^(i-1))*_x[i]};
      x_y = XU*y + sum{i=1(1)n: (2^(i-1))*_x_y[i]};
```

```
{i = 1(1)n:
      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;
}
```

8 CMPL as command line interpreter

8.1 Usage

The CMPL package can be used as a stand-alone interpreter for CMPL files generating MPS or Free-MPS files for use in certain LP or MIP solvers.

Usage:

```
cmpl <options> <input file>
```

Options:

-i[<File>]	input file / reading from `stdin`
-m[<File>]	main output file (generated matrix) / out to `stdout`
-x[<File>]	export model in OSiL XML format in a file or `stdout`
-e[<File>]	output for error messages and warnings -e simple output to `stderr` -e<File> ouput in MprL XML format to file
-l[<File>]	output for replacements for products of variables
-s[<File>]	output for short statistical info
-p[<File>]	output for protocol
-gn	generation option: don't make reductions
-gf	generation option: constraints for products of variables follow the product
-cd	no warning at multiple parameter definition

-ca	no warning at deprecated '=' assignment
-ci<X>	mode for integer expressions (0 - 3), defaults to 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. -ci0 no range check -ci1 default, range check with a type change if necessary -ci2 range check with error message if necessary -ci3 Each numerical operation returns a real result
-fc<X>	format option: maximal length of comment; defaults to 60
-ff	format option: generate free MPS
-f%<format>	format option: format specifier for float number output, defaults to %f
-v	print version number

Examples:

`cmpl test.cmpl` `cmpl -itest.cmpl -mtest.mps`	reads the file `test.cmpl` and generates the MPS-file `test.mps`.
`cmpl -ff -itest.cmpl -mtest.mps`	reads the file `test.cmpl` and generates the Free-MPS-file `test.mps`.
`cmpl -itest.cmpl -xtest.osil`	reads the file `test.cmpl` and generates the OSiL-file `test.osil`.
`cmpl -etest.mprl test.cmpl`	reads the file `test.cmpl` and generates the MPS-file `test.mps`. The general status of the CMPL model `test.cmpl` is written in the MprL-File `test.mprl`.

8.2 Input and output file formats

8.2.1 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.
MprL	output file for the status of the results or errors of a CMPL model XML file in accordance with the MprL schema

8.2.2 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

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

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[1..dim(c[])]: real[0..];

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

constraints:
    machine$2$: A[,] * x[] <= b[];
```

8.2.3 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.mps
* OBJNAME profit
* OBJSENSE max
ROWS
 N  profit
 L  machine1
 L  machine2
COLUMNS
    x1        profit              1   machine1      5.600000
    x1        machine2     9.800000
    x2        profit              2   machine1      7.700000
    x2        machine2     4.200000
    x3        profit              3   machine1     10.500000
    x3        machine2    11.100000
RHS
    RHS       machine1           15   machine2            20
RANGES
BOUNDS
ENDATA
```

8.2.4 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.mps
* OBJNAME profit
* OBJSENSE max
ROWS
 N  profit
 L  machine1
 L  machine2
COLUMNS
 x1  profit  1 machine1  5.600000
 x1  machine2  9.800000
 x2  profit  2 machine1  7.700000
 x2  machine2  4.200000
 x3  profit  3 machine1  10.500000
 x3  machine2  11.100000
RHS
 RHS  machine1  15 machine2 20
```

```
RANGES
BOUNDS
ENDATA
```

8.2.5 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.

For more information please visit the project website of COIN-OR OS project. (https://projects.coin-or.org/OS or http://www.optimizationservices.org)

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/OSiL.xsd">
	<instanceHeader>
		<name>test.gen</name>
		<description>generated by CMPL v1.4.2</description>
	</instanceHeader>
	<instanceData>
		<variables numberOfVariables="3">
			<var name="x1"  type="C" lb="0"/>
			<var name="x2"  type="C" lb="0"/>
			<var name="x3"  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="machine1" ub="15"/>
			<con name="machine2" 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>
</osil>
```

8.2.6 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>
```

8.3 Using CMPL with several solvers

Since CMPL transforms a CMPL model into an MPS, a Free-MPS or an OSiL file the model can solve using most free or commercial solvers.

