GNU Linear Programming Kit

Modeling Language GNU MathProg

Version 4.7

(Draft Edition, August 2004)

Contents

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 following formulation of linear programming (LP) problem is assumed:

minimize (maximize)

$$Z = c_1x_{m+1} + c_2x_{m+2} + \ldots + c_nx_{m+n} + c_0 \tag{1}$$

subject to linear constraints

$$\begin{array}{r@{\;}l} x_1 = & a_{11}x_{m+1} + a_{12}x_{m+2} + \ldots + a_{1n}x_{m+n} \\ x_2 = & a_{21}x_{m+1} + a_{22}x_{m+2} + \ldots + a_{2n}x_{m+n} \\ & \ldots\ldots\ldots\ldots\ldots\ldots \\ x_m = & a_{m1}x_{m+1} + a_{m2}x_{m+2} + \ldots + a_{mn}x_{m+n} \end{array} \tag{2}$$

and bounds of variables

$$\begin{array}{c} l_1 \leq x_1 \leq u_1 \\ l_2 \leq x_2 \leq u_2 \\ \ldots\ldots\ldots \\ l_{m+n} \leq x_{m+n} \leq u_{m+n} \end{array} \tag{3}$$

where: $x_1, x_2, \ldots, x_m$ are auxiliary variables; $x_{m+1}, x_{m+2}, \ldots, x_{m+n}$ are structural variables; Z is the objective function; $c_1, c_2, \ldots, c_n$ are coefficients of the objective function; c_0 is the constant term of the objective function; a_{11}, a_{12}, $\ldots$, a_{mn} are constraint coefficients; $l_1, l_2, \ldots, l_{m+n}$ are lower bounds of variables; $u_1, u_2, \ldots, u_{m+n}$ are upper bounds of variables.

Bounds of variables can be finite as well as infinite. Besides, lower and upper bounds can be equal to each other. Thus, the following types of variables are possible:

Bounds of variable	Type of variable
$-\infty < x_k < +\infty$	Free (unbounded) variable
$l_k \leq x_k < +\infty$	Variable with lower bound
$-\infty < x_k \leq u_k$	Variable with upper bound
$l_k \leq x_k \leq u_k$	Double-bounded variable
$l_k = x_k = u_k$	Fixed variable

Note that the types of variables shown above are applicable to structural as well as to auxiliary variables.

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

[1]The GNU MathProg language is a subset of the AMPL language.

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 that 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

$$A : \Delta \rightarrow \Xi, \tag{4}$$

where $\Delta \subseteq S_1 \times S_2 \times \ldots \times S_n$ is a subset of the Cartesian product of indexing sets, Ξ is a set of the array members. In MathProg the set Δ is called *subscript domain*. Its members are n-tuples $(i_1, i_2, \ldots, i_n)$, where $i_1 \in S_1$, $i_2 \in S_2$, $\ldots$, $i_n \in S_n$.

If $n = 0$, the Cartesian product in (4) has exactly one element (namely, 0-tuple), so it is convenient to think scalar objects as 0-dimensional arrays that have one member.

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

Model object	Array member
Set	Elemental plain set
Parameter	Number or symbol
Variable	Elemental structural 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 2-dimensional parameter built 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 to be solved using that model. For this reason in MathProg the model description consists of two parts: model section and data section.

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.

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

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

2 Coding model description

Model description is coded in plain text format using ASCII character set. Valid characters acceptable in the model description are the following:

- alphabetic characters: `A B ... Z a b ... z _`
- numeric characters: `0 1 ... 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

Symbolic name consists of alphabetic and numeric characters, the first of which must 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) must be unique, i.e. the model description must have no objects with the same name. Symbolic names of dummy indices must be unique within the scope, where they are valid.

2.2 Numeric literals

Numeric literal has the form *xx*E*syy*, where *xx* is a real number with optional decimal point, *s* is the sign `+` or `-`, *yy* is an integer 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

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

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

Examples:

```
'This is a string'
"This is another string"
'1 + 2 = 3'
'That''s all'
"She said: ""No"""
```

String literals are used to represent symbolic quantities.

