Modeling Language GNU MathProg

Language Reference

for GLPK Version 4.58

(DRAFT, February 2016)

Contents

Chapter 1

Introduction

GNU MathProg is a modeling language intended for describing linear mathematical programming models.[1]

Model descriptions written in the GNU MathProg language consist of a set of statements and data blocks constructed by the user from the language elements described in this document.

In a process called *translation*, a program called the *model translator* analyzes the model description and translates it into internal data structures, which may be then used either for generating mathematical programming problem instance or directly by a program called the *solver* to obtain numeric solution of the problem.

1.1 Linear programming problem

In MathProg the linear programming (LP) problem is stated as follows:

minimize (or maximize)

$$z = c_1x_1 + c_2x_2 + \ldots + c_nx_n + c_0 \tag{1.1}$$

subject to linear constraints

$$\begin{array}{r@{\ }c@{\ }r@{\ }c@{\ }r@{\ }c@{\ }r@{\ }c@{\ }l}
L_1 & \leq & a_{11}x_1 & + & a_{12}x_2 & + \ldots + & a_{1n}x_n & \leq & U_1 \\
L_2 & \leq & a_{21}x_1 & + & a_{22}x_2 & + \ldots + & a_{2n}x_n & \leq & U_2 \\
 & & \multicolumn{5}{c}{\cdot\ \cdot\ \cdot\ \cdot\ \cdot\ \cdot\ \cdot\ \cdot\ \cdot\ \cdot} & & \\
L_m & \leq & a_{m1}x_1 & + & a_{m2}x_2 & + \ldots + & a_{mn}x_n & \leq & U_m
\end{array} \tag{1.2}$$

and bounds of variables

$$\begin{array}{rcccl}
l_1 & \leq & x_1 & \leq & u_1 \\
l_2 & \leq & x_2 & \leq & u_2 \\
 & \cdot & \cdot & \cdot & \\
l_n & \leq & x_n & \leq & u_n
\end{array} \tag{1.3}$$

[1]The GNU MathProg language is a subset of the AMPL language. Its GLPK implementation is mainly based on the paper: *Robert Fourer*, *David M. Gay*, and *Brian W. Kernighan*, "A Modeling Language for Mathematical Programming." *Management Science* 36 (1990), pp. 519-54.

where x_1, x_2, ..., x_n are variables; z is the objective function; c_1, c_2, ..., c_n are objective coefficients; c_0 is the constant term ("shift") of the objective function; a_{11}, a_{12}, ..., a_{mn} are constraint coefficients; L_1, L_2, ..., L_m are lower constraint bounds; U_1, U_2, ..., U_m are upper constraint bounds; l_1, l_2, ..., l_n are lower bounds of variables; u_1, u_2, ..., u_n are upper bounds of variables.

Bounds of variables and constraint bounds can be finite as well as infinite. Besides, lower bounds can be equal to corresponding upper bounds. Thus, the following types of variables and constraints are allowed:

$-\infty < x < +\infty$	Free (unbounded) variable
$l \leq x < +\infty$	Variable with lower bound
$-\infty < x \leq u$	Variable with upper bound
$l \leq x \leq u$	Double-bounded variable
$l = x = u$	Fixed variable
$-\infty < \sum a_j x_j < +\infty$	Free (unbounded) linear form
$L \leq \sum a_j x_j < +\infty$	Inequality constraint "greater than or equal to"
$-\infty < \sum a_j x_j \leq U$	Inequality constraint "less than or equal to"
$L \leq \sum a_j x_j \leq U$	Double-bounded inequality constraint
$L = \sum a_j x_j = U$	Equality constraint

In addition to pure LP problems MathProg also allows mixed integer linear programming (MIP) problems, where some or all variables are restricted to be integer or binary.

1.2 Model objects

In MathProg the model is described in terms of sets, parameters, variables, constraints, and objectives, which are called *model objects*.

The user introduces particular model objects using the language statements. Each model object is provided with a symbolic name which uniquely identifies the object and is intended for referencing purposes.

Model objects, including sets, can be multidimensional arrays built over indexing sets. Formally, n-dimensional array A is the mapping:

$$A : \Delta \to \Xi, \tag{1.4}$$

where $\Delta \subseteq S_1 \times \ldots \times S_n$ is a subset of the Cartesian product of indexing sets, Ξ is a set of array members. In MathProg the set Δ is called the *subscript domain*. Its members are n-tuples $(i_1, \ldots, i_n)$, where $i_1 \in S_1$, ..., $i_n \in S_n$.

If $n = 0$, the Cartesian product above has exactly one member (namely, 0-tuple), so it is convenient to think scalar objects as 0-dimensional arrays having one member.

The type of array members is determined by the type of corresponding model object as follows:

Model object	Array member
Set	Elemental plain set
Parameter	Number or symbol
Variable	Elemental variable
Constraint	Elemental constraint
Objective	Elemental objective

In order to refer to a particular object member the object should be provided with *subscripts*. For example, if a is a 2-dimensional parameter defined over $I \times J$, a reference to its particular member can be written as $a[i, j]$, where $i \in I$ and $j \in J$. It is understood that scalar objects being 0-dimensional need no subscripts.

1.3 Structure of model description

It is sometimes desirable to write a model which, at various points, may require different data for each problem instance to be solved using that model. For this reason in MathProg the model description consists of two parts: the *model section* and the *data section*.

The model section is a main part of the model description that contains declarations of model objects and is common for all problems based on the corresponding model.

The data section is an optional part of the model description that contains data specific for a particular problem instance.

Depending on what is more convenient the model and data sections can be placed either in one file or in two separate files. The latter feature allows having arbitrary number of different data sections to be used with the same model section.

Chapter 2

Coding model description

The model description is coded in a plain text format using ASCII character set. Characters valid in the model description are the following:

— alphabetic characters:
`A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`
`a b c d e f g h i j k l m n o p q r s t u v w x y z _`

— numeric characters:
`0 1 2 3 4 5 6 7 8 9`

— special characters:
`! " # & ' ( ) * + , - . / : ; < = > [ ] ^ { | } ~`

— white-space characters:
`SP HT CR NL VT FF`

Within string literals and comments any ASCII characters (except control characters) are valid.

White-space characters are non-significant. They can be used freely between lexical units to improve readability of the model description. They are also used to separate lexical units from each other if there is no other way to do that.

Syntactically model description is a sequence of lexical units in the following categories:

— symbolic names;

— numeric literals;

— string literals;

— keywords;

— delimiters;

— comments.

The lexical units of the language are discussed below.

2.1 Symbolic names

A *symbolic name* consists of alphabetic and numeric characters, the first of which should be alphabetic. All symbolic names are distinct (case sensitive).

Examples

```
alpha123
This_is_a_name
_P123_abc_321
```

Symbolic names are used to identify model objects (sets, parameters, variables, constraints, objectives) and dummy indices.

All symbolic names (except names of dummy indices) should be unique, i.e. the model description should have no objects with identical names. Symbolic names of dummy indices should be unique within the scope, where they are valid.

2.2 Numeric literals

A *numeric literal* has the form *xx*`E`*syy*, where *xx* is a number with optional decimal point, *s* is the sign `+` or `-`, *yy* is a decimal exponent. The letter `E` is case insensitive and can be coded as `e`.

Examples

```
123
3.14159
56.E+5
.78
123.456e-7
```

Numeric literals are used to represent numeric quantities. They have obvious fixed meaning.

2.3 String literals

A *string literal* is a sequence of arbitrary characters enclosed either in single quotes or in double quotes. Both these forms are equivalent.

If a single quote is part of a string literal enclosed in single quotes, it should be coded twice. Analogously, if a double quote is part of a string literal enclosed in double quotes, it should be coded twice.

Examples

```
'This is a string'
"This is another string"
'That''s all'
"""Hello there,"" said the captain."
```

String literals are used to represent symbolic quantities.

2.4 Keywords

A *keyword* is a sequence of alphabetic characters and possibly some special characters.

All keywords fall into two categories: *reserved keywords*, which cannot be used as symbolic names, and *non-reserved keywords*, which are recognized by context and therefore can be used as symbolic names.

The reserved keywords are the following:

`and`	`else`	`mod`	`union`
`by`	`if`	`not`	`within`
`cross`	`in`	`or`	
`diff`	`inter`	`symdiff`	
`div`	`less`	`then`	

Non-reserved keywords are described in following sections.

All the keywords have fixed meaning, which will be explained on discussion of corresponding syntactic constructions, where the keywords are used.

2.5 Delimiters

A *delimiter* is either a single special character or a sequence of two special characters as follows:

`+`	`**`	`<=`	`>`	`&&`	`:`	`|`	`[`	`>>`
`-`	`^`	`=`	`<>`	`||`	`;`	`~`	`]`	`<-`
`*`	`&`	`==`	`!=`	`.`	`:=`	`(`	`{`	
`/`	`<`	`>=`	`!`	`,`	`..`	`)`	`}`	

If the delimiter consists of two characters, there should be no spaces between the characters.

All the delimiters have fixed meaning, which will be explained on discussion corresponding syntactic constructions, where the delimiters are used.

2.6 Comments

For documenting purposes the model description can be provided with *comments*, which may have two different forms. The first form is a *single-line comment*, which begins with the character `#` and extends until end of line. The second form is a *comment sequence*, which is a sequence of any characters enclosed within `/*` and `*/`.

Examples

```
param n := 10; # This is a comment
/* This is another comment */
```

Comments are ignored by the model translator and can appear anywhere in the model description, where white-space characters are allowed.

Chapter 3

Expressions

An *expression* is a rule for computing a value. In model description expressions are used as constituents of certain statements.

In general case expressions consist of operands and operators.

Depending on the type of the resultant value all expressions fall into the following categories:

— numeric expressions;

— symbolic expressions;

— indexing expressions;

— set expressions;

— logical expressions;

— linear expressions.

3.1 Numeric expressions

A *numeric expression* is a rule for computing a single numeric value represented as a floating-point number.

The primary numeric expression may be a numeric literal, dummy index, unsubscripted parameter, subscripted parameter, built-in function reference, iterated numeric expression, conditional numeric expression, or another numeric expression enclosed in parentheses.

Examples

`1.23`	(numeric literal)
`j`	(dummy index)
`time`	(unsubscripted parameter)
`a['May 2003',j+1]`	(subscripted parameter)
`abs(b[i,j])`	(function reference)

```
sum{i in S diff T} alpha[i] * b[i,j]   (iterated expression)
if i in I then 2 * p else q[i+1]       (conditional expression)
(b[i,j] + .5 * c)                      (parenthesized expression)
```

More general numeric expressions containing two or more primary numeric expressions may be constructed by using certain arithmetic operators.

Examples

```
j+1
2 * a[i-1,j+1] - b[i,j]
sum{j in J} a[i,j] * x[j] + sum{k in K} b[i,k] * x[k]
(if i in I and p >= 1 then 2 * p else q[i+1]) / (a[i,j] + 1.5)
```

3.1.1 Numeric literals

If the primary numeric expression is a numeric literal, the resultant value is obvious.

3.1.2 Dummy indices

If the primary numeric expression is a dummy index, the resultant value is current value assigned to that dummy index.

3.1.3 Unsubscripted parameters

If the primary numeric expression is an unsubscripted parameter (which should be 0-dimensional), the resultant value is the value of that parameter.

3.1.4 Subscripted parameters

The primary numeric expression, which refers to a subscripted parameter, has the following syntactic form:

$$\textit{name}\texttt{[}i_1\texttt{,}\ i_2\texttt{,}\ \dots\texttt{,}\ i_n\texttt{]}$$

where *name* is the symbolic name of the parameter, i_1, i_2, ..., i_n are subscripts.

Each subscript should be a numeric or symbolic expression. The number of subscripts in the subscript list should be the same as the dimension of the parameter with which the subscript list is associated.

Actual values of subscript expressions are used to identify a particular member of the parameter that determines the resultant value of the primary expression.

3.1.5 Function references

In MathProg there exist the following built-in functions which may be used in numeric expressions:

`abs(`x`)`	$\|x\|$, absolute value of x
`atan(`x`)`	$\arctan x$, principal value of the arc tangent of x (in radians)
`atan(`y`, `x`)`	$\arctan y/x$, principal value of the arc tangent of y/x (in radians). In this case the signs of both arguments y and x are used to determine the quadrant of the resultant value
`card(`X`)`	$\|X\|$, cardinality (the number of elements) of set X
`ceil(`x`)`	$\lceil x \rceil$, smallest integer not less than x ("ceiling of x")
`cos(`x`)`	$\cos x$, cosine of x (in radians)
`exp(`x`)`	e^x, base-e exponential of x
`floor(`x`)`	$\lfloor x \rfloor$, largest integer not greater than x ("floor of x")
`gmtime()`	the number of seconds elapsed since 00:00:00 Jan 1, 1970, Coordinated Universal Time (for details see Section B.1, page 56)
`length(`s`)`	$\|s\|$, length of character string s
`log(`x`)`	$\log x$, natural logarithm of x
`log10(`x`)`	$\log_{10} x$, common (decimal) logarithm of x
`max(`x_1`, `x_2`, ..., `x_n`)`	the largest of values $x_1, x_2, \ldots, x_n$
`min(`x_1`, `x_2`, ..., `x_n`)`	the smallest of values $x_1, x_2, \ldots, x_n$
`round(`x`)`	rounding x to nearest integer
`round(`x`, `n`)`	rounding x to n fractional decimal digits
`sin(`x`)`	$\sin x$, sine of x (in radians)
`sqrt(`x`)`	$\sqrt{x}$, non-negative square root of x
`str2time(`s`, `f`)`	converting character string s to calendar time (for details see Section B.2, page 56)
`tan(`x`)`	$\tan x$, tangent of x (in radians)
`trunc(`x`)`	truncating x to nearest integer
`trunc(`x`, `n`)`	truncating x to n fractional decimal digits
`Irand224()`	generating pseudo-random integer uniformly distributed in $[0, 2^{24})$
`Uniform01()`	generating pseudo-random number uniformly distributed in $[0, 1)$
`Uniform(`a`, `b`)`	generating pseudo-random number uniformly distributed in $[a, b)$
`Normal01()`	generating Gaussian pseudo-random variate with $\mu = 0$ and $\sigma = 1$
`Normal(`μ`, `σ`)`	generating Gaussian pseudo-random variate with given μ and σ

Arguments of all built-in functions, except `card`, `length`, and `str2time`, should be numeric expressions. The argument of `card` should be a set expression. The argument of `length` and both arguments of `str2time` should be symbolic expressions.