8.3.1 Coliop3

CMPL is recommended for use as an integral part of Coliop3. Coliop3 is an IDE (Integrated Development Environment) intended to solve linear programming (LP) problems and mixed integer programming (MIP) problems. This project contains CMPL and GLPK as solver for LP and MIP problems. CMPL and Coliop3 are projects of the Technical University of Applied Sciences Wildau and the Institute for Operations Research and Business Management at the Martin Luther University Halle-Wittenberg. Coliop3 and CMPL are open source projects licensed under GPL and available for all relevant operating systems. For more information please visit the Coliop3 project website: http://www.coliop.org.

8.3.2 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. It is a set of routines written in ANSI C and organized in the form of a callable library. For more information please visit the GLPK project website: http://www.gnu.org/software/glpk/.

Using CMPL with glpk:

```
cmpl -ff -icmplFilename -mmpsFileName

glpsol --max|min --output solutionName mpsFileName
```

cmplFilename	name of the CMPL file - CMPL input
mpsFileName	name of the MPS file - CMPL output
solutionName	name of the solution file - GLPK output

8.3.3 LPSolve

Mixed Integer Linear Programming (MILP) solver lp_solve solves pure linear, (mixed) integer/binary, semi-continuous and special ordered set (SOS) models. lp_solve is written in ANSI C and can be compiled on many different platforms including Linux and WINDOWS.

For more information please visit the GLPK project website: http://sourceforge.net/projects/lpsolve/.

Using CMPL with LPSolve:

```
cmpl -ff -icmplFilename -mmpsFileName

lp_solve -max|min -fmps mpsFileName -S4 solutionName
```

cmplFilename	name of the CMPL file - CMPL input
mpsFileName	name of the MPS file - CMPL output
solutionName	name of the solution file - LPSolve output

8.3.4 OSSolverService

The OSSolverService is a command line executable designed to pass problem instances in either OSiL, AMPL nl, or MPS format to solvers and get the optimization result back to be displayed either to standard output or a specified browser. The OSSolverService can be used to invoke a solver locally or on a remote server. The OSSolverService was developed by the COIN-OR community (COmputational INfrastructure for Operations Research - open source for the operations research community).

For more information please visit the GLPK project website: https://projects.coin-or.org/OS or http://www.optimizationservices.org.

Using CMPL with a local OSSolverService:

```
cmpl -icmplFilename -xosilFileName

OSSolverService -osil osilFileName -osrl osrlFileName
```

cmplFilename	name of the CMPL file - CMPL input
osilFileName	name of the OSiL file - CMPL output
osrlFileName	name of the OSrL solution file - OSSolverService output

9 Examples

9.1 Selected decision problems

9.1.1 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 based on Fourer/Gay/Kernigham: AMPL, 2nd ed., Thomson 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 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[];
```

Solution (solved with glpk):

```
Problem:    diet.mps
Rows:       5
Columns:    8 (8 integer, 0 binary)
Non-zeros:  39
Status:     INTEGER OPTIMAL
Objective:  cost = 101.14 (MINimum)
```

```
   No.   Row name        Activity     Lower bound   Upper bound
------ ------------    ------------- ------------- -------------
     1 cost                 101.14
     2 A                      1500           700
     3 B1                     1330           700
     4 B2                      860           700
     5 C                       700           700
   No. Column name       Activity     Lower bound   Upper bound