2.4 Keywords

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 being recognized by context can be used as symbolic names.

Reserved keywords are:

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

Delimiter is either a single special character or a sequence of two special characters as follows:

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

If delimiter consists of two characters, there must 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 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 between `/*` and `*/`.

Examples:

```
set s{1..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.

3 Expressions

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

Numeric expression is a rule for computing a single numeric value represented in the form of 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 and p >= 1 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)
```

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

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

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

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

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

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

Each subscript must be a numeric or symbolic expression. The number of subscripts in the subscript list must 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.

Function references. In MathProg there are the following built-in functions which may be used in numeric expressions:

`abs`(x)	absolute value
`ceil`(x)	smallest integer not less than x ("ceiling of x")
`floor`(x)	largest integer not greater than x ("floor of x")
`exp`(x)	base-e exponential e^x
`log`(x)	natural logarithm $\log x$
`log10`(x)	common (decimal) logarithm $\log_{10} x$
`max`$(x_1, x_2, \dots, x_n)$	the largest of values $x_1, x_2, \dots, x_n$
`min`$(x_1, x_2, \dots, x_n)$	the smallest of values $x_1, x_2, \dots, x_n$
`round`(x)	rounding x to nearest integer
`round`(x, n)	rounding x to n fractional decimal digits
`sqrt`(x)	square root $\sqrt{x}$
`trunc`(x)	truncating x to nearest integer
`trunc`(x, n)	truncating x to n fractional decimal digits
`Irand224`$()$	pseudo-random integer uniformly distributed in $[0, 2^{24})$
`Uniform01`$()$	pseudo-random number uniformly distributed in $[0, 1)$
`Uniform`(a, b)	pseudo-random number uniformly distributed in $[a, b)$
`Normal01`$()$	Gaussian pseudo-random variate with $\mu = 0$ and $\sigma = 1$
`Normal`(μ, σ)	Gaussian pseudo-random variate with given μ and σ

Arguments of all built-in functions (if required) must be numeric 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 have 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.

Iterated expressions. 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 are four iterated operators, which may be used in numeric expressions:

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

where $i_1,\dots,i_n$ 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, $x(i_1,\dots,i_n)$ 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.

Conditional expressions. Conditional numeric expression is a primary numeric expression, which has 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 is true, the value of the conditional expression is the value of the expression that follows the keyword `then`. Otherwise, if the logical expression has the value false, the value of the conditional expression is the value of the expression that follows the keyword `else`. If the reduced form of the conditional expression is used and the logical expression has the value false, the resultant value of the conditional expression is zero.

Parenthesized expressions. Any numeric expression may be enclosed in parentheses that syntactically makes it 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.

Arithmetic operators. In MathProg there are 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 (raise 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.

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 (`+`, `-`, `diff`)	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

Symbolic expression is a rule for computing a single symbolic value represented in the form of character string.

The primary symbolic expression may be a string literal, dummy index, unsubscripted parameter, subscripted parameter, 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
`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 entirely analogous to that that given for numeric expressions (see above).

Symbolic operators. Currently in MathProg there is the only symbolic operator:

$$x \ \texttt{\&} \ y$$

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

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 explained in Subsection "Numeric expressions".

3.3 Indexing expressions and dummy indices

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

$$\begin{array}{l} \texttt{\{}entry_1,\ entry_2,\ \ldots,\ entry_m\texttt{\}} \\ \texttt{\{}entry_1,\ entry_2,\ \ldots,\ entry_m\ :\ predicate\texttt{\}} \end{array} \qquad (5)$$

where $entry_1,\ entry_2,\ \ldots,\ entry_m$ are indexing entries, *predicate* is a logical expression which specifies optional predicate.

Each indexing entry in the indexing expression have the following three forms:

$$\begin{array}{l} t\ \texttt{in}\ S \\ (t_1, t_2, \ldots, t_k)\ \texttt{in}\ S \\ S \end{array} \qquad (6)$$

where $t, t_1, t_2, \ldots, t_k$ are indices, S is a set expression which specifies the basic set.