The resultant value of the numeric expression, which is a function reference, is the result of applying the function to its argument(s).

Note that each pseudo-random generator function has a latent argument (i.e. some internal state), which is changed whenever the function has been applied. Thus, if the function is applied repeatedly even to identical arguments, due to the side effect different resultant values are always produced.

3.1.6 Iterated expressions

An *iterated numeric expression* is a primary numeric expression, which has the following syntactic form:

iterated-operator indexing-expression integrand

where *iterated-operator* is the symbolic name of the iterated operator to be performed (see below), *indexing-expression* is an indexing expression which introduces dummy indices and controls iterating, *integrand* is a numeric expression that participates in the operation.

In MathProg there exist four iterated operators, which may be used in numeric expressions:

`sum`	summation	$\displaystyle\sum_{(i_1,\dots,i_n)\in\Delta} f(i_1,\dots,i_n)$
`prod`	production	$\displaystyle\prod_{(i_1,\dots,i_n)\in\Delta} f(i_1,\dots,i_n)$
`min`	minimum	$\displaystyle\min_{(i_1,\dots,i_n)\in\Delta} f(i_1,\dots,i_n)$
`max`	maximum	$\displaystyle\max_{(i_1,\dots,i_n)\in\Delta} f(i_1,\dots,i_n)$

where i_1, ..., i_n are dummy indices introduced in the indexing expression, Δ is the domain, a set of n-tuples specified by the indexing expression which defines particular values assigned to the dummy indices on performing the iterated operation, $f(i_1,\dots,i_n)$ is the integrand, a numeric expression whose resultant value depends on the dummy indices.

The resultant value of an iterated numeric expression is the result of applying of the iterated operator to its integrand over all n-tuples contained in the domain.

3.1.7 Conditional expressions

A *conditional numeric expression* is a primary numeric expression, which has one of the following two syntactic forms:

`if` b `then` x `else` y

`if` b `then` x

where b is an logical expression, x and y are numeric expressions.

The resultant value of the conditional expression depends on the value of the logical expression that follows the keyword `if`. If it takes on the value *true*, the value of the conditional expression is the value of the expression that follows the keyword `then`. Otherwise, if the logical expression takes on the value *false*, the value of the conditional expression is the value of the expression that follows the keyword *else*. If the second, reduced form of the conditional expression is used and the logical expression takes on the value *false*, the resultant value of the conditional expression is zero.

3.1.8 Parenthesized expressions

Any numeric expression may be enclosed in parentheses that syntactically makes it a primary numeric expression.

Parentheses may be used in numeric expressions, as in algebra, to specify the desired order in which operations are to be performed. Where parentheses are used, the expression within the parentheses is evaluated before the resultant value is used.

The resultant value of the parenthesized expression is the same as the value of the expression enclosed within parentheses.

3.1.9 Arithmetic operators

In MathProg there exist the following arithmetic operators, which may be used in numeric expressions:

`+` *x*	unary plus
`-` *x*	unary minus
x `+` *y*	addition
x `-` *y*	subtraction
x `less` *y*	positive difference (if $x < y$ then 0 else $x - y$)
x `*` *y*	multiplication
x `/` *y*	division
x `div` *y*	quotient of exact division
x `mod` *y*	remainder of exact division
x `**` *y*, *x* `^` *y*	exponentiation (raising to power)

where x and y are numeric expressions.

If the expression includes more than one arithmetic operator, all operators are performed from left to right according to the hierarchy of operations (see below) with the only exception that the exponentiaion operators are performed from right to left.

The resultant value of the expression, which contains arithmetic operators, is the result of applying the operators to their operands.

3.1.10 Hierarchy of operations

The following list shows the hierarchy of operations in numeric expressions:

Operation	Hierarchy
Evaluation of functions (`abs`, `ceil`, etc.)	1st
Exponentiation (`**`, `^`)	2nd
Unary plus and minus (`+`, `-`)	3rd
Multiplication and division (`*`, `/`, `div`, `mod`)	4th
Iterated operations (`sum`, `prod`, `min`, `max`)	5th
Addition and subtraction (`+`, `-`, `less`)	6th
Conditional evaluation (`if` ... `then` ... `else`)	7th

This hierarchy is used to determine which of two consecutive operations is performed first. If the first operator is higher than or equal to the second, the first operation is performed. If it is not, the second operator is compared to the third, etc. When the end of the expression is reached, all of the remaining operations are performed in the reverse order.

3.2 Symbolic expressions

A *symbolic expression* is a rule for computing a single symbolic value represented as a character string.

The primary symbolic expression may be a string literal, dummy index, unsubscripted parameter, subscripted parameter, built-in function reference, conditional symbolic expression, or another symbolic expression enclosed in parentheses.

It is also allowed to use a numeric expression as the primary symbolic expression, in which case the resultant value of the numeric expression is automatically converted to the symbolic type.

Examples

`'May 2003'`	(string literal)
`j`	(dummy index)
`p`	(unsubscripted parameter)
`s['abc',j+1]`	(subscripted parameter)
`substr(name[i],k+1,3)`	(function reference)
`if i in I then s[i,j] & "..." else t[i+1]`	(conditional expression)
`((10 * b[i,j]) & '.bis')`	(parenthesized expression)

More general symbolic expressions containing two or more primary symbolic expressions may be constructed by using the concatenation operator.

Examples

```
'abc[' & i & ',' & j & ']'
"from " & city[i] " to " & city[j]
```

The principles of evaluation of symbolic expressions are completely analogous to the ones given for numeric expressions (see above).

3.2.1 Function references

In MathProg there exist the following built-in functions which may be used in symbolic expressions:

`substr(`s`, `x`)`	substring of s starting from position x
`substr(`s`, `x`, `y`)`	substring of s starting from position x and having length y
`time2str(`t`, `f`)`	converting calendar time to character string (for details see Section B.3, page 58)

The first argument of `substr` should be a symbolic expression while its second and optional third arguments should be numeric expressions.

The first argument of `time2str` should be a numeric expression, and its second argument should be a symbolic expression.

The resultant value of the symbolic expression, which is a function reference, is the result of applying the function to its arguments.

3.2.2 Symbolic operators

Currently in MathProg there exists the only symbolic operator:

```
s & t
```

where s and t are symbolic expressions. This operator means concatenation of its two symbolic operands, which are character strings.

3.2.3 Hierarchy of operations

The following list shows the hierarchy of operations in symbolic expressions:

Operation	Hierarchy
Evaluation of numeric operations	1st-7th
Concatenation (`&`)	8th
Conditional evaluation (`if` ... `then` ... `else`)	9th

This hierarchy has the same meaning as was explained above for numeric expressions (see Subsection 3.1.10, page 16).

3.3 Indexing expressions and dummy indices

An *indexing expression* is an auxiliary construction, which specifies a plain set of n-tuples and introduces dummy indices. It has two syntactic forms:

$$\{\ entry_1\texttt{,}\ entry_2\texttt{,}\ \ldots\texttt{,}\ entry_m\ \}$$

$$\{\ entry_1\texttt{,}\ entry_2\texttt{,}\ \ldots\texttt{,}\ entry_m\ \texttt{:}\ predicate\ \}$$

where $entry_1$, $entry_2$, ..., $entry_m$ are indexing entries, *predicate* is a logical expression that specifies an optional predicate (logical condition).

Each *indexing entry* in the indexing expression has one of the following three forms:

$$i\ \texttt{in}\ S$$

$$(i_1\texttt{,}\ i_2\texttt{,}\ \ldots\texttt{,}i_n)\ \texttt{in}\ S$$

$$S$$

where i_1, i_2, ..., i_n are indices, S is a set expression (discussed in the next section) that specifies the basic set.

The number of indices in the indexing entry should be the same as the dimension of the basic set S, i.e. if S consists of 1-tuples, the first form should be used, and if S consists of n-tuples, where $n > 1$, the second form should be used.

If the first form of the indexing entry is used, the index i can be a dummy index only (see below). If the second form is used, the indices i_1, i_2, $\dots$, i_n can be either dummy indices or some numeric or symbolic expressions, where at least one index should be a dummy index. The third, reduced form of the indexing entry has the same effect as if there were i (if S is 1-dimensional) or i_1, i_2, $\dots$, i_n (if S is n-dimensional) all specified as dummy indices.

A *dummy index* is an auxiliary model object, which acts like an individual variable. Values assigned to dummy indices are components of n-tuples from basic sets, i.e. some numeric and symbolic quantities.

For referencing purposes dummy indices can be provided with symbolic names. However, unlike other model objects (sets, parameters, etc.) dummy indices need not be explicitly declared. Each *undeclared* symbolic name being used in the indexing position of an indexing entry is recognized as the symbolic name of corresponding dummy index.

Symbolic names of dummy indices are valid only within the scope of the indexing expression, where the dummy indices were introduced. Beyond the scope the dummy indices are completely inaccessible, so the same symbolic names may be used for other purposes, in particular, to represent dummy indices in other indexing expressions.

The scope of indexing expression, where implicit declarations of dummy indices are valid, depends on the context, in which the indexing expression is used:

— If the indexing expression is used in iterated operator, its scope extends until the end of the integrand.

— If the indexing expression is used as a primary set expression, its scope extends until the end of that indexing expression.

— If the indexing expression is used to define the subscript domain in declarations of some model objects, its scope extends until the end of the corresponding statement.

The indexing mechanism implemented by means of indexing expressions is best explained by some examples discussed below.

Let there be given three sets:

$$\begin{aligned} &A = \{4, 7, 9\}, \\ &B = \{(1, Jan), (1, Feb), (2, Mar), (2, Apr), (3, May), (3, Jun)\}, \\ &C = \{a, b, c\}, \end{aligned}$$

where A and C consist of 1-tuples (singlets), B consists of 2-tuples (doublets). Consider the following indexing expression:

```
{i in A, (j,k) in B, l in C}
```

where `i`, `j`, `k`, and `l` are dummy indices.

Although MathProg is not a procedural language, for any indexing expression an equivalent algorithmic description can be given. In particular, the algorithmic description of the indexing expression above could look like follows:

```
for all $i \in A$ do
    for all $(j, k) \in B$ do
        for all $l \in C$ do
            action;
```

where the dummy indices i, j, k, l are consecutively assigned corresponding components of n-tuples from the basic sets A, B, C, and *action* is some action that depends on the context, where the indexing expression is used. For example, if the action were printing current values of dummy indices, the printout would look like follows:

$$\begin{array}{llll}
i = 4 & j = 1 & k = Jan & l = a \\
i = 4 & j = 1 & k = Jan & l = b \\
i = 4 & j = 1 & k = Jan & l = c \\
i = 4 & j = 1 & k = Feb & l = a \\
i = 4 & j = 1 & k = Feb & l = b \\
 & \multicolumn{2}{c}{\ldots\ldots\ldots} & \\
i = 9 & j = 3 & k = Jun & l = b \\
i = 9 & j = 3 & k = Jun & l = c
\end{array}$$

Let the example indexing expression be used in the following iterated operation:

```
sum{i in A, (j,k) in B, l in C} p[i,j,k,l]
```

where `p` is a 4-dimensional numeric parameter or some numeric expression whose resultant value depends on `i`, `j`, `k`, and `l`. In this case the action is summation, so the resultant value of the primary numeric expression is:

$$\sum_{i \in A, (j,k) \in B, l \in C} (p_{ijkl}).$$

Now let the example indexing expression be used as a primary set expression. In this case the action is gathering all 4-tuples (quadruplets) of the form (i, j, k, l) in one set, so the resultant value of such operation is simply the Cartesian product of the basic sets:

$$A \times B \times C = \{(i, j, k, l) : i \in A, (j, k) \in B, l \in C\}.$$

Note that in this case the same indexing expression might be written in the reduced form:

```
{A, B, C}
```

because the dummy indices i, j, k, and l are not referenced and therefore their symbolic names need not be specified.

Finally, let the example indexing expression be used as the subscript domain in the declaration of a 4-dimensional model object, say, a numeric parameter:

```
param p{i in A, (j,k) in B, l in C} ...;
```

In this case the action is generating the parameter members, where each member has the form $p[i, j, k, l]$.

As was said above, some indices in the second form of indexing entries may be numeric or symbolic expressions, not only dummy indices. In this case resultant values of such expressions play role of some logical conditions to select only that n-tuples from the Cartesian product of basic sets that satisfy these conditions.

Consider, for example, the following indexing expression:

```
{i in A, (i-1,k) in B, l in C}
```

where `i`, `k`, `l` are dummy indices, and `i-1` is a numeric expression. The algorithmic decsription of this indexing expression is the following:

```
for all $i \in A$ do
    for all $(j, k) \in B$ and $j = i - 1$ do
        for all $l \in C$ do
            action;
```

Thus, if this indexing expression were used as a primary set expression, the resultant set would be the following:

$$\{(4, May, a), (4, May, b), (4, May, c), (4, Jun, a), (4, Jun, b), (4, Jun, c)\}.$$

Should note that in this case the resultant set consists of 3-tuples, not of 4-tuples, because in the indexing expression there is no dummy index that corresponds to the first component of 2-tuples from the set B.

The general rule is: the number of components of n-tuples defined by an indexing expression is the same as the number of dummy indices in that expression, where the correspondence between dummy indices and components on n-tuples in the resultant set is positional, i.e. the first dummy index corresponds to the first component, the second dummy index corresponds to the second component, etc.

In some cases it is needed to select a subset from the Cartesian product of some sets. This may be attained by using an optional logical predicate, which is specified in the indexing expression.

Consider, for example, the following indexing expression:

```
{i in A, (j,k) in B, l in C: i <= 5 and k <> 'Mar'}
```

where the logical expression following the colon is a predicate. The algorithmic description of this indexing expression is the following:

```
for all $i \in A$ do
    for all $(j, k) \in B$ do
        for all $l \in C$ do
            if $i \leq 5$ and $k \neq$ '$Mar$' then
                action;
```

Thus, if this indexing expression were used as a primary set expression, the resultant set would be the following:

$$\{(4,1,Jan,a),(4,1,Feb,a),(4,2,Apr,a),\ldots,(4,3,Jun,c)\}.$$

If no predicate is specified in the indexing expression, one, which takes on the value *true*, is assumed.