   No. Column name       Activity     Lower bound   Upper bound
------ ------------    ------------- ------------- -------------
     1 x_BEEF       *            2             2            10
     2 x_CHK        *            8             2            10
     3 x_FISH       *            2             2            10
     4 x_HAM        *            2             2            10
     5 x_MCH        *           10             2            10
     6 x_MTL        *           10             2            10
     7 x_SPG        *           10             2            10
     8 x_TUR        *            2             2            10
```

9.1.2 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:

$$
\begin{aligned}
&75\cdot x_1+80\cdot x_2+50\cdot x_3 \rightarrow max\,!\\
&s.t.\\
&8\cdot x_1+15\cdot x_2+12\cdot x_3 \le 1{,}000\\
&15\cdot x_1+10\cdot x_2+8\cdot x_3 \le 1{,}000\\
&\\
&x_1 \in \{0,1,\ldots,250\}\\
&x_2 \in \{0,1,\ldots,240\}\\
&x_3 \in \{0,1,\ldots,250\}
\end{aligned}
$$

The CMPL model 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:=1(1)dim(price[]):  c[j] := price[j]-costs[j]; }

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

variables:
      x[1..dim(price[])]: integer;

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

constraints:
      res_$2$: a[,] * x[] <= b[];
      0<=x[]<=xMax[];
```

Solution (solved with glpk):

```
Problem:    production-mix.mps
Rows:       3
Columns:    3 (3 integer, 0 binary)
Non-zeros:  9
Status:     INTEGER OPTIMAL
Objective:  profit = 6395 (MAXimum)

   No.   Row name        Activity     Lower bound   Upper bound
------ ------------    ------------- ------------- -------------
     1 profit                   6395
     2 res_1                     999                        1000
     3 res_2                     985                        1000

   No. Column name       Activity     Lower bound   Upper bound
------ ------------    ------------- ------------- -------------
     1 x1           *             33             0           250
     2 x2           *             49             0           240
     3 x3           *              0             0           250
```

9.1.3 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 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:=1(1)dim(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[1..dim(c[])] : binary;
objectives:
      profit: c[]T  * x[] - FC[]T * y[] ->max;
constraints:
      res_$2$: a[,] * x[] <= b[];
      {j:=1(1)dim(c[]): xMin[j] * y[j] <= x[j] <= xMax[j] * y[j]; }
```

Solution (solved with glpk):

```
Problem:    production-mix.mps
Rows:       9
Columns:    6 (6 integer, 3 binary)
Non-zeros:  24
Status:     INTEGER OPTIMAL
Objective:  profit = 4880 (MAXimum)

   No.   Row name        Activity     Lower bound   Upper bound
------ ------------    ------------- ------------- -------------
     1 profit                   4880
     2 res_1                     990                        1000
     3 res_2                     660                        1000
     4 line_4                      0                           0
     5 line_5                      0                           0
     6 line_6                    -21                           0
     7 line_7                   -174                           0
     8 line_8                      0                           0
     9 line_9                      0                           0

   No. Column name       Activity     Lower bound   Upper bound
------ ------------    ------------- ------------- -------------
     1 x1           *              0             0           250
     2 x2           *             66             0           240
     3 x3           *              0             0           250
```

```
   4 y1          *          0          0          1
   5 y2          *          1          0          1
   6 y3          *          0          0          1
```

9.1.4 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:

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

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 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;
```

Solution (solved with glpk):

```
Problem:    knapsack-max-basic.mps
Rows:       2
Columns:    10 (10 integer, 10 binary)
Non-zeros:  20
Status:     INTEGER OPTIMAL
Objective:  sales = 700 (MAXimum)

   No.   Row name        Activity     Lower bound   Upper bound
------ ------------    ------------- ------------- -------------
     1 sales                    700
     2 weigth                    59                           60

   No. Column name       Activity     Lower bound   Upper bound
------ ------------    ------------- ------------- -------------
     1 x1           *              1             0             1
     2 x2           *              1             0             1
     3 x3           *              1             0             1
     4 x4           *              1             0             1
     5 x5           *              0             0             1
     6 x6           *              1             0             1
     7 x7           *              0             0             1
     8 x8           *              0             0             1
     9 x9           *              1             0             1
    10 x10          *              1             0             1
```