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

If the first form of the indexing entry is used, the index t can be a dummy index only. If the second form is used, the indices $t_1, t_2, \ldots, t_k$ can be either dummy indices or some numeric or symbolic expressions, where at least one index must be a dummy index. The third, reduced form of the indexing entry has the same effect as if there were t (if S is 1-dimensional) or $t_1, t_2, \ldots, t_k$ (if S is n-dimensional) all specified as dummy indices.

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 don't need to be explicitly declared. Each *undeclared* symbolic name being used in indexing position of some indexing entry is recognized as 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:

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

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

3. 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 three sets:

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

where A and B consist of 1-tuples (singles), C consists of 2-tuples (doubles). And consider the following indexing expression:

```
{i in A, (j,k) in B, l in C}                    (8)
```

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 (8) is the following:

```
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 output would look like follows:

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

Let the indexing expression (8) be used in the following iterated operation:

```
sum{i in A, (j,k) in B, l in C} p[i,j,k,l] ** 2                (9)
```

where $p[i,j,k,l]$ may be 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 (9) is:

$$\sum_{i\in A,(j,k)\in B,l\in C}(p_{ijkl})^2.$$

Now let the indexing expression (8) be used as a primary set expression. In this case the action is gathering all 4-tuples (quadruples) 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,l are not referenced and therefore their symbolic names are not needed.

Finally, let the indexing expression (8) be used as the subscript domain in declaration of some 4-dimensional model object, for instance, some numeric parameter:

```
par 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 (6) may be numeric or symbolic expressions, not 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, which satisfy these conditions.

Consider, for example, the following indexing expression:

```
{i in A, (i-1,k) in B, l in C}                (10)
```

where i,k,l are dummy indices, and $i-1$ is a numeric expression. The algorithmic decsription of the indexing expression (10) 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 the indexing expression (10) is used as a primary set expression, the resultant set is 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 (10) 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 indexing 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 many cases it is needed to select a subset from some set of from the Cartesian product of some sets. This may be attained by using an optional logical predicate, which is specified in indexing expression after the last or the only indexing entry.

Consider, for example, the following indexing expression:

```
{i in A, (j,k) in B, l in C: i <= 5 and k <> 'Mar'}          (11)
```

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 the indexing expression (11) is used as a primary set expression, the resultant set is 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 indexing expression, one is assumed whose value is true.

3.4 Set expressions

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'})
```

Literal sets. 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 (singles) enumerated within the curly braces. It is allowed to specify an empty set, 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 must have the same number of components.

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

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

$$name[i_1, i_2, \ldots, i_n]$$

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

Each subscript must be a numeric or symbolic expression. The number of subscripts in the subscript list must 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.

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

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

where t_0, t_1, and δt are numeric expressions (the value of δt must 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:

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

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

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

Indexing expressions. If the primary set expression is an indexing expression, the resultant set is determined as described in Subsection "Indexing expressions and dummy indices" (see above).

Iterated expressions. 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:

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

where x is a value of the integrand, $i_1, \ldots, i_n$ 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:

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

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

Conditional expressions. 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 must 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 is true, the resultant set is the value of the expression that follows the keyword `then`. Otherwise, if the logical expression has the value false, the resultant set is the value of the expression that follows the keyword `else`.

Parenthesized expressions. Any set expression may be enclosed in parentheses that syntactically makes it 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.

Set operators. In MathProg there are 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 `inter` Y	intersection $X \cap Y$
X `cross` Y	cross (Cartesian) product $X \times Y$

where X and Y are set expressions, which must define sets of the identical dimension (except for 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 dimension of the operands.

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 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.

3.5 Logical expressions

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
```

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 not zero, otherwise the resultant value of the logical expression is false.

Relational expressions. In MathProg there are 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, x `!=` y	test on $x \neq y$
x `in` Y	test on $x \in Y$
$(x_1, \ldots, x_n)$ `in` Y	test on $(x_1, \ldots, x_n) \in Y$
x `not in` Y, x `!in` Y	test on $x \notin Y$
$(x_1, \ldots, x_n)$ `not in` Y, $(x_1, \ldots, x_n)$ `!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 $x, x_1, \ldots, x_n, y$ are numeric or symbolic expressions, X and Y are set expression.