3.4 Set expressions

A *set expression* is a rule for computing an elemental set, i.e. a collection of n-tuples, where components of n-tuples are numeric and symbolic quantities.

The primary set expression may be a literal set, unsubscripted set, subscripted set, "arithmetic" set, indexing expression, iterated set expression, conditional set expression, or another set expression enclosed in parentheses.

Examples

`{(123,'aaa'), (i+1,'bbb'), (j-1,'ccc')}`	(literal set)
`I`	(unsubscripted set)
`S[i-1,j+1]`	(subscripted set)
`1..t-1 by 2`	("arithmetic" set)
`{t in 1..T, (t+1,j) in S: (t,j) in F}`	(indexing expression)
`setof{i in I, j in J}(i+1,j-1)`	(iterated set expression)
`if i < j then S[i,j] else F diff S[i,j]`	(conditional set expression)
`(1..10 union 21..30)`	(parenthesized set expression)

More general set expressions containing two or more primary set expressions may be constructed by using certain set operators.

Examples

```
(A union B) inter (I cross J)
1..10 cross (if i < j then {'a', 'b', 'c'} else {'d', 'e', 'f'})
```

3.4.1 Literal sets

A *literal set* is a primary set expression, which has the following two syntactic forms:

$$\{e_1,\ e_2,\ \ldots,\ e_m\}$$

$$\{(e_{11},\ \ldots,\ e_{1n}),\ (e_{21},\ \ldots,\ e_{2n}),\ \ldots,\ (e_{m1},\ \ldots,\ e_{mn})\}$$

where $e_1, \ldots, e_m, e_{11}, \ldots, e_{mn}$ are numeric or symbolic expressions.

If the first form is used, the resultant set consists of 1-tuples (singlets) enumerated within the curly braces. It is allowed to specify an empty set as { }, which has no 1-tuples. If the second form is used, the resultant set consists of n-tuples enumerated within the curly braces, where a particular n-tuple consists of corresponding components enumerated within the parentheses. All n-tuples should have the same number of components.

3.4.2 Unsubscripted sets

If the primary set expression is an unsubscripted set (which should be 0-dimensional), the resultant set is an elemental set associated with the corresponding set object.

3.4.3 Subscripted sets

The primary set expression, which refers to a subscripted set, has the following syntactic form:

$$\mathit{name}\texttt{[}i_1\texttt{,}\ i_2\texttt{,}\ \ldots\texttt{,}\ i_n\texttt{]}$$

where *name* is the symbolic name of the set object, i_1, i_2, ..., i_n are subscripts.

Each subscript should be a numeric or symbolic expression. The number of subscripts in the subscript list should be the same as the dimension of the set object with which the subscript list is associated.

Actual values of subscript expressions are used to identify a particular member of the set object that determines the resultant set.

3.4.4 "Arithmetic" sets

The primary set expression, which is an "arithmetic" set, has the following two syntactic forms:

$$t_0\ \texttt{..}\ t_1\ \texttt{by}\ \delta t$$

$$t_0\ \texttt{..}\ t_1$$

where t_0, t_1, and δt are numeric expressions (the value of δt should not be zero). The second form is equivalent to the first form, where $\delta t = 1$.

If $\delta t > 0$, the resultant set is determined as follows:

$$\{t : \exists k \in \mathcal{Z}(t = t_0 + k\delta t,\ t_0 \leq t \leq t_1)\}.$$

Otherwise, if $\delta t < 0$, the resultant set is determined as follows:

$$\{t : \exists k \in \mathcal{Z}(t = t_0 + k\delta t,\ t_1 \leq t \leq t_0)\}.$$

3.4.5 Indexing expressions

If the primary set expression is an indexing expression, the resultant set is determined as described above in Section 3.3, page 18.

3.4.6 Iterated expressions

An *iterated set expression* is a primary set expression, which has the following syntactic form:

`setof` *indexing-expression integrand*

where *indexing-expression* is an indexing expression, which introduces dummy indices and controls iterating, *integrand* is either a single numeric or symbolic expression or a list of numeric and symbolic expressions separated by commae and enclosed in parentheses.

If the integrand is a single numeric or symbolic expression, the resultant set consists of 1-tuples and is determined as follows:

$$\{x : (i_1, \dots, i_n) \in \Delta\},$$

where x is a value of the integrand, i_1, ..., i_n are dummy indices introduced in the indexing expression, Δ is the domain, a set of n-tuples specified by the indexing expression, which defines particular values assigned to the dummy indices on performing the iterated operation.

If the integrand is a list containing m numeric and symbolic expressions, the resultant set consists of m-tuples and is determined as follows:

$$\{(x_1, \dots, x_m) : (i_1, \dots, i_n) \in \Delta\},$$

where x_1, ..., x_m are values of the expressions in the integrand list, i_1, ..., i_n and Δ have the same meaning as above.

3.4.7 Conditional expressions

A *conditional set expression* is a primary set expression that has the following syntactic form:

`if` b `then` X `else` Y

where b is an logical expression, X and Y are set expressions, which should define sets of the same dimension.

The resultant value of the conditional expression depends on the value of the logical expression that follows the keyword `if`. If it takes on the value *true*, the resultant set is the value of the expression that follows the keyword `then`. Otherwise, if the logical expression takes on the value *false*, the resultant set is the value of the expression that follows the keyword `else`.

3.4.8 Parenthesized expressions

Any set expression may be enclosed in parentheses that syntactically makes it a primary set expression.

Parentheses may be used in set expressions, as in algebra, to specify the desired order in which operations are to be performed. Where parentheses are used, the expression within the parentheses is evaluated before the resultant value is used.

The resultant value of the parenthesized expression is the same as the value of the expression enclosed within parentheses.

3.4.9 Set operators

In MathProg there exist the following set operators, which may be used in set expressions:

X `union` Y	union $X \cup Y$
X `diff` Y	difference $X \backslash Y$
X `symdiff` Y	symmetric difference $X \oplus Y = (X \backslash Y) \cup (Y \backslash X)$
X `inter` Y	intersection $X \cap Y$
X `cross` Y	cross (Cartesian) product $X \times Y$

where X and Y are set expressions, which should define sets of identical dimension (except the Cartesian product).

If the expression includes more than one set operator, all operators are performed from left to right according to the hierarchy of operations (see below).

The resultant value of the expression, which contains set operators, is the result of applying the operators to their operands.

The dimension of the resultant set, i.e. the dimension of n-tuples, of which the resultant set consists of, is the same as the dimension of the operands, except the Cartesian product, where the dimension of the resultant set is the sum of the dimensions of its operands.

3.4.10 Hierarchy of operations

The following list shows the hierarchy of operations in set expressions:

Operation	Hierarchy
Evaluation of numeric operations	1st-7th
Evaluation of symbolic operations	8th-9th
Evaluation of iterated or “arithmetic” set (`setof`, `..`)	10th
Cartesian product (`cross`)	11th
Intersection (`inter`)	12th
Union and difference (`union`, `diff`, `symdiff`)	13th
Conditional evaluation (`if` ... `then` ... `else`)	14th

This hierarchy has the same meaning as was explained above for numeric expressions (see Subsection 3.1.10, page 16).

3.5 Logical expressions

A *logical expression* is a rule for computing a single logical value, which can be either *true* or *false.*

The primary logical expression may be a numeric expression, relational expression, iterated logical expression, or another logical expression enclosed in parentheses.

Examples

`i+1`	(numeric expression)
`a[i,j] < 1.5`	(relational expression)
`s[i+1,j-1] <> 'Mar' & year`	(relational expression)
`(i+1,'Jan') not in I cross J`	(relational expression)
`S union T within A[i] inter B[j]`	(relational expression)
`forall{i in I, j in J} a[i,j] < .5 * b[i]`	(iterated logical expression)
`(a[i,j] < 1.5 or b[i] >= a[i,j])`	(parenthesized logical expression)

More general logical expressions containing two or more primary logical expressions may be constructed by using certain logical operators.

Examples

```
not (a[i,j] < 1.5 or b[i] >= a[i,j]) and (i,j) in S
(i,j) in S or (i,j) not in T diff U
```

3.5.1 Numeric expressions

The resultant value of the primary logical expression, which is a numeric expression, is *true*, if the resultant value of the numeric expression is non-zero. Otherwise the resultant value of the logical expression is *false.*

3.5.2 Relational operators

In MathProg there exist the following relational operators, which may be used in logical expressions:

x `<` y	test on $x < y$
x `<=` y	test on $x \leq y$
x `=` y, x `==` y	test on $x = y$
x `>=` y	test on $x \geq y$
x `>` y	test on $x > y$
x `<>` y, x `!=` y	test on $x \neq y$
x `in` Y	test on $x \in Y$
`(`x_1`,...,`x_n`) in` Y	test on $(x_1, \ldots, x_n) \in Y$
x `not in` Y, x `!in` Y	test on $x \notin Y$
`(`x_1`,...,`x_n`) not in` Y, `(`x_1`,...,`x_n`) !in` Y	test on $(x_1, \ldots, x_n) \notin Y$
X `within` Y	test on $X \subseteq Y$
X `not within` Y, X `!within` Y	test on $X \not\subseteq Y$

where x, x_1, ..., x_n, y are numeric or symbolic expressions, X and Y are set expression.

1. In the operations `in`, `not in`, and `!in` the number of components in the first operands should be the same as the dimension of the second operand.

2. In the operations `within`, `not within`, and `!within` both operands should have identical dimension.

All the relational operators listed above have their conventional mathematical meaning. The resultant value is *true*, if corresponding relation is satisfied for its operands, otherwise *false*. (Note that symbolic values are ordered lexicographically, and any numeric value precedes any symbolic value.)

3.5.3 Iterated expressions

An *iterated logical expression* is a primary logical expression, which has the following syntactic form:

iterated-operator indexing-expression integrand

where *iterated-operator* is the symbolic name of the iterated operator to be performed (see below), *indexing-expression* is an indexing expression which introduces dummy indices and controls iterating, *integrand* is a numeric expression that participates in the operation.

In MathProg there exist two iterated operators, which may be used in logical expressions:

`forall`	∀-quantification	$\forall(i_1,\dots,i_n)\in\Delta[f(i_1,\dots,i_n)]$,
`exists`	∃-quantification	$\exists(i_1,\dots,i_n)\in\Delta[f(i_1,\dots,i_n)]$,

where i_1, ..., i_n are dummy indices introduced in the indexing expression, Δ is the domain, a set of n-tuples specified by the indexing expression which defines particular values assigned to the dummy indices on performing the iterated operation, $f(i_1,\dots,i_n)$ is the integrand, a logical expression whose resultant value depends on the dummy indices.

For ∀-quantification the resultant value of the iterated logical expression is *true*, if the value of the integrand is *true* for all n-tuples contained in the domain, otherwise *false*.

For ∃-quantification the resultant value of the iterated logical expression is *false*, if the value of the integrand is *false* for all n-tuples contained in the domain, otherwise *true*.

3.5.4 Parenthesized expressions

Any logical expression may be enclosed in parentheses that syntactically makes it a primary logical expression.

Parentheses may be used in logical expressions, as in algebra, to specify the desired order in which operations are to be performed. Where parentheses are used, the expression within the parentheses is evaluated before the resultant value is used.

The resultant value of the parenthesized expression is the same as the value of the expression enclosed within parentheses.

3.5.5 Logical operators

In MathProg there exist the following logical operators, which may be used in logical expressions:

`not` x, `!`x	negation $\neg\ x$
x `and` y, x `&&` y	conjunction (logical "and") $x\ \&\ y$
x `or` y, x `||` y	disjunction (logical "or") $x \vee y$

where x and y are logical expressions.

If the expression includes more than one logical operator, all operators are performed from left to right according to the hierarchy of the operations (see below). The resultant value of the expression, which contains logical operators, is the result of applying the operators to their operands.

3.5.6 Hierarchy of operations

The following list shows the hierarchy of operations in logical expressions:

Operation	Hierarchy
Evaluation of numeric operations	1st-7th
Evaluation of symbolic operations	8th-9th
Evaluation of set operations	10th-14th
Relational operations (`<`, `<=`, etc.)	15th
Negation (`not`, `!`)	16th
Conjunction (`and`, `&&`)	17th
$\forall$- and $\exists$-quantification (`forall`, `exists`)	18th
Disjunction (`or`, `||`)	19th

This hierarchy has the same meaning as was explained above for numeric expressions (see Subsection 3.1.10, page 16).

3.6 Linear expressions

A *linear expression* is a rule for computing so called a *linear form* or simply a *formula*, which is a linear (or affine) function of elemental variables.

The primary linear expression may be an unsubscripted variable, subscripted variable, iterated linear expression, conditional linear expression, or another linear expression enclosed in parentheses.

It is also allowed to use a numeric expression as the primary linear expression, in which case the resultant value of the numeric expression is automatically converted to a formula that includes the constant term only.

Examples

```
z                                      (unsubscripted variable)
x[i,j]                                 (subscripted variable)
sum{j in J} (a[i,j] * x[i,j] + 3 * y[i-1])   (iterated linear expression)
if i in I then x[i,j] else 1.5 * z + 3.25    (conditional linear expression)
(a[i,j] * x[i,j] + y[i-1] + .1)        (parenthesized linear expression)
```

More general linear expressions containing two or more primary linear expressions may be constructed by using certain arithmetic operators.

Examples

```
2 * x[i-1,j+1] + 3.5 * y[k] + .5 * z
(- x[i,j] + 3.5 * y[k]) / sum{t in T} abs(d[i,j,t])
```

3.6.1 Unsubscripted variables

If the primary linear expression is an unsubscripted variable (which should be 0-dimensional), the resultant formula is that unsubscripted variable.

3.6.2 Subscripted variables

The primary linear expression, which refers to a subscripted variable, has the following syntactic form:

$$\mathit{name}\texttt{[}i_1\texttt{,}\ i_2\texttt{,}\ \ldots\texttt{,}\ i_n\texttt{]}$$

where *name* is the symbolic name of the model variable, i_1, i_2, ..., i_n are subscripts.

Each subscript should be a numeric or symbolic expression. The number of subscripts in the subscript list should be the same as the dimension of the model variable with which the subscript list is associated.

Actual values of the subscript expressions are used to identify a particular member of the model variable that determines the resultant formula, which is an elemental variable associated with corresponding member.

