Linguagem de Modelagem GNU MathProg

Linguagem de Referência

para o GLPK Versão 4.57

(RASCUNHO, Fevereiro 2016)

Título original: Modeling Language GNU MathProg - Language Reference for GLPK Version 4.57

Tradução: João Flávio de Freitas Almeida, Departamento de Engenharia de Produção, Universidade Federal de Minas Gerais, Minas Gerais, Brasil.

Free Software Foundation, Inc., Rua Franklin, 51, 5^o andar, Boston, MA 02110-1301, USA.

Conteúdo

Capítulo 1

Introdução

GNU MathProg é uma linguagem de modelagem projetada para descrever modelos lineares de programação matemática. [1]

A descrição de um modelo escrito na linguagem GNU MathProg consiste em um conjunto de sentenças e blocos de dados construído pelo usuário a partir dos elementos de linguagem descritos neste documento.

Em um processo denominado *tradução*, um programa denominado *tradutor do modelo* analisa a descrição do modelo e o traduz para uma estrutura de dados interna, que pode ser usado tanto para gerar instância de um problema de programação matemática ou obter diretamente a solução numérica do problema por meio de um programa chamado *solver*.

1.1 Problema de programação linear

Em MathProg o problema de programação linear (PL) é expresso da seguinte forma:

minimizar (ou maximizar)

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

subjeito às restrições lineares

$$\begin{array}{l} L_1 \leq a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \leq U_1 \\ L_2 \leq a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \leq U_2 \\ \qquad\qquad . \quad . \quad . \quad . \quad . \quad . \quad . \quad . \quad . \\ L_m \leq a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n \leq U_m \end{array} \tag{1.2}$$

[1] A linguagem GNU MathProg é um subconjunto da linguagem AMPL. A implementação do GLPK é principalmente baseada no artigo: *Robert Fourer*, *David M. Gay*, and *Brian W. Kernighan*, "A Modeling Language for Mathematical Programming." *Management Science* 36 (1990), pp. 519-54.

e os limites das variáveis

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

onde x_1, x_2, ..., x_n são variáveis; z é a função objetivo ; c_1, c_2, ..., c_n são coeficientes da função objetivo; c_0 é o termo constante da função objetivo; a_{11}, a_{12}, ..., a_{mn} são coeficientes das restrições; L_1, L_2, ..., L_m são limites inferiores das restrições; U_1, U_2, ..., U_m são limites superiores das restrições; l_1, l_2, ..., l_n são limites inferiores das variáveis; u_1, u_2, ..., u_n são limites superiores das variáveis.

Os limites das variáveis e das restrições podem ser tanto finitos quanto infinitos. Além disso, os limites inferiores podem ser igual aos limites superiores correspondentes. Logo, os seguintes tipos de variáveis e restrições são permitidos:

$-\infty < x < +\infty$	Variável livre (ilimitada)
$l \leq x < +\infty$	Variável com limite inferior
$-\infty < x \leq u$	Variável com limite superior
$l \leq x \leq u$	Variável duplamente limitada
$l = x = u$	Variável fixa
$-\infty < \sum a_j x_j < +\infty$	Forma linear livre (ilimitada)
$L \leq \sum a_j x_j < +\infty$	Restrição de desigualdade "maior ou igual a"
$-\infty < \sum a_j x_j \leq U$	Restrição de desigualdade "menor ou igual a"
$L \leq \sum a_j x_j \leq U$	Restrição de desigualdade duplamente limitada
$L = \sum a_j x_j = U$	Restrição de igualdade

Além de problemas puramente PL, MathProg também permite problemas de programação inteira mista (PIM), onde algumas ou todas as variáveis são restritas a serem inteiras ou binárias.

1.2 Objetos do modelo

Em MathProg o modelo é descrito em termos de conjuntos, parâmetros, variáveis, restrições e objetivos, que se denominam *objetos do modelo*.

O usuário introduz objetos particulares do modelo usando as sentenças da linguagem. Cada objeto do modelo possui um nome simbólico que o identifica de maneira única sendo projetado para propósitos de referenciação.

Objetos do modelo, incluindo conjuntos, podem ser matrizes multidimensionais construídos sobre conjuntos indexantes. Formalmente, uma matriz n-dimensional A é o mapeamento:

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

onde $\Delta \subseteq S_1 \times \ldots \times S_n$ é um subconjunto do produto Cartesiano de conjuntos indexantes, Ξ é um conjunto dos membros da matriz. Em MathProg o conjunto Δ é chamado o *domínio do subíndice*. Seus membros são n-tuplas $(i_1, \ldots, i_n)$, onde $i_1 \in S_1$, $\ldots$, $i_n \in S_n$.

Se $n = 0$, o produto Cartesiano acima possui exatamente um membro (denominado 0-tupla), portanto, é conveniente pensar nos objetos escalares como sendo matrizes 0-dimensionais que possuem apenas um membro.

O tipo dos membros da matriz é determinado pelo tipo de objeto do modelo correspondente, como segue:

Objeto do modelo	Membro da matriz
Conjunto	Conjunto plano elementar
Parâmetro	Número ou símbolo
Variável	Variável elementar
Restrição	Restrição elementar
Objetivo	Objetivo elementar

Para referir a um membro particular de um objeto, este deve possuir *subíndices*. Por exemplo, se a é um parâmetro 2-dimensional definido sobre $I \times J$, uma referência a seus membros particulares pode ser escrito como $a[i, j]$, onde $i \in I$ e $j \in J$. Entende-se que objetos escalares não necessitam de subíndices por serem 0-dimensionais.