Notes:

1. If x and y are symbolic expressions, only the relational operators `=`, `==`, `<>`, and `!=` can be used.

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

3. In the operations `within`, `not within`, and `!within` both operands must have identical dimension.

All the relational operators have their conventional mathematical meaning. The resultant value takes on the value true if the corresponding relation is satisfied for its operands, otherwise false.

Iterated expressions. 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 logical expression that participates in the operation.

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

`forall`	$\forall$-quantification	$\forall(i_1,\dots,i_n)_{\in\Delta}[x(i_1,\dots,i_n)]$
`exists`	$\exists$-quantification	$\exists(i_1,\dots,i_n)_{\in\Delta}[x(i_1,\dots,i_n)]$

where $i_1,\dots,i_n$ 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, $x(i_1,\dots,i_n)$ is the integrand, a logical expression whose resultant value depends on the dummy indices.

For $\forall$-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 $\exists$-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.

Parenthesized expressions. Any logical expression may be enclosed in parentheses that syntactically makes it 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.

Logical operators. In MathProg there are the following logical operators, which may be used in logical expressions:

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

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 operations (see below).

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

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 explained in Subsection "Numeric expressions".

3.6 Linear expressions

Linear expression is a rule for computing so called *linear form* or simply *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 the formula that consists of the only constant term.