3.6.3 Iterated expressions

An *iterated linear expression* is a primary linear expression, which has the following syntactic form:

$$\texttt{sum}\ \mathit{indexing\text{-}expression}\ \mathit{integrand}$$

where *indexing-expression* is an indexing expression, which introduces dummy indices and controls iterating, *integrand* is a linear expression that participates in the operation.

The iterated linear expression is evaluated exactly in the same way as the iterated numeric expression (see Subection 3.1.6, page 15) with exception that the integrand participated in the summation is a formula, not a numeric value.

3.6.4 Conditional expressions

A *conditional linear expression* is a primary linear expression, which has one of the following two syntactic forms:

$$\texttt{if}\ b\ \texttt{then}\ f\ \texttt{else}\ g$$

$$\texttt{if}\ b\ \texttt{then}\ f$$

where b is an logical expression, f and g are linear expressions.

The conditional linear expression is evaluated exactly in the same way as the conditional numeric expression (see Subsection 3.1.7, page 15) with exception that operands participated in the operation are formulae, not numeric values.

3.6.5 Parenthesized expressions

Any linear expression may be enclosed in parentheses that syntactically makes it a primary linear expression.

Parentheses may be used in linear expressions, as in algebra, to specify the desired order in which operations are to be performed. Where parentheses are used, the expression within the parentheses is evaluated before the resultant formula is used.

The resultant value of the parenthesized expression is the same as the value of the expression enclosed within parentheses.

3.6.6 Arithmetic operators

In MathProg there exists the following arithmetic operators, which may be used in linear expressions:

+ f	unary plus
- f	unary minus
f + g	addition
f - g	subtraction
x * f, f * x	multiplication
f / x	division

where f and g are linear expressions, x is a numeric expression (more precisely, a linear expression containing only the constant term).

If the expression includes more than one arithmetic operator, all operators are performed from left to right according to the hierarchy of operations (see below). The resultant value of the expression, which contains arithmetic operators, is the result of applying the operators to their operands.

3.6.7 Hierarchy of operations

The hierarchy of arithmetic operations used in linear expressions is the same as for numeric expressions (see Subsection 3.1.10, page 16).

Chapter 4

Statements

Statements are basic units of the model description. In MathProg all statements are divided into two categories: declaration statements and functional statements.

Declaration statements (set statement, parameter statement, variable statement, constraint statement, objective statement) are used to declare model objects of certain kinds and define certain properties of such objects.

Functional statements (solve statement, check statement, display statement, printf statement, loop statement, table statement) are intended for performing some specific actions.

Note that declaration statements may follow in arbitrary order, which does not affect the result of translation. However, any model object should be declared before it is referenced in other statements.

4.1 Set statement

```
set name alias domain , attrib , . . . , attrib ;
```

name is a symbolic name of the set;

alias is an optional string literal, which specifies an alias of the set;

domain is an optional indexing expression, which specifies a subscript domain of the set;

attrib, . . . , *attrib* are optional attributes of the set. (Commae preceding attributes may be omitted.)

Optional attributes

`dimen` *n*
specifies the dimension of n-tuples which the set consists of;

`within` *expression*
specifies a superset which restricts the set or all its members (elemental sets) to be within that superset;

`:=` *expression*
: specifies an elemental set assigned to the set or its members;

`default` *expression*
: specifies an elemental set assigned to the set or its members whenever no appropriate data are available in the data section.

Examples

```
set nodes;
set arcs within nodes cross nodes;
set step{s in 1..maxiter} dimen 2 := if s = 1 then arcs else step[s-1]
   union setof{k in nodes, (i,k) in step[s-1], (k,j) in step[s-1]}(i,j);
set A{i in I, j in J}, within B[i+1] cross C[j-1], within D diff E,
   default {('abc',123), (321,'cba')};
```

The set statement declares a set. If the subscript domain is not specified, the set is a simple set, otherwise it is an array of elemental sets.

The `dimen` attribute specifies the dimension of n-tuples, which the set (if it is a simple set) or its members (if the set is an array of elemental sets) consist of, where n should be an unsigned integer from 1 to 20. At most one `dimen` attribute can be specified. If the `dimen` attribute is not specified, the dimension of n-tuples is implicitly determined by other attributes (for example, if there is a set expression that follows `:=` or the keyword `default`, the dimension of n-tuples of corresponding elemental set is used). If no dimension information is available, `dimen 1` is assumed.

The `within` attribute specifies a set expression whose resultant value is a superset used to restrict the set (if it is a simple set) or its members (if the set is an array of elemental sets) to be within that superset. Arbitrary number of `within` attributes may be specified in the same set statement.

The assign (`:=`) attribute specifies a set expression used to evaluate elemental set(s) assigned to the set (if it is a simple set) or its members (if the set is an array of elemental sets). If the assign attribute is specified, the set is *computable* and therefore needs no data to be provided in the data section. If the assign attribute is not specified, the set should be provided with data in the data section. At most one assign or default attribute can be specified for the same set.

The `default` attribute specifies a set expression used to evaluate elemental set(s) assigned to the set (if it is a simple set) or its members (if the set is an array of elemental sets) whenever no appropriate data are available in the data section. If neither assign nor default attribute is specified, missing data will cause an error.

4.2 Parameter statement

```
param name alias domain , attrib , . . . , attrib ;
```

name is a symbolic name of the parameter;

alias is an optional string literal, which specifies an alias of the parameter;

domain is an optional indexing expression, which specifies a subscript domain of the parameter;

attrib, . . . , *attrib* are optional attributes of the parameter. (Commae preceding attributes may be omitted.)

Optional attributes

`integer`
specifies that the parameter is integer;

`binary`
specifies that the parameter is binary;

`symbolic`
specifies that the parameter is symbolic;

relation expression
(where *relation* is one of: `<`, `<=`, `=`, `==`, `>=`, `>`, `<>`, `!=`)
specifies a condition that restricts the parameter or its members to satisfy that condition;

`in` *expression*
specifies a superset that restricts the parameter or its members to be in that superset;

`:=` *expression*
specifies a value assigned to the parameter or its members;

`default` *expression*
specifies a value assigned to the parameter or its members whenever no appropriate data are available in the data section.

Examples

```
param units{raw, prd} >= 0;
param profit{prd, 1..T+1};
param N := 20 integer >= 0 <= 100;
param comb 'n choose k' {n in 0..N, k in 0..n} :=
   if k = 0 or k = n then 1 else comb[n-1,k-1] + comb[n-1,k];
param p{i in I, j in J}, integer, >= 0, <= i+j, in A[i] symdiff B[j],
   in C[i,j], default 0.5 * (i + j);
param month symbolic default 'May' in {'Mar', 'Apr', 'May'};
```

The parameter statement declares a parameter. If a subscript domain is not specified, the parameter is a simple (scalar) parameter, otherwise it is a n-dimensional array.

The type attributes `integer`, `binary`, and `symbolic` qualify the type of values that can be assigned to the parameter as shown below:

Type attribute	Assigned values
(not specified)	Any numeric values
`integer`	Only integer numeric values
`binary`	Either 0 or 1
`symbolic`	Any numeric and symbolic values

The `symbolic` attribute cannot be specified along with other type attributes. Being specified it should precede all other attributes.

The condition attribute specifies an optional condition that restricts values assigned to the parameter to satisfy that condition. This attribute has the following syntactic forms:

`<` v	check for $x < v$
`<=` v	check for $x \leq v$
`=` v, `==` v	check for $x = v$
`>=` v	check for $x \geq v$
`>` v	check for $x \geq v$
`<>` v, `!=` v	check for $x \neq v$

where x is a value assigned to the parameter, v is the resultant value of a numeric or symbolic expression specified in the condition attribute. Arbitrary number of condition attributes can be specified for the same parameter. If a value being assigned to the parameter during model evaluation violates at least one of specified conditions, an error is raised. (Note that symbolic values are ordered lexicographically, and any numeric value precedes any symbolic value.)

The `in` attribute is similar to the condition attribute and specifies a set expression whose resultant value is a superset used to restrict numeric or symbolic values assigned to the parameter to be in that superset. Arbitrary number of the `in` attributes can be specified for the same parameter. If a value being assigned to the parameter during model evaluation is not in at least one of specified supersets, an error is raised.

The assign (`:=`) attribute specifies a numeric or symbolic expression used to compute a value assigned to the parameter (if it is a simple parameter) or its member (if the parameter is an array). If the assign attribute is specified, the parameter is *computable* and therefore needs no data to be provided in the data section. If the assign attribute is not specified, the parameter should be provided with data in the data section. At most one assign or `default` attribute can be specified for the same parameter.

The `default` attribute specifies a numeric or symbolic expression used to compute a value assigned to the parameter or its member whenever no appropriate data are available in the data section. If neither assign nor `default` attribute is specified, missing data will cause an error.

4.3 Variable statement

```
var name alias domain , attrib , ... , attrib ;
```

name is a symbolic name of the variable;

alias is an optional string literal, which specifies an alias of the variable;

domain is an optional indexing expression, which specifies a subscript domain of the variable;

attrib, ..., *attrib* are optional attributes of the variable. (Commae preceding attributes may be omitted.)

Optional attributes

`integer`
restricts the variable to be integer;

`binary`
restricts the variable to be binary;

`>=` *expression*
specifies an lower bound of the variable;

`<=` *expression*
specifies an upper bound of the variable;

`=` *expression*
specifies a fixed value of the variable;

Examples

```
var x >= 0;
var y{I,J};
var make{p in prd}, integer, >= commit[p], <= market[p];
var store{raw, 1..T+1} >= 0;
var z{i in I, j in J} >= i+j;
```

The variable statement declares a variable. If a subscript domain is not specified, the variable is a simple (scalar) variable, otherwise it is a n-dimensional array of elemental variables.

Elemental variable(s) associated with the model variable (if it is a simple variable) or its members (if it is an array) correspond to the variables in the LP/MIP problem formulation (see Section 1.1, page 6). Note that only elemental variables actually referenced in some constraints and/or objectives are included in the LP/MIP problem instance to be generated.

The type attributes `integer` and `binary` restrict the variable to be integer or binary, respectively. If no type attribute is specified, the variable is continuous. If all variables in the model are continuous, the corresponding problem is of LP class. If there is at least one integer or binary variable, the problem is of MIP class.

The lower bound (`>=`) attribute specifies a numeric expression for computing an lower bound of the variable. At most one lower bound can be specified. By default all variables (except binary ones) have no lower bound, so if a variable is required to be non-negative, its zero lower bound should be explicitly specified.

The upper bound (`<=`) attribute specifies a numeric expression for computing an upper bound of the variable. At most one upper bound attribute can be specified.

The fixed value (`=`) attribute specifies a numeric expression for computing a value, at which the variable is fixed. This attribute cannot be specified along with the bound attributes.

4.4 Constraint statement

`s.t.` *name alias domain* `:` *expression* `,` `=` *expression* `;`

`s.t.` *name alias domain* `:` *expression* `,` `<=` *expression* `;`

`s.t.` *name alias domain* `:` *expression* `,` `>=` *expression* `;`

`s.t.` *name alias domain* `:` *expression* `,` `<=` *expression* `,` `<=` *expression* `;`

`s.t.` *name alias domain* `:` *expression* `,` `>=` *expression* `,` `>=` *expression* `;`

name is a symbolic name of the constraint;

alias is an optional string literal, which specifies an alias of the constraint;

domain is an optional indexing expression, which specifies a subscript domain of the constraint;

expression is a linear expression used to compute a component of the constraint. (Commae following expressions may be omitted.)

(The keyword `s.t.` may be written as `subject to` or as `subj to`, or may be omitted at all.)

Examples

```
s.t. r: x + y + z, >= 0, <= 1;
limit{t in 1..T}: sum{j in prd} make[j,t] <= max_prd;
subject to balance{i in raw, t in 1..T}:
   store[i,t+1] = store[i,t] - sum{j in prd} units[i,j] * make[j,t];
subject to rlim 'regular-time limit' {t in time}:
   sum{p in prd} pt[p] * rprd[p,t] <= 1.3 * dpp[t] * crews[t];
```

The constraint statement declares a constraint. If a subscript domain is not specified, the constraint is a simple (scalar) constraint, otherwise it is a n-dimensional array of elemental constraints.

Elemental constraint(s) associated with the model constraint (if it is a simple constraint) or its members (if it is an array) correspond to the linear constraints in the LP/MIP problem formulation (see Section 1.1, page 6).

If the constraint has the form of equality or single inequality, i.e. includes two expressions, one of which follows the colon and other follows the relation sign `=`, `<=`, or `>=`, both expressions in the statement can be linear expressions. If the constraint has the form of double inequality, i.e. includes three expressions, the middle expression can be a linear expression while the leftmost and rightmost ones can be only numeric expressions.

Generating the model is, roughly speaking, generating its constraints, which are always evaluated for the entire subscript domain. Evaluation of the constraints leads, in turn, to evaluation of other model objects such as sets, parameters, and variables.

Constructing an actual linear constraint included in the problem instance, which (constraint) corresponds to a particular elemental constraint, is performed as follows.

If the constraint has the form of equality or single inequality, evaluation of both linear expressions gives two resultant linear forms:

$$\begin{aligned} f &= a_1x_1 + a_2x_2 + \ldots + a_nx_n + a_0, \\ g &= b_1x_1 + a_2x_2 + \ldots + a_nx_n + b_0, \end{aligned}$$

where $x_1, x_2, \ldots, x_n$ are elemental variables; $a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n$ are numeric coefficients; a_0 and b_0 are constant terms. Then all linear terms of f and g are carried to the left-hand side, and the constant terms are carried to the right-hand side, that gives the final elemental constraint in the standard form:

$$(a_1 - b_1)x_1 + (a_2 - b_2)x_2 + \ldots + (a_n - b_n)x_n \left\{ \begin{array}{c} = \\ \leq \\ \geq \end{array} \right\} b_0 - a_0.$$

If the constraint has the form of double inequality, evaluation of the middle linear expression gives the resultant linear form:

$$f = a_1x_1 + a_2x_2 + \ldots + a_nx_n + a_0,$$

and evaluation of the leftmost and rightmost numeric expressions gives two numeric values l and u, respectively. Then the constant term of the linear form is carried to both left-hand and righthandsides that gives the final elemental constraint in the standard form:

$$l - a_0 \leq a_1x_1 + a_2x_2 + \ldots + a_nx_n \leq u - a_0.$$

4.5 Objective statement

```
minimize name alias domain : expression ;

maximize name alias domain : expression ;
```

name is a symbolic name of the objective;

alias is an optional string literal, which specifies an alias of the objective;

domain is an optional indexing expression, which specifies a subscript domain of the objective;

expression is a linear expression used to compute the linear form of the objective.