1.3 Estrutura da descrição do modelo

Às vezes é desejável escrever um modelo que, por diferentes motivos, tenha que requerer diferentes dados para resolver cada instância do problema usando o mesmo modelo. Por esta razão, em MathProg a descrição do modelo consiste em duas partes: a *seção de modelo* e a *seção de dados*.

A seção de modelo é a principal parte da descrição do modelo, pois ela contém as declarações dos objetos do modelo. Ela também é comum a todos os problemas baseados no modelo correspondente.

A seção de dados é uma parte opcional da descrição do modelo que contém dados específicos para uma instância particular do problema.

Dependendo do que seja mais conveniente, as seções de modelo e de dados podem ser dispostas em um arquivo único ou em dois arquivos separados. Esta última funcionalidade permite que se tenha um quantidade arbitrária de seções de dados diferentes a serem usadas com a mesma seção de modelo.

Capítulo 2

Codificação da descrição do modelo

A descrição do modelo é codificada em um formato de arquivo plano de texto usando o conjunto de caracteres ASCII. Os caracteres válidos na descrição do modelo são os seguintes:

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

— caracteres numéricos:
`0 1 2 3 4 5 6 7 8 9`

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

— caracteres de espaço em branco:
`SP HT CR NL VT FF`

Dentro de literais de cadeia e comentários quaisquer caracteres ASCII (exceto caracteres de controle) são válidos.

Caracteres de espaço-em-branco não são significativos. Eles podem ser usados livremente entre unidades léxicas para melhorar a legibilidade da descrição do modelo. Eles também são usados para separar unidades léxicas entre si, no caso de não existir outra forma de fazê-lo.

Sintaticamente a descrição do modelo é uma sequência de unidades léxicas nas seguintes categorias:

— nomes simbólicos;

— literais numéricos;

— literais de cadeia;

— palavras-chave;

— delimitadores;

— comentários.

As unidades léxicas da linguagem são discutidas abaixo.

2.1 Nomes simbólicos

Um *nome simbólico* consiste de caracteres alfabéticos e numéricos, em que o primeiro deste deve ser alfabético. Todos os nomes simbólicos devem ser distintos (sensibilidade às maiúsculas: case-sensitive).

Exemplos

```
alfa123
Este_eh_um_nome
_P123_abc_321
```

Nomes simbólicos são usados para identificar objetos do modelo (conjuntos, parâmetros, variáveis, restrições, objetivos) e os índices.

Todos os nomes simbólicos (exceto os nomes dos índices) devem ser únicos, i.e., a descrição do modelo não pode ter objetos com nomes idênticos. Nomes simbólicos de índices devem ser únicos dentro do escopo em que são válidos.

2.2 Literais numéricos

Um *literal numérico* possui a forma *xx*`E`*syy*, onde *xx* é um número com ponto decimal opcional, *s* é o sinal `+` ou `-`, *yy* é um expoente decimal. A letra `E` é insensível à maiúsculas (case-insensitive) e pode ser codificada como `e`.

Exemplos

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

Literais numéricos são usados para representar quantidades numéricas. Eles possuem significado fixo óbvio.

2.3 Literais de cadeia

Uma *literal de cadeia* é uma sequência arbitrária de caracteres cercada por aspas tanto simples como duplas. Ambas formas são equivalentes.

Se uma aspa simples é parte de uma literal de cadeia cercada por aspas simples, ela deve ser codificada duas vezes. De forma análoga, se uma aspa dupla é parte de um literal de cadeia cercada

por aspas duplas, ela deve ser codificada duas vezes.

Exemplos

```
'Esta eh uma string'
"Esta eh outra string"
'Copo d''agua'
"""Beleza"" disse o capitao."
```

Literais de cadeia são usadas para representar quantidades simbólicas.

2.4 Palavras-chave

Uma *palavra-chave* é uma sequência de caracteres alfabéticos e possivelmente alguns caracteres especiais.

Todas as palavras-chave caem em algumas das duas categorias: *palavras-chave reservadas*, que não podem ser usadas como nomes simbólicos, e *palavras-chave não-reservadas*, que são reconhecidas pelo contexto, portanto, podem ser usadas como nomes simbólicos.

As palavras-chave reservadas são as seguintes:

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

Palavras-chave não-reservadas são descritas nas seções posteriores.

Todas as palavras-chave possuem um significado fixo, a ser explicado nas discussões das construções sintáticas correspondentes, onde as palavras-chave são usadas.

2.5 Delimitadores

Um *delimitador* é tanto um caractere especial único quanto uma sequência de dois caracteres especiais, como segue:

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

Se um delimitador consiste de dois caracteres, não deve haver espaços entre os eles.

Todos os delimitadores possuem um significado fixo, a ser explicado nas discussões das construções sintáticas correspondentes, onde os delimitadores são usados.

2.6 Comentários

Com propósitos de documentação, a descrição do modelo pode conter *comentários*, que podem ter duas formas diferentes. A primeira forma é um *comentário de linha-única*, que começa com o caractere `#` e se estende até o final da linha. A segunda forma é uma *sequência de comentários*, que é uma sequência de quaisquer caracteres cercados por `/*` e `*/`.