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])
```

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

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

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

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

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

Actual values of 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 the corresponding member.

Iterated expressions. Iterated linear expression is a primary linear expression, which has the following syntactic form:

`sum` *indexing-expression 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 Subsection “Numeric expressions” above) with the exception that integrand participated in the summation is a formula, not a numeric value.

Conditional expressions. Conditional linear expression is a primary linear expression, which has the following two syntactic forms:

`if` b `then` f `else` g
`if` b `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 "Numeric expressions" above) with the exception that operands participated in the operation are formulae, not numeric values.

Parenthesized expressions. Any linear expression may be enclosed in parentheses that syntactically makes it 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.

Arithmetic operators. In MathProg there are 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 numeric expression.

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.

Hierarchy of operations. The hierarchy of arithmetic operations used in linear expressions is the same as for numeric expressions (for details see Subsection "Numeric expressions").

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, and objective statement) are used to declare model objects of certain kinds and define certain properties of that objects.

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

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

4.1 Set statement

General Form

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

Where:	*name* is the symbolic name of a set.
	alias is an optional string literal which specifies the alias of the set.
	domain is an optional indexing expression which specifies the subscript domain of the set.
	attrib, . . . , *attrib* are optional attributes of the set. (Commae preceding attributes may be omitted.)

The attributes are:

`dimen` *n*	specifies 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 this 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 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* must be unsigned integer from 1 to 20. At most one `dimen` attribute can be specified. If the `dimen` attribute is not specified, 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 the 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 this 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 must 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

General Form

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

Where:	*name* is the symbolic name of a parameter.
	alias is an optional string literal which specifies the alias of the parameter.
	domain is an optional indexing expression which specifies the subscript domain of the parameter.
	attrib, . . . , *attrib* are optional attributes of the parameter. (Commae preceding attributes may be omitted.)

The attributes are:

`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 this condition.
`in` *expression*	specifies a superset that restricts the parameter or its members to be in this 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 the 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 values which 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 must precede all other attributes.

The condition attribute specifies an optional condition that restricts values assigned to the parameter to satisfy this 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 $x > v$
`<>` 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. If the parameter is symbolic, conditions in the form of inequality cannot be specified. 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 specified condition, an error is raised.

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 this 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 specified superset, 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 must 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

General Form

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

Where:	*name* is the symbolic name of a variable.
	alias is an optional string literal which specifies the alias of the variable.
	domain is an optional indexing expression which specifies the subscript domain of the variable.
	attrib, . . . , *attrib* are optional attributes of the variable. (Commae preceding attributes may be omitted.)

The attributes are:

`integer`	restricts the variable to be integer.
`binary`	restricts the variable to be binary.
`>=` *expression*	specifies a lower bound of the variable.
`<=` *expression*	specifies an upper bound of the variable.
`=` *expression*, `==` *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 the 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 structural variables in the LP/MIP problem formulation (see Subsection "Linear programming problem"). Should note that only the elemental variables actually referenced in some constraints and 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 the lower bound of the variable. At most one lower bound can be specified. By default variables (except binary ones) have no lower bounds, 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 the upper bound of the variable. At most one upper bound attribute can be specified.

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

4.4 Constraint statement

General Form

`subject to` *name alias domain* : *expression* , `=` *expression* ;

`subject to` *name alias domain* : *expression* , `<=` *expression* ;

`subject to` *name alias domain* : *expression* , `>=` *expression* ;

`subject to` *name alias domain* : *expression* , `<=` *expression* , `<=` *expression* ;

`subject to` *name alias domain* : *expression* , `>=` *expression* , `>=` *expression* ;

Where:	*name* is the symbolic name of a constraint. *alias* is an optional string literal which specifies the alias of the constraint. *domain* is an optional indexing expression which specifies the subscript domain of the constraint. *expressions* are linear expressions for computing components of the constraint. (Commae following expressions may be omitted.)
Note:	The keyword `subject to` may be reduced to `subj to`, or to `s.t.` or 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 the 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 general linear constraints in the LP/MIP problem formulation (see Subsection "Linear programming problem").

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 linear expression while the leftmost and rightmost ones can be only numeric expressions.

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

Constructing the actual linear constraint included in the problem instantce, 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:

$$\begin{aligned} f &= a_1x_{m+1} + a_2x_{m+2} + \ldots + a_nx_{m+n} + a_0, \\ g &= b_1x_{m+1} \;+ b_2x_{m+2} + \ldots + b_nx_{m+n} \;+ b_0, \end{aligned}$$

where $x_{m+1}, x_{m+2}, \ldots, x_{m+n}$ 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:

$$x_i : (a_1 - b_1)x_{m+1} + (a_2 - b_2)x_{m+2} + \ldots + (a_n - b_n)x_{m+n} \left\{ \begin{array}{c} = \\ \leq \\ \geq \end{array} \right\} b_0 - a_0,$$

where x_i is the implicit auxiliary variable of the constraint (see Subsection "Linear programming problem").

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

$$f = a_1x_{m+1} + a_2x_{m+2} + \ldots + a_nx_{m+n} + a_0,$$

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

$$l - a_0 \leq x_i : a_1x_{m+1} + a_2x_{m+2} + \ldots + a_nx_{m+n} \leq u - a_0,$$

where x_i is the implicit auxiliary variable of the constraint.

4.5 Objective statement

General Form

`minimize` *name alias domain* : *expression* ;

`maximize` *name alias domain* : *expression* ;

Where:	*name* is the symbolic name of an objective.
	alias is an optional string literal which specifies the objective alias.
	domain is an optional indexing expression which specifies the subscript domain of the objective.
	expression is an linear expression which specifies 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 the 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 Subsection "Linear programming problem"). However, unlike constraints the corresponding linear forms are free (unbounded).

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

$$f = a_1 x_{m+1} + a_2 x_{m+2} + \ldots + a_n x_{m+n} + a_0,$$

where $x_{m+1}, x_{m+2}, \ldots, x_{m+n}$ 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:

$$-\infty \leq x_i : a_1 x_{m+1} + a_2 x_{m+2} + \ldots + a_n x_{m+n} + a_0 \leq +\infty,$$

where x_i is the implicit free (unbounded) auxiliary variable of the constraint.

As a rule the model description contains only one objective statement that defines the objective function (1) 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 don't affect the objective function.

4.6 Solve statement

General Form

```
solve ;
```

Note:	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.

Examples:

```
solve;
```

The solve statement causes solving the model, that is 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 variable, constraint, and objective statements cannot be used below the solve statement, i.e. all principal components of the model must be described above the solve statement.

4.7 Check statement

General Form

`check` *domain* : *expression* ;

Where: *domain* is an optional indexing expression which specifies the 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, the model translator reports an error.

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

4.8 Display statement

General Form

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

Where: *domain* is an optional indexing expression which specifies the 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 in plain text format.

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

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. the members actually generated during the model evaluation) is displayed. Note that if the display statement is used above the solve statement and the item is a variable, its displayed "value" means "elemental variable", not a numeric value, which the variable could have in some solution obtained by the solver. To display a numeric value of a variable the display statement should be used below the solve statement. Analogously, if the item is a constraint or objective, its "value" means "elemental constraint" or "elemental objective", not a numeric value.

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

4.9 Printf statement

General Form

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

Where: *domain* is an optional indexing expression which specifies the subscript domain of the printf statement.
format is a symbolic expression whose value specifies a format control string.
expression , . . . , *expression* are zero or more expressions whose values have to be formatted and printed. Each expression must be of numeric, symbolic, or logical type. (The colon preceding the format expression may be omitted.)

Examples:

```
printf 'Hello, world!\n';
printf: "x = %.3f; y = %.3f; z = %.3f\n", x, y, z;
printf{i in I, j in J}: "flow from %s to %s is %d\n", i, j, x[i,j];
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 the user specifying a format control string used to format data written on the standard output.

If the subscript domain is not specified, the printf statement is executed only once. Specifying the subscript domain causes executing the printf statement for every n-tuple in the domain set. In the latter case *format* and *expressions* may include dummy indices introduced in the 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: ordinary characters (not %), which are copied unchanged to the standard output, and conversion specifications, each of which causes evaluating of corresponding *expression* specified in the printf statement, formatting it, and writing the resultant value to the standard output.

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

4.10 For statement

General Form

`for` *domain* : *statement*

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

Where:	*domain* is an indexing expression which specifies the subscript domain of the for statement. (The colon following the indexing expression may be omitted.) *statement* is a statement which has to be executed under control of the for statement. *statement* ... *statement* is a sequence of statements (enclosed in curly braces) which have to be executed under control of the for statement.
Note:	Only following statements are allowed 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 executing a statement or a sequence of statements specified as part of the for statement for every n-tuple in the domain set. Thus, statements within the for statement may refer to dummy indices introduced in the corresponding indexing expression.

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 other way, when the model description is divided into two parts: model section and data section.

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.

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.

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;
```

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

```
statement          data;
statement          data block
...                data block
statement          ...
end;               data block
                   end;
   Model file         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 subsections. The order, in which data blocks follow in the data section, may be arbitrary, not necessarily the same as in which the 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 (for details see Section "Coding model description"), 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 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 (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 must 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

General Form

`set` *name* `,` *record* `,` ... `,` *record* `;`

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

Where:	*name* is the symbolic name of a set.
	symbol, ..., *symbol* are subscripts that specify a particular member of the set (if the set is an array, i.e. set of sets).
	record, ..., *record* are data records.
Note:	Commae preceding data records may be omitted.

The data records are:

`:=`	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 2 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 the set `month` is a simple set of singles, `A` is a 2-dimensional array of doubles, and `B` is a simple set of triples. 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).[2]

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

If the set is a simple set, only its symbolic name should be given 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 must be the same as the dimension of the set array, where each subscript must be a number or symbol.

The elemental set defined in the set data block is coded as a sequence of data records described below.[3]

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

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:

$$(s_1, s_2, \ldots, s_n)$$

where $s_1, s_2, \ldots, s_n$ 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 must 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 (quadruples), the slice must have four components. The number of asterisks in the slice is called *slice dimension*.

The effect of using slices is the following. If a m-dimensional slice (i.e. a slice that has m asterisks) is specified in the data block, all subsequent data records must specifiy tuples of the dimension m. Whenever a m-tuple is encountered, each asterisk in the slice

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

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

is replaced by corresponding components of the m-tuple that gives the resultant n-tuple, which is included in the elemental set to be defined. For example, if the slice $(a, *, 1, 2, *)$ is in effect, and the 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 that has 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 has been encountered. Note that if there is no slice specified in the data block, a dummy one, components of which are all asterisks, is assumed.