Examples

```
minimize obj: x + 1.5 * (y + z);
maximize total_profit: sum{p in prd} profit[p] * make[p];
```

The objective statement declares an objective. If a subscript domain is not specified, the objective is a simple (scalar) objective. Otherwise it is a n-dimensional array of elemental objectives.

Elemental objective(s) associated with the model objective (if it is a simple objective) or its members (if it is an array) correspond to general linear constraints in the LP/MIP problem formulation (see Section 1.1, page 6). However, unlike constraints the corresponding linear forms are free (unbounded).

Constructing an actual linear constraint included in the problem instance, which (constraint) corresponds to a particular elemental constraint, is performed as follows. The linear expression specified in the objective statement is evaluated that, gives the resultant linear form:

$$f = a_1x_1 + a_2x_2 + \ldots + a_nx_n + a_0,$$

where x_1, x_2, $\ldots$, x_n are elemental variables; a_1, a_2, $\ldots$, a_n are numeric coefficients; a_0 is the constant term. Then the linear form is used to construct the final elemental constraint in the standard form:

$$-\infty < a_1x_1 + a_2x_2 + \ldots + a_nx_n + a_0 < +\infty.$$

As a rule the model description contains only one objective statement that defines the objective function used in the problem instance. However, it is allowed to declare arbitrary number of objectives, in which case the actual objective function is the first objective encountered in the model description. Other objectives are also included in the problem instance, but they do not affect the objective function.

4.6 Solve statement

```
solve ;
```

The solve statement is optional and can be used only once. If no solve statement is used, one is assumed at the end of the model section.

The solve statement causes the model to be solved, that means computing numeric values of all model variables. This allows using variables in statements below the solve statement in the same way as if they were numeric parameters.

Note that the variable, constraint, and objective statements cannot be used below the solve statement, i.e. all principal components of the model should be declared above the solve statement.

4.7 Check statement

```
check domain : expression ;
```

domain is an optional indexing expression, which specifies a subscript domain of the check statement;

expression is an logical expression which specifies the logical condition to be checked. (The colon preceding *expression* may be omitted.)

Examples

```
check: x + y <= 1 and x >= 0 and y >= 0;
check sum{i in ORIG} supply[i] = sum{j in DEST} demand[j];
check{i in I, j in 1..10}: S[i,j] in U[i] union V[j];
```

The check statement allows checking the resultant value of an logical expression specified in the statement. If the value is *false*, an error is reported.

If the subscript domain is not specified, the check is performed only once. Specifying the subscript domain allows performing multiple check for every n-tuple in the domain set. In the latter case the logical expression may include dummy indices introduced in corresponding indexing expression.

4.8 Display statement

```
display domain : item , ... , item ;
```

domain is an optional indexing expression, which specifies a subscript domain of the display statement;

item, ..., *item* are items to be displayed. (The colon preceding the first item may be omitted.)

Examples

```
display: 'x =', x, 'y =', y, 'z =', z;
display sqrt(x ** 2 + y ** 2 + z ** 2);
display{i in I, j in J}: i, j, a[i,j], b[i,j];
```

The display statement evaluates all items specified in the statement and writes their values on the standard output (terminal) in plain text format.

If a subscript domain is not specified, items are evaluated and then displayed only once. Specifying the subscript domain causes items to be evaluated and displayed for every n-tuple in the domain set. In the latter case items may include dummy indices introduced in corresponding indexing expression.

An item to be displayed can be a model object (set, parameter, variable, constraint, objective) or an expression.

If the item is a computable object (i.e. a set or parameter provided with the assign attribute), the object is evaluated over the entire domain and then its content (i.e. the content of the object array) is displayed. Otherwise, if the item is not a computable object, only its current content (i.e. members actually generated during the model evaluation) is displayed.

If the item is an expression, the expression is evaluated and its resultant value is displayed.

4.9 Printf statement

`printf` *domain* : *format* , *expression* , . . . , *expression* ;

`printf` *domain* : *format* , *expression* , . . . , *expression* `>` *filename* ;

`printf` *domain* : *format* , *expression* , . . . , *expression* `>>` *filename* ;

domain is an optional indexing expression, which specifies a subscript domain of the printf statement;

format is a symbolic expression whose value specifies a format control string. (The colon preceding the format expression may be omitted.)

expression, . . . , *expression* are zero or more expressions whose values have to be formatted and printed. Each expression should be of numeric, symbolic, or logical type.

filename is a symbolic expression whose value specifies a name of a text file, to which the output is redirected. The flag `>` means creating a new empty file while the flag `>>` means appending the output to an existing file. If no file name is specified, the output is written on the standard output (terminal).

Examples

```
printf 'Hello, world!\n';
printf: "x = %.3f; y = %.3f; z = %.3f\n", x, y, z > "result.txt";
printf{i in I, j in J}: "flow from %s to %s is %d\n", i, j, x[i,j]
   >> result_file & ".txt";
printf{i in I} 'total flow from %s is %g\n', i, sum{j in J} x[i,j];
printf{k in K} "x[%s] = " & (if x[k] < 0 then "?" else "%g"),
   k, x[k];
```

The printf statement is similar to the display statement, however, it allows formatting data to be written.

If a subscript domain is not specified, the printf statement is executed only once. Specifying a subscript domain causes executing the printf statement for every n-tuple in the domain set. In the latter case the format and expression may include dummy indices introduced in corresponding indexing expression.

The format control string is a value of the symbolic expression *format* specified in the printf statement. It is composed of zero or more directives as follows: ordinary characters (not `%`), which are copied unchanged to the output stream, and conversion specifications, each of which causes evaluating corresponding expression specified in the printf statement, formatting it, and writing its resultant value to the output stream.

Conversion specifications that may be used in the format control string are the following: `d`, `i`, `f`, `F`, `e`, `E`, `g`, `G`, and `s`. These specifications have the same syntax and semantics as in the C programming language.

4.10 For statement

> `for` *domain* `:` *statement* `;`
>
> `for` *domain* `:` `{` *statement* ... *statement* `}` `;`

domain is an indexing expression which specifies a subscript domain of the for statement. (The colon following the indexing expression may be omitted.)

statement is a statement, which should be executed under control of the for statement;

statement, ..., *statement* is a sequence of statements (enclosed in curly braces), which should be executed under control of the for statement.

Only the following statements can be used within the for statement: check, display, printf, and another for.

Examples

```
for {(i,j) in E: i != j}
{  printf "flow from %s to %s is %g\n", i, j, x[i,j];
   check x[i,j] >= 0;
}
for {i in 1..n}
{  for {j in 1..n} printf " %s", if x[i,j] then "Q" else ".";
   printf("\n");
}
for {1..72} printf("*");
```

The for statement causes a statement or a sequence of statements specified as part of the for statement to be executed for every n-tuple in the domain set. Thus, statements within the for statement may include dummy indices introduced in corresponding indexing expression.

4.11 Table statement

> `table` *name alias* `IN` *driver arg* . . . *arg* `:`
>   *set* `<-` `[` *fld* `,` . . . `,` *fld* `]` `,` *par* `~` *fld* `,` . . . `,` *par* `~` *fld* `;`
>
> `table` *name alias domain* `OUT` *driver arg* . . . *arg* `:`
>   *expr* `~` *fld* `,` . . . `,` *expr* `~` *fld* `;`

name is a symbolic name of the table;

alias is an optional string literal, which specifies an alias of the table;

domain is an indexing expression, which specifies a subscript domain of the (output) table;

`IN` means reading data from the input table;

`OUT` means writing data to the output table;

driver is a symbolic expression, which specifies the driver used to access the table (for details see Appendix C, page 61);

arg is an optional symbolic expression, which is an argument passed to the table driver. This symbolic expression should not include dummy indices specified in the domain;

set is the name of an optional simple set called *control set*. It can be omitted along with the delimiter `<-`;

fld is a field name. Within square brackets at least one field should be specified. The field name following a parameter name or expression is optional and can be omitted along with the delimiter `~`, in which case the name of corresponding model object is used as the field name;

par is a symbolic name of a model parameter;

expr is a numeric or symbolic expression.

Examples

```
table data IN "CSV" "data.csv": S <- [FROM,TO], d~DISTANCE,
   c~COST;
table result{(f,t) in S} OUT "CSV" "result.csv": f~FROM, t~TO,
   x[f,t]~FLOW;
```

The table statement allows reading data from a table into model objects such as sets and (non-scalar) parameters as well as writing data from the model to a table.

4.11.1 Table structure

A *data table* is an (unordered) set of *records*, where each record consists of the same number of *fields*, and each field is provided with a unique symbolic name called the *field name*. For example:

		First field ↓	Second field ↓	. . .	Last field ↓
Table header	→	`FROM`	`TO`	`DISTANCE`	`COST`
First record	→	`Seattle`	`New-York`	`2.5`	`0.12`
Second record	→	`Seattle`	`Chicago`	`1.7`	`0.08`
		`Seattle`	`Topeka`	`1.8`	`0.09`
. . .		`San-Diego`	`New-York`	`2.5`	`0.15`
		`San-Diego`	`Chicago`	`1.8`	`0.10`
Last record	→	`San-Diego`	`Topeka`	`1.4`	`0.07`

4.11.2 Reading data from input table

The input table statement causes reading data from the specified table record by record.

Once a next record has been read, numeric or symbolic values of fields, whose names are enclosed in square brackets in the table statement, are gathered into n-tuple, and if the control set is specified in the table statement, this n-tuple is added to it. Besides, a numeric or symbolic value of each field associated with a model parameter is assigned to the parameter member identified by subscripts, which are components of the n-tuple just read.

For example, the following input table statement:

```
table data IN "...": S <- [FROM,TO], d~DISTANCE, c~COST;
```

causes reading values of four fields named `FROM`, `TO`, `DISTANCE`, and `COST` from each record of the specified table. Values of fields `FROM` and `TO` give a pair (f,t), which is added to the control set `S`. The value of field `DISTANCE` is assigned to parameter member `d`$[f,t]$, and the value of field `COST` is assigned to parameter member `c`$[f,t]$.

Note that the input table may contain extra fields whose names are not specified in the table statement, in which case values of these fields on reading the table are ignored.

4.11.3 Writing data to output table

The output table statement causes writing data to the specified table. Note that some drivers (namely, CSV and xBASE) destroy the output table before writing data, i.e. delete all its existing records.

Each n-tuple in the specified domain set generates one record written to the output table. Values of fields are numeric or symbolic values of corresponding expressions specified in the table statement. These expressions are evaluated for each n-tuple in the domain set and, thus, may include dummy indices introduced in the corresponding indexing expression.

For example, the following output table statement:

```
table result{(f,t) in S} OUT "...": f~FROM, t~TO, x[f,t]~FLOW;
```

causes writing records, by one record for each pair (f, t) in set `S`, to the output table, where each record consists of three fields named `FROM`, `TO`, and `FLOW`. The values written to fields `FROM` and `TO` are current values of dummy indices `f` and `t`, and the value written to field `FLOW` is a value of member `x`$[f, t]$ of corresponding subscripted parameter or variable.

Chapter 5

Model data

Model data include elemental sets, which are "values" of model sets, and numeric and symbolic values of model parameters.

In MathProg there are two different ways to saturate model sets and parameters with data. One way is simply providing necessary data using the assign attribute. However, in many cases it is more practical to separate the model itself and particular data needed for the model. For the latter reason in MathProg there is another way, when the model description is divided into two parts: model section and data section.

A *model section* is a main part of the model description that contains declarations of all model objects and is common for all problems based on that model.

A *data section* is an optional part of the model description that contains model data specific for a particular problem.

In MathProg model and data sections can be placed either in one text file or in two separate text files.

1. If both model and data sections are placed in one file, the file is composed as follows:

```
statement;
statement;
   . . .
statement;
data;
data block;
data block;
   . . .
data block;
end;
```

2. If the model and data sections are placed in two separate files, the files are composed as follows:

```
statement;
statement;
   . . .
statement;
end;
```

Model file

```
data;
data block;
data block;
   . . .
data block;
end;
```

Data file

Note: If the data section is placed in a separate file, the keyword `data` is optional and may be omitted along with the semicolon that follows it.

5.1 Coding data section

The *data section* is a sequence of data blocks in various formats, which are discussed in following sections. The order, in which data blocks follow in the data section, may be arbitrary, not necessarily the same, in which corresponding model objects follow in the model section.

The rules of coding the data section are commonly the same as the rules of coding the model description (see Section 2, page 9), i.e. data blocks are composed from basic lexical units such as symbolic names, numeric and string literals, keywords, delimiters, and comments. However, for the sake of convenience and for improving readability there is one deviation from the common rule: if a string literal consists of only alphanumeric characters (including the underscore character), the signs `+` and `-`, and/or the decimal point, it may be coded without bordering by (single or double) quotes.

All numeric and symbolic material provided in the data section is coded in the form of numbers and symbols, i.e. unlike the model section no expressions are allowed in the data section. Nevertheless, the signs `+` and `-` can precede numeric literals to allow coding signed numeric quantities, in which case there should be no white-space characters between the sign and following numeric literal (if there is at least one white-space, the sign and following numeric literal are recognized as two different lexical units).

5.2 Set data block

> `set` *name* `,` *record* `, . . . ,` *record* `;`
>
> `set` *name* `[` *symbol* `, . . . ,` *symbol* `]` `,` *record* `, . . . ,` *record* `;`

name is a symbolic name of the set;

symbol, . . . , *symbol* are subscripts, which specify a particular member of the set (if the set is an array, i.e. a set of sets);

record, . . . , *record* are data records.

Commae preceding data records may be omitted.

Data records

`:=`
: is a non-significant data record, which may be used freely to improve readability;

`(` *slice* `)`
: specifies a slice;

simple-data
: specifies set data in the simple format;

`:` *matrix-data*
: specifies set data in the matrix format;

`(tr) :` *matrix-data*
: specifies set data in the transposed matrix format. (In this case the colon following the keyword `(tr)` may be omitted.)