Exemplos

```
param n := 10; # Este é um comentario
/* Este é outro comentário */
```

Comentários e caracteres de espaço-em-branco são ignorados pelo tradutor do modelo e podem aparecer em qualquer local da descrição do modelo.

Capítulo 3

Expressões

Uma *expressão* é uma regra para calcular um valor. Na descrição de um modelo, expressões são usadas como constituintes de certas sentenças.

No geral, expressões são constituídas de operandos e operadores.

Dependendo do tipo de valor resultante, todas expressões se enquadram nas seguintes categorias:

— expressões numéricas;

— expressões simbólicas;

— expressões indexantes;

— expressões de conjuntos;

— expressões lógicas;

— expressões lineares.

3.1 Expressões numéricas

Uma *expressão numérica* é uma regra para calcular um valor numérico individual representado como um número de ponto-flutuante.

A expressão numérica primária pode ser um literal numérico, um índice, um parâmetro não-indexado, um parâmetro indexado, uma função interna de referência, uma expressão numérica iterada, uma expressão numérica condicional ou outra expressão cercada por parênteses.

Exemplos

`1.23`	(literal numérico)
`j`	(índice)
`time`	(parâmetro não-indexado)
`a['May 2003',j+1]`	(parâmetro indexado)
`abs(b[i,j])`	(função de referência)

```
sum{i in S diff T} alpha[i] * b[i,j]   (expressão iterada)
if i in I then 2 * p else q[i+1]       (expressão condicional)
(b[i,j] + .5 * c)                      (expressão entre parênteses)
```

Expressões numéricas mais genéricas, contendo duas ou mais expressões numéricas primárias, podem ser construídas usando determinados operadores aritméticos.

Exemplos

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

3.1.1 Literais numéricos

Se a expressão numérica primária é um literal numérico, o valor resultante é óbvio.

3.1.2 Índices

Se a expressão numérica primária é um índice, o valor resultante é o valor corrente atribuído àquele índice.

3.1.3 Parâmetros não-indexados

Se a expressão numérica primária é um parâmetro não-indexado (que deve ser 0-dimensional), o valor resultante é o valor do parâmetro.

3.1.4 Parâmetros indexados

A expressão numérica primária, que se refere ao parâmetro indexado, possui a seguinte forma sintática:

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

onde *nome* é o nome simbólico do parâmetro e i_1, i_2, ..., i_n são subíndices.

Cada subíndice deve ser uma expressão numérica ou simbólica. O número de subíndices na lista de subíndices deve ser o mesmo da dimensão do parâmetro com o qual a lista de subíndices está associada.

Os valores reais das expressões de subíndice são usadas para identificar um membro particular do parâmetro que determina o valor resultante da expressão primária.

3.1.5 Funções de referência

Em MathProg existem as seguintes funções internas, que podem ser usadas como expressões numéricas:

`abs(`x`)`	$\lvert x\rvert$, valor absoluto de x
`atan(`x`)`	$\arctan x$, valor principal do arco tangente de x (em radianos)
`atan(`y`, `x`)`	$\arctan y/x$, valor principal do arco tangente de y/x (em radianos). Neste caso, os sinais de ambos argumentos y e x são usados para determinar o quadrante do valor resultante
`card(`X`)`	$\lvert X\rvert$, cardinalidade (o número de elementos) do conjunto X
`ceil(`x`)`	$\lceil x\rceil$, menor inteiro não menor que x ("teto de x")
`cos(`x`)`	$\cos x$, cosseno de x (em radianos)
`exp(`x`)`	e^x, exponencial de x na base-e
`floor(`x`)`	$\lfloor x\rfloor$, maior inteiro não maior que x ("piso de x")
`gmtime()`	o número de segundos decorridos deste 00:00:00 de 01 de Jan de 1970, Tempo Universal Coordenado (detalhes na Seção B.1, página 58)
`length(`s`)`	$\lvert s\rvert$, comprimento da cadeia de caracteres s
`log(`x`)`	$\log x$, logaritmo natural de x
`log10(`x`)`	$\log_{10} x$, logaritmo comum (decimal) de x
`max(`x_1`, `x_2`, ..., `x_n`)`	o maior dos valores x_1, x_2, ..., x_n
`min(`x_1`, `x_2`, ..., `x_n`)`	o menor dos valores x_1, x_2, ..., x_n
`round(`x`)`	arrendondamento de x ao inteiro mais próximo
`round(`x`, `n`)`	arrendondamento de x a n dígitos decimais
`sin(`x`)`	$\sin x$, seno de x (em radianos)
`sqrt(`x`)`	$\sqrt{x}$, raiz quadrada não-negativa de x
`str2time(`s`, `f`)`	conversão de uma cadeia de caracteres s ao tempo calendário (detalhes na Seção B.2, página 58)
`trunc(`x`)`	truncado de x ao inteiro mais próximo
`trunc(`x`, `n`)`	truncado de x a n dígitos decimais
`Irand224()`	gera inteiro pseudo-aleatório uniformemente distribuído em $[0, 2^{24})$
`Uniform01()`	gera número pseudo-aleatório uniformemente distribuído em $[0, 1)$
`Uniform(`a`, `b`)`	gera número pseudo-aleatório uniformemente distribuído em $[a, b)$
`Normal01()`	gera variável Gaussiana pseudo-aleatória com $\mu = 0$ e $\sigma = 1$
`Normal(`μ`, `σ`)`	gera variável Gaussiana pseudo-aleatória com μ e σ dados

Os argumentos de todas as funções internas, exceto `card`, `length`, e `str2time`, devem ser expressões numéricas. O argumento de `card` deve ser uma expressão de conjunto. O argumento de `length` e ambos argumentos de `str2time` devem ser expressões simbólicas.