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

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

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

Matrix data record. The matrix data record defines several 2-tuples (doubles) in matrix format and has the following syntactic form:

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

where $r_1, r_2, \ldots, r_m$ are numbers and/or symbols that correspond to rows of the matrix, $c_1, c_2, \ldots, c_n$ are numbers and/or symbols that correspond to columns of the matrix, $a_{11}, a_{12}, \ldots, a_{mn}$ are the matrix elements, which can be the signs `+` and `-`. (In this data record the delimiter `:` that precedes the column list and the delimiter `:=` that follows 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 the 2-tuple (r_i, c_j). If a_{ij} is the plus (`+`) sign, the corresponding 2-tuple (or longer n-tuple, if a slice is used) is included in the elemental set. Otherwise, if a_{ij} is the minus (`-`) sign, the corresponding 2-tuple is not included in the elemental set.

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

Transposed matrix data record. The transposed matrix data record has the following syntactic form:

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

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

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

Being once specified the `(tr)` indicator effects on *all* subsequent data records until either a slice or the end of data block has been encountered.

5.3 Parameter data block

General Form

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

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

`param :` *tabbing-data* ;

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

Where:	*name* is the symbolic name of a parameter.
	value is an optional default value of the parameter.
	record, . . . , *record* are data records.
	tabbing-data specifies parameter data in the tabbing format.
Note:	Commae preceding data records may be omitted.

The data records are:

`:=`	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 parameter 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) whose name is given in the block.

Data blocks can be specified only for the parameters, which are non-computable, i.e. 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, if 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(s).

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

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:

$$[s_1, s_2, \ldots, s_n]$$

where $s_1, s_2, \ldots, s_n$ 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 must be the same as the dimension of the parameter. For instance, if the parameter is a 4-dimensional array, the slice must have four components. The number of asterisks in the slice is called *slice dimension*.

The effect of using slices is the following. If a m-dimensional slice (i.e. a slice that has m asterisks) is specified in the data block, all subsequent data records must 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 the 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 that has no asterisks. Such slice itself defines a complete subscript list, in which case the next data record can define only a single value of the corresponding parameter member.

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

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

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

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

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

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:

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

where $r_1, r_2, \dots, r_m$ are numbers and/or symbols that correspond to rows of the matrix, $c_1, c_2, \dots, c_n$ are numbers and/or symbols that correspond to columns of the table, $a_{11}, a_{12}, \dots, a_{mn}$ are the table elements. Each element can be a number or symbol or the single decimal point. (In this data record the delimiter `:` that precedes the column list and the delimiter `:=` that follows 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 must be 2-dimensional.

Transposed tabular data record. The transposed tabular data record has the following syntactic form:

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

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

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

Being once specified the `(tr)` indicator effects on all subsequent data records until either a slice or the end of data block has been encountered.

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

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

Notes:
1. The keyword `default` may be omitted along with a value that follows it.
2. The symbolic name s of a set may be omitted along with the colon that follows it.
3. All comae 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}`;`

A Using the MathProg translator with GLPK API

The GLPK package includes the API routine `lpx_read_model`, which is a high-level interface to the GNU MathProg translator.

Synopsis

```
#include "glpk.h"
LPX *lpx_read_model(char *model, char *data, char *output);
```

Description The routine `lpx_read_model` reads and translates LP/MIP model (problem) written in the GNU MathProg modeling language.

The character string `model` specifies name of input text file, which contains model section and, optionally, data section. This parameter cannot be `NULL`.

The character string `data` specifies name of input text file, which contains data section. This parameter can be `NULL`. (If the data file is specified and the model file also contains data section, that section is ignored and data section from the data file is used.)

The character string `output` specifies name of output text file, to which the output produced by display statements is written. If the parameter output is `NULL`, the display output is sent to stdout via the routine `print`.

Returns If no errors occurred, the routine returns a pointer to the created problem object. Otherwise the routine sends diagnostics to the standard output and returns `NULL`.