Examples

```
set month := Jan Feb Mar Apr May Jun;
set month "Jan", "Feb", "Mar", "Apr", "May", "Jun";
set A[3,Mar] := (1,2) (2,3) (4,2) (3,1) (2,2) (4,4) (3,4);
set A[3,'Mar'] := 1 2 2 3 4 2 3 1 2 2 4 4 3 4;
set A[3,'Mar'] : 1 2 3 4 :=
               1 - + - -
               2 - + + -
               3 + - - +
               4 - + - + ;
set B := (1,2,3) (1,3,2) (2,3,1) (2,1,3) (1,2,2) (1,1,1) (2,1,1);
set B := (*,*,*) 1 2 3, 1 3 2, 2 3 1, 2 1 3, 1 2 2, 1 1 1, 2 1 1;
set B := (1,*,2) 3 2 (2,*,1) 3 1 (1,2,3) (2,1,3) (1,1,1);
set B := (1,*,*) : 1 2 3 :=
                 1 + - -
                 2 - + +
                 3 - + -
```

```
(2,*,*) : 1 2 3 :=
          1 + - +
          2 - - -
          3 + - - ;
```

(In these examples `month` is a simple set of singlets, `A` is a 2-dimensional array of doublets, and `B` is a simple set of triplets. Data blocks for the same set are equivalent in the sense that they specify the same data in different formats.)

The *set data block* is used to specify a complete elemental set, which is assigned to a set (if it is a simple set) or one of its members (if the set is an array of sets).[1]

Data blocks can be specified only for non-computable sets, i.e. for sets, which have no assign attribute (`:=`) in the corresponding set statements.

If the set is a simple set, only its symbolic name should be specified in the header of the data block. Otherwise, if the set is a n-dimensional array, its symbolic name should be provided with a complete list of subscripts separated by commae and enclosed in square brackets to specify a particular member of the set array. The number of subscripts should be the same as the dimension of the set array, where each subscript should be a number or symbol.

An elemental set defined in the set data block is coded as a sequence of data records described below.[2]

5.2.1 Assign data record

The *assign data record* (`:=`) is a non-signficant element. It may be used for improving readability of data blocks.

5.2.2 Slice data record

The *slice data record* is a control record, which specifies a *slice* of the elemental set defined in the data block. It has the following syntactic form:

$$(\; s_1 \; , \; s_2 \; , \; \dots \; , \; s_n \;)$$

where s_1, s_2, ..., s_n are components of the slice.

Each component of the slice can be a number or symbol or the asterisk (`*`). The number of components in the slice should be the same as the dimension of n-tuples in the elemental set to be defined. For instance, if the elemental set contains 4-tuples (quadruplets), the slice should have four components. The number of asterisks in the slice is called the *slice dimension*.

The effect of using slices is the following. If a m-dimensional slice (i.e. a slice having m asterisks) is specified in the data block, all subsequent data records should specify tuples of the dimension m. Whenever a m-tuple is encountered, each asterisk in the slice is replaced by corresponding components of the m-tuple that gives the resultant n-tuple, which is included in the elemental set to

[1]There is another way to specify data for a simple set along with data for parameters. This feature is discussed in the next section.

[2]*Data record* is simply a technical term. It does not mean that data records have any special formatting.

be defined. For example, if the slice $(a, *, 1, 2, *)$ is in effect, and 2-tuple $(3, b)$ is encountered in a subsequent data record, the resultant 5-tuple included in the elemental set is $(a, 3, 1, 2, b)$.

The slice having no asterisks itself defines a complete n-tuple, which is included in the elemental set.

Being once specified the slice effects until either a new slice or the end of data block is encountered. Note that if no slice is specified in the data block, one, components of which are all asterisks, is assumed.

5.2.3 Simple data record

The *simple data record* defines one n-tuple in a simple format and has the following syntactic form:

$$t_1 \text{ , } t_2 \text{ , } \ldots \text{ , } t_n$$

where t_1, t_2, ..., t_n are components of the n-tuple. Each component can be a number or symbol. Commae between components are optional and may be omitted.

5.2.4 Matrix data record

The *matrix data record* defines several 2-tuples (doublets) in a matrix format and has the following syntactic form:

$$\begin{array}{cccccc}
: & c_1 & c_2 & \ldots & c_n & := \\
r_1 & a_{11} & a_{12} & \ldots & a_{1n} & \\
r_2 & a_{21} & a_{22} & \ldots & a_{2n} & \\
\multicolumn{5}{c}{\dots\dots\dots} & \\
r_m & a_{m1} & a_{m2} & \ldots & a_{mn} &
\end{array}$$

where r_1, r_2, ..., r_m are numbers and/or symbols corresponding to rows of the matrix; c_1, c_2, ..., c_n are numbers and/or symbols corresponding to columns of the matrix, a_{11}, a_{12}, ..., a_{mn} are matrix elements, which can be either + or -. (In this data record the delimiter : preceding the column list and the delimiter := following the column list cannot be omitted.)

Each element a_{ij} of the matrix data block (where $1 \leq i \leq m$, $1 \leq j \leq n$) corresponds to 2-tuple (r_i, c_j). If a_{ij} is the plus sign (+), that 2-tuple (or a longer n-tuple, if a slice is used) is included in the elemental set. Otherwise, if a_{ij} is the minus sign (-), that 2-tuple is not included in the elemental set.

Since the matrix data record defines 2-tuples, either the elemental set should consist of 2-tuples or the slice currently used should be 2-dimensional.

5.2.5 Transposed matrix data record

The *transposed matrix data record* has the following syntactic form:

$$\begin{array}{cccccc}
\texttt{(tr)}\ : & c_1 & c_2 & \ldots & c_n & \texttt{:=} \\
r_1 & a_{11} & a_{12} & \ldots & a_{1n} & \\
r_2 & a_{21} & a_{22} & \ldots & a_{2n} & \\
\multicolumn{5}{c}{\cdots\cdots\cdots\cdots} & \\
r_m & a_{m1} & a_{m2} & \ldots & a_{mn} &
\end{array}$$

(In this case the delimiter : following the keyword `(tr)` is optional and may be omitted.)

This data record is completely analogous to the matrix data record (see above) with only exception that in this case each element a_{ij} of the matrix corresponds to 2-tuple (c_j, r_i) rather than (r_i, c_j).

Being once specified the `(tr)` indicator affects all subsequent data records until either a slice or the end of data block is encountered.

5.3 Parameter data block

param *name* , *record* , . . . , *record* ;

param *name* **default** *value* , *record* , . . . , *record* ;

param : *tabbing-data* ;

param **default** *value* : *tabbing-data* ;

name is a symbolic name of the parameter;

value is an optional default value of the parameter;

record, . . . , *record* are data records;

tabbing-data specifies parameter data in the tabbing format.

Commae preceding data records may be omitted.

Data records

`:=`
is a non-significant data record, which may be used freely to improve readability;

`[` *slice* `]`
specifies a slice;

plain-data
specifies parameter data in the plain format;

: *tabular-data*
specifies parameter data in the tabular format;

`(tr) :` *tabular-data*

specifies set data in the transposed tabular format. (In this case the colon following the keyword `(tr)` may be omitted.)

Examples

```
param T := 4;
param month := 1 Jan 2 Feb 3 Mar 4 Apr 5 May;
param month := [1] 'Jan', [2] 'Feb', [3] 'Mar', [4] 'Apr', [5] 'May';
param init_stock := iron 7.32 nickel 35.8;
param init_stock [*] iron 7.32, nickel 35.8;
param cost [iron] .025 [nickel] .03;
param value := iron -.1, nickel .02;
param      : init_stock  cost  value :=
      iron       7.32     .025   -.1
      nickel    35.8      .03     .02 ;
param : raw : init stock  cost  value :=
        iron     7.32     .025   -.1
        nickel  35.8      .03     .02 ;
param demand default 0 (tr)
        :  FRA  DET  LAN  WIN  STL  FRE  LAF :=
   bands  300   .   100   75    .   225  250
   coils  500  750  400  250    .   850  500
   plate  100   .     .    50  200    .   250 ;
param trans_cost :=
   [*,*,bands]:  FRA  DET  LAN  WIN  STL  FRE  LAF :=
         GARY     30   10    8   10   11   71    6
         CLEV     22    7   10    7   21   82   13
         PITT     19   11   12   10   25   83   15
   [*,*,coils]:  FRA  DET  LAN  WIN  STL  FRE  LAF :=
         GARY     39   14   11   14   16   82    8
         CLEV     27    9   12    9   26   95   17
         PITT     24   14   17   13   28   99   20
   [*,*,plate]:  FRA  DET  LAN  WIN  STL  FRE  LAF :=
         GARY     41   15   12   16   17   86    8
         CLEV     29    9   13    9   28   99   18
         PITT     26   14   17   13   31  104   20 ;
```

The *parameter data block* is used to specify complete data for a parameter (or parameters, if data are specified in the tabbing format).

Data blocks can be specified only for non-computable parameters, i.e. for parameters, which have no assign attribute (`:=`) in the corresponding parameter statements.

Data defined in the parameter data block are coded as a sequence of data records described below. Additionally the data block can be provided with the optional `default` attribute, which specifies a default numeric or symbolic value of the parameter (parameters). This default value is assigned to the parameter or its members when no appropriate value is defined in the parameter data block. The `default` attribute cannot be used, if it is already specified in the corresponding parameter statement.

5.3.1 Assign data record

The *assign data record* (`:=`) is a non-signficant element. It may be used for improving readability of data blocks.

5.3.2 Slice data record

The *slice data record* is a control record, which specifies a *slice* of the parameter array. It has the following syntactic form:

$$\texttt{[}\ s_1\ \texttt{,}\ s_2\ \texttt{,}\ \dots\ \texttt{,}\ s_n\ \texttt{]}$$

where s_1, s_2, ..., s_n are components of the slice.

Each component of the slice can be a number or symbol or the asterisk (`*`). The number of components in the slice should be the same as the dimension of the parameter. For instance, if the parameter is a 4-dimensional array, the slice should have four components. The number of asterisks in the slice is called the *slice dimension*.

The effect of using slices is the following. If a m-dimensional slice (i.e. a slice having m asterisks) is specified in the data block, all subsequent data records should specify subscripts of the parameter members as if the parameter were m-dimensional, not n-dimensional.

Whenever m subscripts are encountered, each asterisk in the slice is replaced by corresponding subscript that gives n subscripts, which define the actual parameter member. For example, if the slice $[a, *, 1, 2, *]$ is in effect, and subscripts 3 and b are encountered in a subsequent data record, the complete subscript list used to choose a parameter member is $[a, 3, 1, 2, b]$.

It is allowed to specify a slice having no asterisks. Such slice itself defines a complete subscript list, in which case the next data record should define only a single value of corresponding parameter member.

Being once specified the slice effects until either a new slice or the end of data block is encountered. Note that if no slice is specified in the data block, one, components of which are all asterisks, is assumed.

5.3.3 Plain data record

The *plain data record* defines a subscript list and a single value in the plain format. This record has the following syntactic form:

$$t_1\ \texttt{,}\ t_2\ \texttt{,}\ \dots\ \texttt{,}\ t_n\ \texttt{,}\ v$$

where t_1, t_2, ..., t_n are subscripts, and v is a value. Each subscript as well as the value can be a number or symbol. Commae following subscripts are optional and may be omitted.

In case of 0-dimensional parameter or slice the plain data record has no subscripts and consists of a single value only.

5.3.4 Tabular data record

The *tabular data record* defines several values, where each value is provided with two subscripts. This record has the following syntactic form:

$$\begin{array}{ccccc} \texttt{:} & c_1 & c_2 & \dots & c_n & \texttt{:=} \\ r_1 & a_{11} & a_{12} & \dots & a_{1n} \\ r_2 & a_{21} & a_{22} & \dots & a_{2n} \\ \multicolumn{5}{c}{.\ .\ .\ .\ .\ .\ .\ .\ .} \\ r_m & a_{m1} & a_{m2} & \dots & a_{mn} \end{array}$$

where r_1, r_2, ..., r_m are numbers and/or symbols corresponding to rows of the table; c_1, c_2, ..., c_n are numbers and/or symbols corresponding to columns of the table, a_{11}, a_{12}, ..., a_{mn} are table elements. Each element can be a number or symbol or the single decimal point (`.`). (In this data record the delimiter `:` preceding the column list and the delimiter `:=` following the column list cannot be omitted.)

Each element a_{ij} of the tabular data block ($1 \leq i \leq m$, $1 \leq j \leq n$) defines two subscripts, where the first subscript is r_i, and the second one is c_j. These subscripts are used in conjunction with the current slice to form the complete subscript list that identifies a particular member of the parameter array. If a_{ij} is a number or symbol, this value is assigned to the parameter member. However, if a_{ij} is the single decimal point, the member is assigned a default value specified either in the parameter data block or in the parameter statement, or, if no default value is specified, the member remains undefined.

Since the tabular data record provides two subscripts for each value, either the parameter or the slice currently used should be 2-dimensional.

5.3.5 Transposed tabular data record

The *transposed tabular data record* has the following syntactic form:

$$\begin{array}{cccccc} \texttt{(tr)}\ \texttt{:} & c_1 & c_2 & \dots & c_n & \texttt{:=} \\ r_1 & a_{11} & a_{12} & \dots & a_{1n} \\ r_2 & a_{21} & a_{22} & \dots & a_{2n} \\ \multicolumn{5}{c}{.\ .\ .\ .\ .\ .\ .\ .\ .} \\ r_m & a_{m1} & a_{m2} & \dots & a_{mn} \end{array}$$

(In this case the delimiter `:` following the keyword `(tr)` is optional and may be omitted.)

This data record is completely analogous to the tabular data record (see above) with only exception that the first subscript defined by element a_{ij} is c_j while the second one is r_i.

Being once specified the `(tr)` indicator affects all subsequent data records until either a slice or the end of data block is encountered.