O valor resultante da expressão numérica, que é uma função de referência, é o resultado da aplicação da função ao(s) seu(s) argumento(s).

Note que cada função geradora de números pseudo-aleatórios possui um argumento latente (i.e., algum estado interno), que é alterado sempre que função é aplicada. Assim, se a função é aplicada repetidamente mesmos aos argumentos idênticos, devido ao efeito secundário, sempre se produzirão valores resultantes diferentes.

3.1.6 Expressões iteradas

Uma *expressão numérica iterada* é uma expressão numérica primária, que possui a seguinte forma sintática:

operador-iterado expressão-indexada integrando

onde o *operador-iterado* é o nome simbólico do operador de iteração a ser executado (veja abaixo), *expressão-indexada* é uma expressão indexada que introduz índices e controla a iteração e *integrando* é uma expressão numérica que participa da operação.

Em MathProg existem quatro operadores iterados, que podem ser usados em expressões numéricas:

`sum`	somatório	$\sum_{(i_1,\ldots,i_n)\in\Delta} f(i_1,\ldots,i_n)$
`prod`	produtório	$\prod_{(i_1,\ldots,i_n)\in\Delta} f(i_1,\ldots,i_n)$
`min`	mínimo	$\min_{(i_1,\ldots,i_n)\in\Delta} f(i_1,\ldots,i_n)$
`max`	máximo	$\max_{(i_1,\ldots,i_n)\in\Delta} f(i_1,\ldots,i_n)$

onde $i_1, \ldots, i_n$ são índices introduzidos nas expressões indexadas, Δ é o domínio, um conjunto de n-tuplas especificado pela expressão indexada que define valores particulares atribuído aos índices ao executar a operação iterada e $f(i_1, \ldots, i_n)$ é o integrando, uma expressão numérica cujo valor resultante depende dos índices.

O valor resultante de uma expressão numérica iterada é o resultado da aplicação do operador iterado ao seu integrando por todas as n-tuplas contidas no domínio.

3.1.7 Expressões condicionais

Uma *expressão numérica condicional* é uma expressão numérica primária que possui uma das seguintes formas sintáticas:

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

`if` b `then` x

onde b é uma expressão lógica, enquanto que x e y são expressões numéricas.

O valor resultante da expressão condicional depende do valor da expressão lógica que segue a palavra-chave `if`. Se ela recebe o valor *verdadeiro*, o valor da expressão condicional é o valor da expressão que segue a palavra-chave `then`. Caso contrário, se a expressão lógica recebe o valor *falso*, o valor da expressão condicional é o valor da expressão que segue a palavra-chave *else*. Se ocorre a segunda forma sintática da expressão condicional, a reduzida, e a expressão lógica recebe o valor *falso*, então o valor resultante da expressão condicional é zero.

3.1.8 Expressões parentizadas

Qualquer expressão numérica pode ser cercada por parênteses, o que as torna sintaticamente uma expressão numérica primária.

Parênteses podem ser usados em expressões numéricas, como em álgebra, para especificar a ordem desejada na qual as operações devem ser realizadas. Quando se usam parênteses, a expressão entre parênteses é avaliada antes e seu o valor resultante é usado.

O valor resultante de uma expressão parentizada é o mesmo valor de uma expressão cercada entre parênteses.

3.1.9 Operadores aritméticos

Em MathProg exitem os seguintes operadores aritméticos, que podem ser usados em expressões numéricas:

`+` x	mais unário
`-` x	menos unário
x `+` y	adição
x `-` y	subtração
x `less` y	diferença positiva (se $x < y$ então 0 senão $x - y$)
x `*` y	multiplicação
x `/` y	divisão
x `div` y	quociente da divisão exata
x `mod` y	resto da divisão exata
x `**` y, x `^` y	exponenciação (elevação a uma potência)

onde x e y são expressões numéricas.

Se a expressão inclui mais de um operador aritmético, todos operadores são executados da esquerda para a direita de acordo com a hierarquia das operações (veja abaixo) com a única exceção de que os operadores de exponenciação são executados da direita para a esquerda.

O valor resultante da expressão, que contém operadores aritméticos, é o resultado da aplicação dos operadores aos seus operandos.

3.1.10 Hierarquia das operações

A lista seguinte apresenta a hierarquia das operações em expressões numéricas:

Operação	Hierarquia
Avaliação das funções (`abs`, `ceil`, etc.)	1ª
Exponenciação (`**`, `^`)	2ª
Mais e menos unário (`+`, `-`)	3ª
Multiplicação e divisão (`*`, `/`, `div`, `mod`)	4ª
Operações iteradas (`sum`, `prod`, `min`, `max`)	5ª
Adição e subtração (`+`, `-`, `less`)	6ª
Avaliação condicional (`if` ... `then` ... `else`)	7ª

Esta hierarquia é usada para determinar qual de duas operações consecutivas deve ser executada primeiro. Se o primeiro operador possui hierarquia maior ou igual ao segundo, a primeira operação é executada. Caso contrário, a segunda operação é comparada à terceira e assim sucessivamente. Quando se alcança o fim da expressão, todas as operações remanescentes são executadas na ordem inversa.