B Solving models with the solver `glpsol`

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

In order to tell the solver that the input file contains a model description, the option `--model` should be specified in the command line. For example:

```
glpsol --model foobar.mod
```

Sometimes it is necessary to use the data section placed in another file, in which case the following command may be used:

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

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

If the model description contains some display statements, by default the display output goes onto the screen (more precisely, onto the standard output device). In order to redirect the display output to a file the following command may be used:

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

If you need to look at the problem which has been generated by the model translator, the option `--wtxt` should be specified in the command line as follows:

```
glpsol --model foobar.mod --wtxt foobar.txt
```

in which case the problem will be written to the file `foobar.txt` in plain text format suitable for visual analysis.

Sometimes it is necessary merely to check the model description not solving the generated problem. In this case the option `--check` should be given in the command line, for example:

```
glpsol --check --model foobar.mod --wtxt foobar.txt
```

In order to write a numeric solution obtained by the solver the following command may be used

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

in which case the solution will be written to the file `foobar.sol` in plain text format.

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

C Example of model description

C.1 Model description written in GNU MathProg

This is a complete example of 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;
```

C.2 Generated LP problem

This is the LP problem that has been generated by the model translator for the example model description.

```
Problem:    TRANSP
Class:      LP
Rows:       6
Columns:    6
Non-zeros:  18

*** OBJECTIVE FUNCTION ***

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]

*** ROWS (CONSTRAINTS) ***

Row 1: cost free
                 0.126 x[San-Diego,Topeka]
                 0.225 x[San-Diego,New-York]
                 0.153 x[Seattle,Chicago]
                 0.225 x[Seattle,New-York]
                 0.162 x[Seattle,Topeka]
                 0.162 x[San-Diego,Chicago]

Row 2: supply[Seattle] <= 350
                     1 x[Seattle,Chicago]
                     1 x[Seattle,New-York]
                     1 x[Seattle,Topeka]

Row 3: supply[San-Diego] <= 600
                     1 x[San-Diego,Chicago]
                     1 x[San-Diego,New-York]
                     1 x[San-Diego,Topeka]
```

```
Row 4: demand[New-York] >= 325
                     1 x[Seattle,New-York]
                     1 x[San-Diego,New-York]

Row 5: demand[Chicago] >= 300
                     1 x[Seattle,Chicago]
                     1 x[San-Diego,Chicago]

Row 6: demand[Topeka] >= 275
                     1 x[Seattle,Topeka]
                     1 x[San-Diego,Topeka]

*** COLUMNS (VARIABLES) ***

Col 1: x[Seattle,New-York] >= 0
                 0.225 (objective)
                 0.225 cost
                     1 supply[Seattle]
                     1 demand[New-York]

Col 2: x[Seattle,Chicago] >= 0
                 0.153 (objective)
                 0.153 cost
                     1 supply[Seattle]
                     1 demand[Chicago]

Col 3: x[Seattle,Topeka] >= 0
                 0.162 (objective)
                 0.162 cost
                     1 supply[Seattle]
                     1 demand[Topeka]

Col 4: x[San-Diego,New-York] >= 0
                 0.225 (objective)
                 0.225 cost
                     1 supply[San-Diego]
                     1 demand[New-York]

Col 5: x[San-Diego,Chicago] >= 0
                 0.162 (objective)
                 0.162 cost
                     1 supply[San-Diego]
                     1 demand[Chicago]

Col 6: x[San-Diego,Topeka] >= 0
                 0.126 (objective)
                 0.126 cost
                     1 supply[San-Diego]
                     1 demand[Topeka]

End of output
```

C.3 Optimal solution of the generated LP problem

This is the optimal solution of the generated LP problem that has been found by the GLPK simplex solver.

```
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]
                    B            300                         350
     3 supply[San-Diego]
                    NU           600                         600         < eps
     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              0             0
     2 x[Seattle,Chicago]
                    B            300             0
     3 x[Seattle,Topeka]
                    NL             0             0                       0.036
     4 x[San-Diego,New-York]
                    B            325             0
     5 x[San-Diego,Chicago]
                    NL             0             0                       0.009
     6 x[San-Diego,Topeka]
                    B            275             0

End of output
```