5.3.6 Tabbing data format

The parameter data block in the *tabbing format* has the following syntactic form:

$$\begin{array}{llllllll}
\texttt{param default}\ \textit{value} : s : & p_1 \texttt{ ,} & p_2 \texttt{ ,} & \ldots \texttt{ ,} & p_r & \texttt{:=} & & \\
r_{11} \texttt{ ,} & r_{12} \texttt{ ,} & \ldots \texttt{ ,} & r_{1n} \texttt{ ,} & a_{11} \texttt{ ,} & a_{12} \texttt{ ,} & \ldots \texttt{ ,} & a_{1r} \texttt{ ,} \\
r_{21} \texttt{ ,} & r_{22} \texttt{ ,} & \ldots \texttt{ ,} & r_{2n} \texttt{ ,} & a_{21} \texttt{ ,} & a_{22} \texttt{ ,} & \ldots \texttt{ ,} & a_{2r} \texttt{ ,} \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
r_{m1} \texttt{ ,} & r_{m2} \texttt{ ,} & \ldots \texttt{ ,} & r_{mn} \texttt{ ,} & a_{m1} \texttt{ ,} & a_{m2} \texttt{ ,} & \ldots \texttt{ ,} & a_{mr} \texttt{ ;}
\end{array}$$

1. The keyword `default` may be omitted along with a value following it.
2. Symbolic name s may be omitted along with the colon following it.
3. All commae are optional and may be omitted.

The data block in the tabbing format shown above is exactly equivalent to the following data blocks:

`set` s `:=` `(`r_{11}`,`r_{12}`,`$\ldots$`,`r_{1n}`)` `(`r_{21}`,`r_{22}`,`$\ldots$`,`r_{2n}`)` $\ldots$ `(`r_{m1}`,`r_{m2}`,`$\ldots$`,`r_{mn}`);`

`param` p_1 `default` *value* `:=`

`[`r_{11}`,`r_{12}`,`$\ldots$`,`r_{1n}`]` a_{11} `[`r_{21}`,`r_{22}`,`$\ldots$`,`r_{2n}`]` a_{21} $\ldots$ `[`r_{m1}`,`r_{m2}`,`$\ldots$`,`r_{mn}`]` a_{m1}`;`

`param` p_2 `default` *value* `:=`

`[`r_{11}`,`r_{12}`,`$\ldots$`,`r_{1n}`]` a_{12} `[`r_{21}`,`r_{22}`,`$\ldots$`,`r_{2n}`]` a_{22} $\ldots$ `[`r_{m1}`,`r_{m2}`,`$\ldots$`,`r_{mn}`]` a_{m2}`;`

. . . .

`param` p_r `default` *value* `:=`

`[`r_{11}`,`r_{12}`,`$\ldots$`,`r_{1n}`]` a_{1r} `[`r_{21}`,`r_{22}`,`$\ldots$`,`r_{2n}`]` a_{2r} $\ldots$ `[`r_{m1}`,`r_{m2}`,`$\ldots$`,`r_{mn}`]` a_{mr}`;`

Appendix A

Using suffixes

Suffixes can be used to retrieve additional values associated with model variables, constraints, and objectives.

A *suffix* consists of a period (`.`) followed by a non-reserved keyword. For example, if `x` is a two-dimensional variable, `x[i,j].lb` is a numeric value equal to the lower bound of elemental variable `x[i,j]`, which (value) can be used everywhere in expressions like a numeric parameter.

For model variables suffixes have the following meaning:

`.lb`	lower bound
`.ub`	upper bound
`.status`	status in the solution: 0 — undefined 1 — basic 2 — non-basic on lower bound 3 — non-basic on upper bound 4 — non-basic free (unbounded) variable 5 — non-basic fixed variable
`.val`	primal value in the solution
`.dual`	dual value (reduced cost) in the solution

For model constraints and objectives suffixes have the following meaning:

`.lb`	lower bound of the linear form
`.ub`	upper bound of the linear form
`.status`	status in the solution: 0 — undefined 1 — non-active 2 — active on lower bound 3 — active on upper bound 4 — active free (unbounded) row 5 — active equality constraint
`.val`	primal value of the linear form in the solution
`.dual`	dual value (reduced cost) of the linear form in the solution

Note that suffixes `.status`, `.val`, and `.dual` can be used only below the solve statement.

Appendix B

Date and time functions

by Andrew Makhorin `<mao@gnu.org>`
and Heinrich Schuchardt `<heinrich.schuchardt@gmx.de>`

B.1 Obtaining current calendar time

To obtain the current calendar time in MathProg there exists the function `gmtime`. It has no arguments and returns the number of seconds elapsed since 00:00:00 on January 1, 1970, Coordinated Universal Time (UTC). For example:

```
param utc := gmtime();
```

MathProg has no function to convert UTC time returned by the function `gmtime` to *local* calendar times. Thus, if you need to determine the current local calendar time, you have to add to the UTC time returned the time offset from UTC expressed in seconds. For example, the time in Berlin during the winter is one hour ahead of UTC that corresponds to the time offset +1 hour = +3600 secs, so the current winter calendar time in Berlin may be determined as follows:

```
param now := gmtime() + 3600;
```

Similarly, the summer time in Chicago (Central Daylight Time) is five hours behind UTC, so the corresponding current local calendar time may be determined as follows:

```
param now := gmtime() - 5 * 3600;
```

Note that the value returned by `gmtime` is volatile, i.e. being called several times this function may return different values.

B.2 Converting character string to calendar time

The function `str2time(`*s*`,` *f*`)` converts a character string (timestamp) specified by its first argument *s*, which should be a symbolic expression, to the calendar time suitable for arithmetic calculations. The conversion is controlled by the specified format string *f* (the second argument), which also should be a symbolic expression.

The result of conversion returned by `str2time` has the same meaning as values returned by the function `gmtime` (see Subsection B.1, page 56). Note that `str2time` does `not` correct the calendar time returned for the local timezone, i.e. being applied to 00:00:00 on January 1, 1970 it always returns 0.

For example, the model statements:

```
    param s, symbolic, := "07/14/98 13:47";
    param t := str2time(s, "%m/%d/%y %H:%M");
    display t;
```

produce the following printout:

```
    t = 900424020
```

where the calendar time printed corresponds to 13:47:00 on July 14, 1998.

The format string passed to the function `str2time` consists of conversion specifiers and ordinary characters. Each conversion specifier begins with a percent (`%`) character followed by a letter.

The following conversion specifiers may be used in the format string:

`%b`	The abbreviated month name (case insensitive). At least three first letters of the month name should appear in the input string.
`%d`	The day of the month as a decimal number (range 1 to 31). Leading zero is permitted, but not required.
`%h`	The same as `%b`.
`%H`	The hour as a decimal number, using a 24-hour clock (range 0 to 23). Leading zero is permitted, but not required.
`%m`	The month as a decimal number (range 1 to 12). Leading zero is permitted, but not required.
`%M`	The minute as a decimal number (range 0 to 59). Leading zero is permitted, but not required.
`%S`	The second as a decimal number (range 0 to 60). Leading zero is permitted, but not required.
`%y`	The year without a century as a decimal number (range 0 to 99). Leading zero is permitted, but not required. Input values in the range 0 to 68 are considered as the years 2000 to 2068 while the values 69 to 99 as the years 1969 to 1999.
`%z`	The offset from GMT in ISO 8601 format.
`%%`	A literal `%` character.

All other (ordinary) characters in the format string should have a matching character in the input string to be converted. Exceptions are spaces in the input string which can match zero or more space characters in the format string.

If some date and/or time component(s) are missing in the format and, therefore, in the input string, the function `str2time` uses their default values corresponding to 00:00:00 on January 1, 1970, that is, the default value of the year is 1970, the default value of the month is January, etc.

The function `str2time` is applicable to all calendar times in the range 00:00:00 on January 1, 0001 to 23:59:59 on December 31, 4000 of the Gregorian calendar.

B.3 Converting calendar time to character string

The function `time2str(`*t*`,` *f*`)` converts the calendar time specified by its first argument *t*, which should be a numeric expression, to a character string (symbolic value). The conversion is controlled by the specified format string *f* (the second argument), which should be a symbolic expression.

The calendar time passed to `time2str` has the same meaning as values returned by the function `gmtime` (see Subsection B.1, page 56). Note that `time2str` does *not* correct the specified calendar time for the local timezone, i.e. the calendar time 0 always corresponds to 00:00:00 on January 1, 1970.

For example, the model statements:

```
param s, symbolic, := time2str(gmtime(), "%FT%TZ");
display s;
```

may produce the following printout:

```
s = '2008-12-04T00:23:45Z'
```

which is a timestamp in the ISO format.

The format string passed to the function `time2str` consists of conversion specifiers and ordinary characters. Each conversion specifier begins with a percent (`%`) character followed by a letter.

The following conversion specifiers may be used in the format string:

`%a`	The abbreviated (2-character) weekday name.
`%A`	The full weekday name.
`%b`	The abbreviated (3-character) month name.
`%B`	The full month name.
`%C`	The century of the year, that is the greatest integer not greater than the year divided by 100.
`%d`	The day of the month as a decimal number (range 01 to 31).
`%D`	The date using the format `%m/%d/%y`.
`%e`	The day of the month like with `%d`, but padded with blank rather than zero.
`%F`	The date using the format `%Y-%m-%d`.
`%g`	The year corresponding to the ISO week number, but without the century (range 00 to 99). This has the same format and value as `%y`, except that if the ISO week number (see `%V`) belongs to the previous or next year, that year is used instead.

`%G`	The year corresponding to the ISO week number. This has the same format and value as `%Y`, except that if the ISO week number (see `%V`) belongs to the previous or next year, that year is used instead.
`%h`	The same as `%b`.
`%H`	The hour as a decimal number, using a 24-hour clock (range 00 to 23).
`%I`	The hour as a decimal number, using a 12-hour clock (range 01 to 12).
`%j`	The day of the year as a decimal number (range 001 to 366).
`%k`	The hour as a decimal number, using a 24-hour clock like `%H`, but padded with blank rather than zero.
`%l`	The hour as a decimal number, using a 12-hour clock like `%I`, but padded with blank rather than zero.
`%m`	The month as a decimal number (range 01 to 12).
`%M`	The minute as a decimal number (range 00 to 59).
`%p`	Either `AM` or `PM`, according to the given time value. Midnight is treated as `AM` and noon as `PM`.
`%P`	Either `am` or `pm`, according to the given time value. Midnight is treated as `am` and noon as `pm`.
`%R`	The hour and minute in decimal numbers using the format `%H:%M`.
`%S`	The second as a decimal number (range 00 to 59).
`%T`	The time of day in decimal numbers using the format `%H:%M:%S`.
`%u`	The day of the week as a decimal number (range 1 to 7), Monday being 1.
`%U`	The week number of the current year as a decimal number (range 00 to 53), starting with the first Sunday as the first day of the first week. Days preceding the first Sunday in the year are considered to be in week 00.
`%V`	The ISO week number as a decimal number (range 01 to 53). ISO weeks start with Monday and end with Sunday. Week 01 of a year is the first week which has the majority of its days in that year; this is equivalent to the week containing January 4. Week 01 of a year can contain days from the previous year. The week before week 01 of a year is the last week (52 or 53) of the previous year even if it contains days from the new year. In other word, if 1 January is Monday, Tuesday, Wednesday or Thursday, it is in week 01; if 1 January is Friday, Saturday or Sunday, it is in week 52 or 53 of the previous year.
`%w`	The day of the week as a decimal number (range 0 to 6), Sunday being 0.
`%W`	The week number of the current year as a decimal number (range 00 to 53), starting with the first Monday as the first day of the first week. Days preceding the first Monday in the year are considered to be in week 00.
`%y`	The year without a century as a decimal number (range 00 to 99), that is the year modulo 100.

`%Y` The year as a decimal number, using the Gregorian calendar.

`%%` A literal `%` character.

All other (ordinary) characters in the format string are simply copied to the resultant string.

The first argument (calendar time) passed to the function `time2str` should be in the range from −62135596800 to +64092211199 that corresponds to the period from 00:00:00 on January 1, 0001 to 23:59:59 on December 31, 4000 of the Gregorian calendar.

Appendix C

Table drivers

by Andrew Makhorin <mao@gnu.org>
and Heinrich Schuchardt <heinrich.schuchardt@gmx.de>

The *table driver* is a program module which provides transmitting data between MathProg model objects and data tables.

Currently the GLPK package has four table drivers:

— built-in CSV table driver;

— built-in xBASE table driver;

— ODBC table driver;

— MySQL table driver.

C.1 CSV table driver

The CSV table driver assumes that the data table is represented in the form of a plain text file in the CSV (comma-separated values) file format as described below.

To choose the CSV table driver its name in the table statement should be specified as `"CSV"`, and the only argument should specify the name of a plain text file containing the table. For example:

```
table data IN "CSV" "data.csv": ... ;
```

The filename suffix may be arbitrary, however, it is recommended to use the suffix '`.csv`'.

On reading input tables the CSV table driver provides an implicit field named `RECNO`, which contains the current record number. This field can be specified in the input table statement as if there were the actual field named `RECNO` in the CSV file. For example:

```
table list IN "CSV" "list.csv": num <- [RECNO], ... ;
```

CSV format[1]

The CSV (comma-separated values) format is a plain text file format defined as follows.

1. Each record is located on a separate line, delimited by a line break. For example:

```
aaa,bbb,ccc\n
xxx,yyy,zzz\n
```

where `\n` means the control character `LF` (`0x0A`).

2. The last record in the file may or may not have an ending line break. For example:

```
aaa,bbb,ccc\n
xxx,yyy,zzz
```

3. There should be a header line appearing as the first line of the file in the same format as normal record lines. This header should contain names corresponding to the fields in the file. The number of field names in the header line should be the same as the number of fields in the records of the file. For example:

```
name1,name2,name3\n
aaa,bbb,ccc\n
xxx,yyy,zzz\n
```

4. Within the header and each record there may be one or more fields separated by commas. Each line should contain the same number of fields throughout the file. Spaces are considered as part of a field and therefore not ignored. The last field in the record should not be followed by a comma. For example:

```
aaa,bbb,ccc\n
```

5. Fields may or may not be enclosed in double quotes. For example:

```
"aaa","bbb","ccc"\n
zzz,yyy,xxx\n
```

6. If a field is enclosed in double quotes, each double quote which is part of the field should be coded twice. For example:

```
"aaa","b""bb","ccc"\n
```

Example

```
FROM,TO,DISTANCE,COST
Seattle,New-York,2.5,0.12
Seattle,Chicago,1.7,0.08
Seattle,Topeka,1.8,0.09
San-Diego,New-York,2.5,0.15
San-Diego,Chicago,1.8,0.10
San-Diego,Topeka,1.4,0.07
```

[1]This material is based on the RFC document 4180.