3.2 Expressões simbólicas

Uma *expressão simbólica* é uma regra para calcular um valor simbólico individual representado como uma cadeia de caracteres.

A expressão simbólica primária pode ser uma cadeia literal, um índice, um parâmetro não-indexado, um parâmetro indexado, uma função interna de referência, uma expressão simbólica condicional ou outra expressão simbólica cercada entre parênteses.

Também é permitido usar uma expressão numérica como a expressão simbólica primária, neste caso o valor resultante da expressão numérica é automaticamente convertido em um tipo simbólico.

Exemplos

`'Maio de 2003'`	(literal de cadeia)
`j`	(índice)
`p`	(parâmetro não-indexado)
`s['abc',j+1]`	(parâmetro indexado)
`substr(name[i],k+1,3)`	(função de referência)
`if i in I then s[i,j] & "..." else t[i+1]`	(expressão condicional)
`((10 * b[i,j]) & '.bis')`	(expressão parentizada)

Expressões simbólicas mais genéricas contendo duas ou mais expressões simbólicas primárias podem ser construídas usando o operador de concatenação.

Exemplos

```
'abc[' & i & ',' & j & ']'
"de " & cidade[i] " para " & cidade[j]
```

Os princípios da avaliação de expressões simbólicas são completamente análogos àqueles dados para expressões numéricas (veja acima).

3.2.1 Funções de referência

Em MathProg existem as seguintes funções internas que podem ser usadas em expressões simbólicas:

`substr(`s`,` x`)`	substring de s iniciado na posição x
`substr(`s`,` x`,` y`)`	substring de s iniciado na posição x com tamanho y
`time2str(`t`,` f`)`	converte o tempo de calendário para uma cadeia de caracteres (detalhes na Seção B.3, página 60)

O primeiro argumento de `substr` deve ser uma expressão simbólica enquanto que o segundo e o

terceiro (opcional) argumentos devem ser expressões numéricas.

O primeiro argumento de `time2str` deve ser uma expressão numérica, e seu segundo argumento deve ser uma expressão simbólica.

O valor resultante da expressão simbólica, que é uma função de referência, é o resultado de aplicar a função aos seus argumentos.

3.2.2 Operadores simbólicos

Atualmente, em MathProg exite um único operador simbólico:

`s & t`

onde s e t são expressões simbólicas. Este operador implica na concatenação dos seus dois operandos simbólicos, que são cadeias de caracteres.

3.2.3 Hierarquia de operações

A lista seguinte mostra a hierarquia das operações em expressões simbólicas:

Operação	Hierarquia
Avaliação de operações numéricas	1ª-7ª
Concatenação (`&`)	8ª
Avaliação condicional (`if` ... `then` ... `else`)	9ª

Esta hierarquia possui o mesmo significado como explicado acima para expressões numéricas (ver Subseção 3.1.10, página 17).

3.3 Expressões de indexação e índices

Uma *expressão indexante* é uma construção auxiliar que especifica um conjunto plano de n-tuplas e introduz índices. Possui duas formas sintáticas:

`{` *entrada*$_1$`,` *entrada*$_2$`, ...,` *entrada*$_m$ `}`

`{` *entrada*$_1$`,` *entrada*$_2$`, ...,` *entrada*$_m$ `:` *predicado* `}`

onde *entrada*$_1$, *entrada*$_2$, ..., *entrada*$_m$ são entradas indexantes, *predicado* é uma expressão lógica que especifica um predicado opcional (condição lógica).

Cada *entrada indexante* na expressão indexante possui uma das três formas seguintes:

i `in` S

`(`i_1`,` i_2`, ...,`i_n`)` `in` S

S

onde i_1, i_2, ..., i_n são índices e S é uma expressão de conjunto (discutido na próxima seção) que especifica o conjunto básico.

O número de índices na entrada indexante deve coincidir com a dimensão do conjunto básico S, i.e., se S consiste de 1-tuplas, deve-se usar a primeira forma. Se S consiste de n-tuplas, onde $n > 1$, a segunda forma deve ser usada.

Se a primeira forma da entrada indexante é usada, o índice i pode ser apenas um índice (veja mais adiante). Se a segunda forma é utilizada, os índices i_1, i_2, ..., i_n podem ser tanto índices como alguma expressão numérica ou simbólica, em que pelo menos um dos índices deve ser um índice. Na terceira, a forma reduzida da entrada indexante possui o mesmo efeito se houver i (se S é 1-dimensional) ou i_1, i_2, ..., i_n (se S é n-dimensional) onde todos são especificados como índices.

Um *índice* é um objeto auxiliar do modelo, que atua como uma variável individual. Os valores atribuídos aos índices são componentes das n-tuplas dos conjuntos básicos, i.e., algumas quantidades numéricas e simbólicas.

Com propósitos de referência, índices podem ter nomes simbólicos. No entanto, diferentemente de outros objetos de modelo (conjuntos, parâmetros, etc.) índices não precisam ser explicitamente declarados. Cada nome simbólico *não-declarado* usado na posição indexante de uma entrada indexante é reconhecida como o nome simbólico do índice correspondente.

Os nomes simbólicos dos índices são válidos somente dentro do escopo da expressão indexante, onde o índice foi inserido. Além do escopo, os índices são completamente inacessíveis, de modo que os mesmos nomes simbólicos podem ser usados para outros propósitos, em particular, para representar índices em outras expressões indexantes.