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 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;
```

Solution (solved with glpk):

```
Problem:     knapsack-min-basic.mps
Rows:        2
Columns:     10 (10 integer, 10 binary)
Non-zeros:   20
Status:      INTEGER OPTIMAL
Objective:   weight = 47 (MINimum)

   No.   Row name        Activity     Lower bound   Upper bound
------ ------------    ------------- ------------- -------------
     1 weight                     47
     2 sales                     630           600

   No. Column name       Activity     Lower bound   Upper bound
------ ------------    ------------- ------------- -------------
     1 x1           *              1             0             1
     2 x2           *              1             0             1
     3 x3           *              0             0             1
     4 x4           *              1             0             1
     5 x5           *              0             0             1
     6 x6           *              0             0             1
     7 x7           *              0             0             1
     8 x8           *              0             0             1
     9 x9           *              1             0             1
    10 x10          *              1             0             1
```

9.1.5 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

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

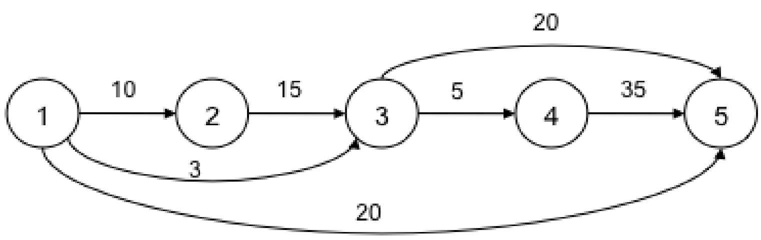
The general model can be formulated as follows:

$$\sum_{h=1}^{n}\sum_{\substack{i=1\\ i\neq h}}^{n}\sum_{j=1}^{n}\sum_{\substack{k=1\\ i\neq j}}^{n} t_{hi}\cdot d_{jk}\cdot x_{hj}\cdot x_{ik}\rightarrow min!$$
$$s.t.$$
$$\sum_{j=1}^{n} x_{hj}=1 \quad ;h=1(1)n$$
$$\sum_{h=1}^{n} x_{hj}=1 \quad ;j=1(1)n$$
$$x_{hj}\in\{0,1\} \quad ;h=1(1)n,\ j=1(1)n$$

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	1	M	1	2	3
3	2	1	M	1	2
4	3	2	1	M	1
5	4	3	2	1	M

The CMPL model can be formulated as follows:

```
parameters:
      n:=5;
      M:=1000;
      d[,]:= ( ( M, 1, 2, 3, 4),
             ( 1, M, 1, 2, 3),
             ( 2, 1, M, 1, 2),
             ( 3, 2, 1, M, 1),
             ( 4, 3, 2, 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]; }
      }
      { h:=1(1)n: sos1_$h$: sum{ j:=1(1)n: x[h,j] } = 1; }
      { j:=1(1)n: sos2_$j$: sum{ h:=1(1)n: x[h,j] } = 1; }
```

Solution (solved with glpk):

```
Problem:    quadratic-assignment.mps
Rows:       711
Columns:    375 (25 integer, 25 binary)
Non-zeros:  1800
Status:     INTEGER OPTIMAL
Objective:  costs = 155 (MINimum)

   No.   Row name        Activity     Lower bound   Upper bound
------ ------------    ------------- ------------- -------------
     1 costs                   155
   ...
```

```
   177 sos1_1                        1             1             =
   178 sos1_2                        1             1             =
   179 sos1_3                        1             1             =
   180 sos1_4                        1             1             =
   181 sos1_5                        1             1             =
   182 sos2_1                        1             1             =
   183 sos2_2                        1             1             =
   184 sos2_3                        1             1             =
   185 sos2_4                        1             1             =
   186 sos2_5                        1             1             =
 ...