C.2 xBASE table driver

The xBASE table driver assumes that the data table is stored in the .dbf file format.

To choose the xBASE table driver its name in the table statement should be specified as `"xBASE"`, and the first argument should specify the name of a .dbf file containing the table. For the output table there should be the second argument defining the table format in the form `"FF...F"`, where `F` is either `C(`n`)`, which specifies a character field of length n, or `N(`n`[,`p`])`, which specifies a numeric field of length n and precision p (by default p is 0).

The following is a simple example which illustrates creating and reading a .dbf file:

```
table tab1{i in 1..10} OUT "xBASE" "foo.dbf"
   "N(5)N(10,4)C(1)C(10)": 2*i+1 ~ B, Uniform(-20,+20) ~ A,
   "?" ~ FOO, "[" & i & "]" ~ C;
set S, dimen 4;
table tab2 IN "xBASE" "foo.dbf": S <- [B, C, RECNO, A];
display S;
end;
```

C.3 ODBC table driver

The ODBC table driver allows connecting to SQL databases using an implementation of the ODBC interface based on the Call Level Interface (CLI).[2]

Debian GNU/Linux. Under Debian GNU/Linux the ODBC table driver uses the iODBC package,[3] which should be installed before building the GLPK package. The installation can be effected with the following command:

```
      sudo apt-get install libiodbc2-dev
```

Note that on configuring the GLPK package to enable using the iODBC library the option '`--enable-odbc`' should be passed to the configure script.

The individual databases should be entered for systemwide usage in `/etc/odbc.ini` and `/etc/odbcinst.ini`. Database connections to be used by a single user are specified by files in the home directory (`.odbc.ini` and `.odbcinst.ini`).

Microsoft Windows. Under Microsoft Windows the ODBC table driver uses the Microsoft ODBC library. To enable this feature the symbol:

```
      #define ODBC_DLNAME "odbc32.dll"
```

should be defined in the GLPK configuration file '`config.h`'.

Data sources can be created via the Administrative Tools from the Control Panel.

To choose the ODBC table driver its name in the table statement should be specified as `'ODBC'` or `'iODBC'`.

[2]The corresponding software standard is defined in ISO/IEC 9075-3:2003.

[3]See `<http://www.iodbc.org/>`.

The argument list is specified as follows.

The first argument is the connection string passed to the ODBC library, for example:

`'DSN=glpk;UID=user;PWD=password'`, or

`'DRIVER=MySQL;DATABASE=glpkdb;UID=user;PWD=password'`.

Different parts of the string are separated by semicolons. Each part consists of a pair *fieldname* and *value* separated by the equal sign. Allowable fieldnames depend on the ODBC library. Typically the following fieldnames are allowed:

`DATABASE`	database;
`DRIVER`	ODBC driver;
`DSN`	name of a data source;
`FILEDSN`	name of a file data source;
`PWD`	user password;
`SERVER`	database;
`UID`	user name.

The second argument and all following are considered to be SQL statements

SQL statements may be spread over multiple arguments. If the last character of an argument is a semicolon this indicates the end of a SQL statement.

The arguments of a SQL statement are concatenated separated by space. The eventual trailing semicolon will be removed.

All but the last SQL statement will be executed directly.

For IN-table the last SQL statement can be a SELECT command starting with the capitalized letters `'SELECT '`. If the string does not start with `'SELECT '` it is considered to be a table name and a SELECT statement is automatically generated.

For OUT-table the last SQL statement can contain one or multiple question marks. If it contains a question mark it is considered a template for the write routine. Otherwise the string is considered a table name and an INSERT template is automatically generated.

The writing routine uses the template with the question marks and replaces the first question mark by the first output parameter, the second question mark by the second output parameter and so forth. Then the SQL command is issued.

The following is an example of the output table statement:

```
table ta { l in LOCATIONS } OUT
   'ODBC'
   'DSN=glpkdb;UID=glpkuser;PWD=glpkpassword'
   'DROP TABLE IF EXISTS result;'
   'CREATE TABLE result ( ID INT, LOC VARCHAR(255), QUAN DOUBLE );'
   'INSERT INTO result 'VALUES ( 4, ?, ? )' :
   l ~ LOC, quantity[l] ~ QUAN;
```

Alternatively it could be written as follows:

```
table ta { l in LOCATIONS } OUT
   'ODBC'
   'DSN=glpkdb;UID=glpkuser;PWD=glpkpassword'
   'DROP TABLE IF EXISTS result;'
   'CREATE TABLE result ( ID INT, LOC VARCHAR(255), QUAN DOUBLE );'
   'result' :
   l ~ LOC, quantity[l] ~ QUAN, 4 ~ ID;
```

Using templates with '?' supports not only INSERT, but also UPDATE, DELETE, etc. For example:

```
table ta { l in LOCATIONS } OUT
   'ODBC'
   'DSN=glpkdb;UID=glpkuser;PWD=glpkpassword'
   'UPDATE result SET DATE = ' & date & ' WHERE ID = 4;'
   'UPDATE result SET QUAN = ? WHERE LOC = ? AND ID = 4' :
   quantity[l], l;
```

C.4 MySQL table driver

The MySQL table driver allows connecting to MySQL databases.

Debian GNU/Linux. Under Debian GNU/Linux the MySQL table driver uses the MySQL package,[4] which should be installed before building the GLPK package. The installation can be effected with the following command:

```
sudo apt-get install libmysqlclient15-dev
```

Note that on configuring the GLPK package to enable using the MySQL library the option '`--enable-mysql`' should be passed to the configure script.

Microsoft Windows. Under Microsoft Windows the MySQL table driver also uses the MySQL library. To enable this feature the symbol:

```
#define MYSQL_DLNAME "libmysql.dll"
```

should be defined in the GLPK configuration file '`config.h`'.

To choose the MySQL table driver its name in the table statement should be specified as `'MySQL'`.

The argument list is specified as follows.

The first argument specifies how to connect the data base in the DSN style, for example:

`'Database=glpk;UID=glpk;PWD=gnu'`.

Different parts of the string are separated by semicolons. Each part consists of a pair *fieldname* and *value* separated by the equal sign. The following fieldnames are allowed:

[4]For download development files see `<http://dev.mysql.com/downloads/mysql/>`.

`Server`	server running the database (defaulting to localhost);
`Database`	name of the database;
`UID`	user name;
`PWD`	user password;
`Port`	port used by the server (defaulting to 3306).

The second argument and all following are considered to be SQL statements.

SQL statements may be spread over multiple arguments. If the last character of an argument is a semicolon this indicates the end of a SQL statement.

The arguments of a SQL statement are concatenated separated by space. The eventual trailing semicolon will be removed.

All but the last SQL statement will be executed directly.

For IN-table the last SQL statement can be a SELECT command starting with the capitalized letters `'SELECT '`. If the string does not start with `'SELECT '` it is considered to be a table name and a SELECT statement is automatically generated.

For OUT-table the last SQL statement can contain one or multiple question marks. If it contains a question mark it is considered a template for the write routine. Otherwise the string is considered a table name and an INSERT template is automatically generated.

The writing routine uses the template with the question marks and replaces the first question mark by the first output parameter, the second question mark by the second output parameter and so forth. Then the SQL command is issued.

The following is an example of the output table statement:

```
table ta { l in LOCATIONS } OUT
   'MySQL'
   'Database=glpkdb;UID=glpkuser;PWD=glpkpassword'
   'DROP TABLE IF EXISTS result;'
   'CREATE TABLE result ( ID INT, LOC VARCHAR(255), QUAN DOUBLE );'
   'INSERT INTO result VALUES ( 4, ?, ? )' :
   l ~ LOC, quantity[l] ~ QUAN;
```

Alternatively it could be written as follows:

```
table ta { l in LOCATIONS } OUT
   'MySQL'
   'Database=glpkdb;UID=glpkuser;PWD=glpkpassword'
   'DROP TABLE IF EXISTS result;'
   'CREATE TABLE result ( ID INT, LOC VARCHAR(255), QUAN DOUBLE );'
   'result' :
   l ~ LOC, quantity[l] ~ QUAN, 4 ~ ID;
```

Using templates with '?' supports not only INSERT, but also UPDATE, DELETE, etc. For example:

```
table ta { l in LOCATIONS } OUT
   'MySQL'
   'Database=glpkdb;UID=glpkuser;PWD=glpkpassword'
   'UPDATE result SET DATE = ' & date & ' WHERE ID = 4;'
   'UPDATE result SET QUAN = ? WHERE LOC = ? AND ID = 4' :
   quantity[l], l;
```

Appendix D

Solving models with glpsol

The GLPK package[1] includes the program `glpsol`, a stand-alone LP/MIP solver. This program can be launched from the command line or from the shell to solve models written in the GNU MathProg modeling language.

To tell the solver that the input file contains a model description you need to specify the option `--model` in the command line. For example:

```
glpsol --model foo.mod
```

Sometimes it is necessary to use the data section placed in a separate file, in which case you may use the following command:

```
glpsol --model foo.mod --data foo.dat
```

Note that if the model file also contains the data section, that section is ignored.

It is also allowed to specify more than one file containing the data section, for example:

```
glpsol --model foo.mod --data foo1.dat --data foo2.dat
```

If the model description contains some display and/or printf statements, by default the output is sent to the terminal. If you need to redirect the output to a file, you may use the following command:

```
glpsol --model foo.mod --display foo.out
```

If you need to look at the problem, which has been generated by the model translator, you may use the option `--wlp` as follows:

```
glpsol --model foo.mod --wlp foo.lp
```

In this case the problem data is written to file `foo.lp` in CPLEX LP format suitable for visual analysis.

Sometimes it is needed merely to check the model description not solving the generated problem instance. In this case you may specify the option `--check`, for example:

```
glpsol --check --model foo.mod --wlp foo.lp
```

[1] `http://www.gnu.org/software/glpk/`

If you need to write a numeric solution obtained by the solver to a file, you may use the following command:

```
glpsol --model foo.mod --output foo.sol
```

in which case the solution is written to file `foo.sol` in a plain text format suitable for visual analysis.

The complete list of the `glpsol` options can be found in the GLPK reference manual included in the GLPK distribution.

Appendix E

Example model description

E.1 Model description written in MathProg

Below here is a complete example of the model description written in the GNU MathProg modeling language.

```
# A TRANSPORTATION PROBLEM
#
# This problem finds a least cost shipping schedule that meets
# requirements at markets and supplies at factories.
#
#  References:
#              Dantzig G B, "Linear Programming and Extensions."
#              Princeton University Press, Princeton, New Jersey, 1963,
#              Chapter 3-3.

set I;
/* canning plants */

set J;
/* markets */

param a{i in I};
/* capacity of plant i in cases */

param b{j in J};
/* demand at market j in cases */

param d{i in I, j in J};
/* distance in thousands of miles */
```

```
param f;
/* freight in dollars per case per thousand miles */

param c{i in I, j in J} := f * d[i,j] / 1000;
/* transport cost in thousands of dollars per case */

var x{i in I, j in J} >= 0;
/* shipment quantities in cases */

minimize cost: sum{i in I, j in J} c[i,j] * x[i,j];
/* total transportation costs in thousands of dollars */

s.t. supply{i in I}: sum{j in J} x[i,j] <= a[i];
/* observe supply limit at plant i */

s.t. demand{j in J}: sum{i in I} x[i,j] >= b[j];
/* satisfy demand at market j */

data;

set I := Seattle San-Diego;

set J := New-York Chicago Topeka;

param a := Seattle     350
           San-Diego   600;

param b := New-York    325
           Chicago     300
           Topeka      275;

param d :              New-York   Chicago   Topeka :=
           Seattle     2.5        1.7       1.8
           San-Diego   2.5        1.8       1.4  ;

param f := 90;

end;
```

E.2 Generated LP problem instance

Below here is the result of the translation of the example model produced by the solver `glpsol` and written in CPLEX LP format with the option `--wlp`.

```
\* Problem: transp *\

Minimize
 cost: + 0.225 x(Seattle,New~York) + 0.153 x(Seattle,Chicago)
 + 0.162 x(Seattle,Topeka) + 0.225 x(San~Diego,New~York)
 + 0.162 x(San~Diego,Chicago) + 0.126 x(San~Diego,Topeka)

Subject To
 supply(Seattle): + x(Seattle,New~York) + x(Seattle,Chicago)
 + x(Seattle,Topeka) <= 350
 supply(San~Diego): + x(San~Diego,New~York) + x(San~Diego,Chicago)
 + x(San~Diego,Topeka) <= 600
 demand(New~York): + x(Seattle,New~York) + x(San~Diego,New~York) >= 325
 demand(Chicago): + x(Seattle,Chicago) + x(San~Diego,Chicago) >= 300
 demand(Topeka): + x(Seattle,Topeka) + x(San~Diego,Topeka) >= 275

End
```

E.3 Optimal LP solution

Below here is the optimal solution of the generated LP problem instance found by the solver `glpsol` and written in plain text format with the option `--output`.

```
Problem:    transp
Rows:       6
Columns:    6
Non-zeros:  18
Status:     OPTIMAL
Objective:  cost = 153.675 (MINimum)

   No.   Row name   St   Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------- ------------- ------------- -------------
     1 cost         B        153.675
     2 supply[Seattle]
                    NU           350                         350         < eps
     3 supply[San-Diego]
                    B            550                         600
     4 demand[New-York]
                    NL           325           325                       0.225
     5 demand[Chicago]
                    NL           300           300                       0.153
     6 demand[Topeka]
                    NL           275           275                       0.126
   No. Column name  St   Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------- ------------- ------------- -------------
     1 x[Seattle,New-York]
                    B             50             0
     2 x[Seattle,Chicago]
                    B            300             0
     3 x[Seattle,Topeka]
                    NL             0             0                       0.036
     4 x[San-Diego,New-York]
                    B            275             0
     5 x[San-Diego,Chicago]
                    NL             0             0                       0.009
     6 x[San-Diego,Topeka]
                    B            275             0

End of output
```

Acknowledgements

The authors would like to thank the following people, who kindly read, commented, and corrected the draft of this document:

Juan Carlos Borras <borras@cs.helsinki.fi>

Harley Mackenzie <hjm@bigpond.com>

Robbie Morrison <robbie@actrix.co.nz>