O escopo da expressão indexante, em que declarações implícitas de índices são válidas, depende do contexto em que a expressão indexante é usada:

— Se a expressão indexante é usada em um operador-iterado, seu escopo se estende até o final do integrando.

— Se a expressão indexante é usada como uma expressão de conjunto primária, seu escopo de estende até o final desta expressão indexante.

— Se a expressão indexante é usada para definir o domínio do subíndice na declaração de alguns objetos de modelo, seu escopo se estende até o final da declaração correspondente.

O mecanismo de indexação implementado para indexar expressões é melhor explicado por alguns exemplos discutidos abaixo.

Sejam três conjuntos:

$$A = \{4, 7, 9\},$$
$$B = \{(1, Jan), (1, Fev), (2, Mar), (2, Abr), (3, Mai), (3, Jun)\},$$
$$C = \{a, b, c\},$$

onde A e C consistem de 1-tuplas (singletos), B consiste de 2-tuplas (dobletes). Considere a seguinte expressão indexante:

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

onde `i`, `j`, `k`, e `l` são índices.

Embora MathProg não seja uma linguagem procedural, para qualquer expressão indexante uma descrição algorítmica equivalente pode ser dada. Em particular, a descrição algorítmica da expressão indexante acima poderia ser vista como segue:

para todo $i \in A$ **faça**
  para todo $(j, k) \in B$ **faça**
    para todo $l \in C$ **faça**
      ação;

onde os índices i, j, k, l são consecutivamente atribuídos aos componentes correspondentes das n-tuplas dos conjuntos básicos A, B, C, e *ação* é alguma ação que dependa do contexto no qual a expressão indexante é usada. Por exemplo, se a ação fosse imprimir os valores atuais dos índices, a impressão seria vista como segue:

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

Seja o exemplo da expressão indexante usado na seguinte operação iterada:

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

onde `p` é uma parâmetro numérico 4-dimensional ou alguma expressão numérica cujo valor resultante dependa de `i`, `j`, `k` e `l`. Neste caso, a ação é o somatório, de forma que o valor resultante da expressão numérica primária é:

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

Agora seja a expressão indexante do exemplo usada como uma expressão de conjunto primária. Neste caso, a ação é reunir todas as 4-tuplas (quádruplas) da forma (i, j, k, l) em um conjunto, de forma que o valor resultante de tal operação é simplesmente o produto Cartesiano dos conjuntos básicos:

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

Note que neste caso, a mesma expressão indexante pode ser escrita na forma reduzida:

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

pois os índices i, j, k e l não são referenciados, portanto, seus nomes simbólicos não precisam ser especificados.

Finalmente, seja a expressão indexante do exemplo usada como o domínio do subíndice na declaração de um objeto de modelo 4-dimensional, por exemplo, um parâmetro numérico:

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

Neste caso, a ação é gerar os membros do parâmetro, onde cada membro possui a forma $p[i, j, k, l]$.

Como mencionado anteriormente, alguns índices da segunda forma das entradas indexantes podem ser expressões numéricas ou simbólicas, não apenas índices. Neste caso, os valores resultantes destas expressões desempenham o papel de algumas condições lógicas para selecionar apenas aquelas n-tuplas do produto Cartesiano dos conjuntos básicos que satisfaçam estas condições.

Considere, por exemplo, a seguinte expressão indexante:

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

onde `i`, `k` e `l` são índices, e `i-1` é uma expressão numérica. A descrição algorítmica desta expressão indexante é a seguinte:

para todo $i \in A$ **faça**
  para todo $(j, k) \in B$ **e** $j = i - 1$ **faça**
    para todo $l \in C$ **faça**
      ação;

Assim, se esta expressão indexante fosse usada como uma expressão de conjunto primária, o conjunto resultante seira o seguinte:

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

Deve-se notar que neste caso o conjunto resultante consiste em 3-tuplas, e não de 4-tuplas, porque na expressão indexante não há índice que corresponda ao primeiro componente das 2-tuplas do conjunto B.

A regra geral é: o número de componentes de n-tuplas definido por uma expressão indexante é o mesmo do número de índices naquela expressão, onde a correspondência entre índices e componentes nas n-tuplas no conjunto resultante é posicional, i.e., o primeiro índice corresponde ao primeiro componente, o segundo índice corresponde ao segundo componente, etc.

Em alguns casos é necessário selecionar um subconjunto do produto Cartesiano de alguns conjuntos. Isto pode ser alcançado mediante o emprego de um predicado lógico opcional, que é especificado na expressão indexante.

Considere, por exemplo, a seguinte expressão indexante:

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

onde a expressão lógica após os dois pontos é um predicado. A descrição algorítmica desta expressão indexante é a seguinte:

para todo $i \in A$ **faça**
  para todo $(j, k) \in B$ **faça**
    para todo $l \in C$ **faça**
      se $i \leq 5$ **e** $k \neq$ 'Mar' **então**
        ação;

Assim, se a expressão indexante fosse usada como uma expressão de conjunto primária, o conjunto resultante seria o seguinte:

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

Se o predicado não é especificado na expressão indexante assume-se um, que recebe o valor *verdadeiro*.

3.4 Expressões de conjunto

Uma *expressão de conjunto* é uma regra para calcular um conjunto elementar, i.e., uma coleção de n-tuplas, onde os componentes das n-tuplas são quantidades numéricas e simbólicas.

A expressão de conjunto primária pode ser um conjunto de literais, um conjunto não-indexado, um conjunto indexado, um conjunto "aritmético", uma expressão indexante, uma expressão de conjunto iterada, uma expressão de conjunto condicional ou outra expressão cercada por parênteses.