   No. Column name       Activity     Lower bound   Upper bound
------ ------------    ------------- ------------- -------------
   ...
   351 x1_1         *              0             0             1
   352 x1_2         *              0             0             1
   353 x1_3         *              1             0             1
   354 x1_4         *              0             0             1
   355 x1_5         *              0             0             1
   356 x2_1         *              0             0             1
   357 x2_2         *              0             0             1
   358 x2_3         *              0             0             1
   359 x2_4         *              0             0             1
   360 x2_5         *              1             0             1
   361 x3_1         *              0             0             1
   362 x3_2         *              0             0             1
   363 x3_3         *              0             0             1
   364 x3_4         *              1             0             1
   365 x3_5         *              0             0             1
   366 x4_1         *              1             0             1
   367 x4_2         *              0             0             1
   368 x4_3         *              0             0             1
   369 x4_4         *              0             0             1
   370 x4_5         *              0             0             1
   371 x5_1         *              0             0             1
   372 x5_2         *              1             0             1
   373 x5_3         *              0             0             1
   374 x5_4         *              0             0             1
   375 x5_5         *              0             0             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			

9.2 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.

9.2.1 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 section 9.1.4 a CMPL model to maximize the total sales relative to capacity can be formulated as follows.

Model 1: maximize the total sales

```
include "knapsack-data.cmpl"

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

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

```
                        val[j]:=0;
                        break j;
            }
      }
}
echo "Solution found";
echo "Optimal total sales: "+ sumSales;
echo "Total weight : " + sumWeight;
{j in boxes: echo "x_"+ j + ": " + x[j]; }
```

Solution (solved with CMPL):

```
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 51 but good enough as an approximate solution.

Model 2: minimize the total weight

```

include "knapsack-data.cmpl"

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

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

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

Solution (solved with CMPL):

```
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 9.1.4 .

9.2.2 Finding the maximum of a negative convex function with the golden ratio method

One of the alternative methods for finding the maximum of a negative convex function is the golden ratio method. (see e.g. http://math.fullerton.edu/mathews/n2003/GoldenRatioSearchMod.html)

A CMPL program to find the maximum of $f(x)=-0.5\cdot x^2-e^{-x}$ can be formulated as follows:

```
parameters:
      #golden ratio delta
      d:=0.382;
      #distance epsilon
      e:=0.0001;

      #initial solution
      a= 0;
      b= 1;

      l:= a + d*(b-a);
      m:= a+ (1-d)*(b-a);
```

```
	Fl:= -0.5 * l^2 - exp(-l);
	Fm:= -0.5 * m^2 - exp(-m);

	{ (b-a)>=e :
		{ Fl<Fm:
			a:=l;
			l:=m;
			m:=a+(1-d)*(b-a);
			Fl:=Fm;
			Fm:=-0.5 * m^2 - exp(-m);
		}
		{ Fl>=Fm:
			b:=m;
			m:=l;
			l:=a+d*(b-a);
			Fm:=Fl;
			Fl:= -0.5 * l^2 - exp(-l);
		}
		repeat;
	}
	echo "Optimal solution found";
	x:=round(b*1000)/1000;
	echo "x: "+ format("%2.3lf",x);
	echo "function value: " + (-0.5 * x^2 - exp(-x));
```

Solution (solved with CMPL):

```
Optimal solution found
x: 0.562
function value: -0.727990
```

9.3 Several selected CMPL applications

9.3.1 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:

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

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
```

9.3.2 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:

```
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 := 1(1)dim(P[]), t != 0:
			t := i mod P[j];
		}
		# If number is prime, save then as prime number
		{t != 0:
			P[dim(P[]) + 1] = i;
		}
	}
	echo "The prime sequence in the range 3 until 10";
	{i:=1(1)dim(P[]): echo "element " + i + ": " + P[i]; }
```

Calculated sequence:

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

10 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 Prof. Dr. Mike Steglich

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