Exemplos

`{(123,'aaa'), (i+1,'bbb'), (j-1,'ccc')}`	(conjunto de literais)
`I`	(conjunto não-indexado)
`S[i-1,j+1]`	(conjunto indexado)
`1..t-1 by 2`	(conjunto "aritmético")
`{t in 1..T, (t+1,j) in S: (t,j) in F}`	(expressão indexante)
`setof{i in I, j in J}(i+1,j-1)`	(expressão de conjunto iterado)
`if i < j then S[i,j] else F diff S[i,j]`	(expressão de conjunto condicional)
`(1..10 union 21..30)`	(expressão de conjunto parentizado)

Expressões de conjuntos mais genéricas contendo duas ou mais expressões de conjunto primárias podem ser construídas usando operadores específicos de conjunto .

Exemplos

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

3.4.1 Conjuntos de literais

Um *conjunto de literais* é uma expressão de conjunto primária que possui as duas formas sintáticas seguintes:

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

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

onde $e_1, \ldots, e_m, e_{11}, \ldots, e_{mn}$ são expressões numéricas ou simbólicas.

Se a primeira forma é adotada, o conjunto resultante consiste de 1-tuplas (singletos) enumerados entre as chaves. É permitido especificar um conjunto vazio como `{ }`, que não possui 1-tuplas.

Se a segunda forma é adotada, o conjunto resultante consiste de n-tuplas enumeradas entre as chaves, onde uma n-tupla particular consiste nos componentes correspondentes enumerados entre parênteses. Todas as n-tuplas devem ter o mesmo número de componentes.

3.4.2 Conjuntos não-indexados

Se a expressão de conjunto primária é um conjunto não-indexado (que deve ser 0-dimensional), o conjunto resultante é um conjunto elementar associado com o objeto conjunto correspondente.

3.4.3 Conjuntos indexados

A expressão de conjunto primária, que se refere a um conjunto indexado, tem a seguinte forma sintática:

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

onde *nome* é o nome simbólico do objeto conjunto e i_1, i_2, ..., i_n são subíndices.

Cada subíndice deve ser uma expressão numérica ou simbólica. O número de subíndices nas lista de subíndices deve ser o mesmo da dimensão do objeto conjunto com o qual as lista de subíndice está associada.

Os valores correntes das expressões de subíndices são usados para identificar um membro particular do objeto conjunto que determina o conjunto resultante.

3.4.4 Conjuntos "aritméticos"

A expressão de conjunto primária que constitui um conjunto "aritmético", possui as duas formas sintáticas seguintes:

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

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

onde t_0, t_1, e δt são expressões numéricas (o valor de δt não deve ser zero). A segunda forma é equivalente a primeira forma, onde $\delta t = 1$.

Se $\delta t > 0$, o conjunto resultante é determinado como segue:

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

Caso contrário, se $\delta t < 0$, o conjunto resultante é determinado como segue:

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

3.4.5 Expressões indexantes

Se a expressão primária é uma expressão indexante, o conjunto resultante é determinado como descrito anteriormente, na Seção 3.3, página 19.

3.4.6 Expressões iteradas

Uma *expressão de conjunto iterada* é uma expressão de conjunto primária, que possui a seguinte forma sintática:

`setof` *expressão-indexante integrando*

onde *expressão-indexante* é uma expressão indexante, que introduz índices e controla a iteração, *integrando* é tanto uma expressão numérica ou simbólica individual, como uma lista de expressões numéricas ou simbólicas separadas por vírgula e cercadas entre parênteses.

Se o integrando é uma expressão numérica ou simbólica individual, o conjunto resultante consiste de 1-tuplas, sendo determinado como segue:

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

onde x é um valor do integrando, i_1, ..., i_n são índices introduzidos na expressão indexante, Δ é o domínio, um conjunto de n-tuplas especificado pela expressão indexante que define valores particulares atribuídos aos índices ao realizar a operação de iteração.

Se o integrando é uma lista contendo m expressões numéricas e simbólicas, o conjunto resultante consiste de m-tuplas, sendo determinado como segue:

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

onde x_1, ..., x_m são valores das expressões na lista de integrandos, i_1, ..., i_n e Δ possuem o mesmo significado anterior.

3.4.7 Expressões condicionais

Uma *expressão de conjunto condicional* é uma expressão de conjunto primária que possui a seguinte forma sintática:

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

onde b é uma expressão lógica, X e Y são expressões de conjunto, que devem definir conjuntos da mesma dimensão.

O valor resultante da expressão condicional depende do valor da expressão lógica que segue a palavra-chave `if`. Se ela recebe o valor *verdadeiro*, o conjunto resultante é o valor da expressão que segue a palavra-chave `then`. Caso contrário, se a expressão lógica recebe o valor *falso*, o conjunto resultante é o valor da expressão que segue a palavra-chave `else`.

3.4.8 Expressões parentizadas

Qualquer expressão de conjunto pode ser cercada entre parênteses, o que as tornam sintaticamente uma expressão de conjunto primária. Parênteses podem ser usados em expressões de conjunto, como em álgebra, para especificar a ordem desejada nas quais as operações devem ser executadas. Quando se usam parênteses, a expressão entre parênteses é avaliada antes que o valor resultante seja usado. O valor resultante de uma expressão parentizada é idêntico ao valor da expressão cercada entre parênteses.

3.4.9 Operadores de conjunto

Em MathProg existem os seguintes operadores de conjunto, que podem ser usados em expressões de conjunto:

X `union` Y	união $X \cup Y$
X `diff` Y	diferença $X \backslash Y$
X `symdiff` Y	diferença simétrica $X \oplus Y = (X \backslash Y) \cup (Y \backslash X)$
X `inter` Y	interseção $X \cap Y$
X `cross` Y	produto Cartesiano (cruzado) $X \times Y$

onde X e Y são expressões de conjunto, que devem definir conjuntos de dimensões idênticas (exceto o produto Cartesiano).

Se a expressão inclui mais de um operador de conjunto, todos operadores são executados da esquerda para a direita de acordo com a hierarquia das operações (veja adiante).

O valor resultante da expressão, que contém operadores de conjunto, é o resultado da aplicação dos operadores aos seus operandos.

A dimensão do conjunto resultante, i.e., a dimensão das n-tuplas, dos quais consistem o conjunto resultante, é a mesma da dimensão dos operandos, exceto o produto Cartesiano, onde a dimensão do conjunto resultante é a soma das dimensões dos seus operandos.

3.4.10 Hierarquia das operações

A lista seguinte mostra a hierarquia das operações em expressões de conjunto:

Operação	Hierarquia
Avaliação de operações numéricas	1ª-7ª
Avaliação de operações simbólicas	8ª-9ª
Avaliação de conjuntos iterados ou "aritméticos" (`setof`, `..`)	10ª
Produto Cartesiano (`cross`)	11ª
Interseção (`inter`)	12ª
União e diferença (`union`, `diff`, `symdiff`)	13ª
Avaliação condicional (`if` ... `then` ... `else`)	14ª

Esta hierarquia possui o mesmo significado como explicado anteriormente para expressões numéricas (ver Subseção 3.1.10, página 17).

3.5 Expressões lógicas

Uma *expressão lógica* é uma regra para calcular um valor lógico individual, que pode ser *verdadeiro* ou *falso*.

A expressão lógica primária pode ser uma expressão numérica, uma expressão relacional, uma expressão lógica iterada ou outra expressão lógica cercada entre parênteses.

Exemplos

`i+1`	(expressão numérica)
`a[i,j] < 1.5`	(expressão relacional)
`s[i+1,j-1] <> 'Mar' & year`	(expressão relacional)
`(i+1,'Jan') not in I cross J`	(expressão relacional)
`S union T within A[i] inter B[j]`	(expressão relacional)
`forall{i in I, j in J} a[i,j] < .5 * b[i]`	(expressão lógica iterada)
`(a[i,j] < 1.5 or b[i] >= a[i,j])`	(expressão lógica parentizada)

Expressões lógicas mais genéricas, contendo duas ou mais expressões lógicas primárias, podem ser construídas usando determinados operadores lógicos.

Exemplos

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

3.5.1 Expressões numéricas

O valor resultante da expressão lógica primária, que é uma expressão numérica, é *verdadeiro*, se o valor resultante da expressão numérica é diferente de zero. Caso contrário o valor resultante da expressão lógica é *falso*.

3.5.2 Operadores relacionais

Em MathProg existem os seguintes operadores relacionais, que podem ser usados em expressões lógicas:

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

onde x, x_1, ..., x_n, y são expressões numéricas ou simbólicas, X e Y são expressões de conjunto.

1. Nas operações `in`, `not in` e `!in` o número de componentes nos primeiros operandos deve ser igual a dimensão do segundo operando.

2. Nas operações `within`, `not within` e `!within` ambos operandos devem ter a mesma dimensão.

Todos operadores relacionais listados acima têm seus significados matemáticos convencionais. O valor resultante é *verdadeiro*, se a relação correspondente é satisfeita para seus operandos, caso contrário é *falso*. (Note que valores simbólicos são ordenados de forma lexicográfica e qualquer valor numérico precede qualquer valor simbólico.)

3.5.3 Expressões iteradas

Uma *expressão lógica iterada* é uma expressão lógica primária com a seguinte forma sintática:

operador-iterado expressão-indexante integrando

onde *operador-iterado* é o nome simbólico do operador iterado a ser executado (veja adiante), *expressão-indexante* é uma expressão indexante que introduz índices e controla a iteração, *integrando* é uma expressão numérica que participa da operação.

Em MathProg existem dois operadores iterados que podem ser usados em expressões lógicas:

`forall`	quantificador-$\forall$	$\forall(i_1, \ldots, i_n) \in \Delta[f(i_1, \ldots, i_n)]$,
`exists`	quantificador-$\exists$	$\exists(i_1, \ldots, i_n) \in \Delta[f(i_1, \ldots, i_n)]$,

onde i_1, $\ldots$, i_n são índices introduzidos na expressão indexante, Δ é o domínio, um conjunto de n-tuplas especificado pela expressão indexante que define valores específicos atribuídos aos índices ao executar a operação iterada, e $f(i_1, \ldots, i_n)$ é o integrando, uma expressão lógica cujo valor resultante depende dos índices.

Para o quantificador $\forall$, o valor resultante da expressão lógica iterada é *verdadeiro*, se o valor do integrando é *verdadeiro* para todas as n-tuplas contidas no domínio, caso contrário, é *falso*.

Para o quantificador $\exists$ o valor resultante da expressão lógica iterada é *falso*, se o valor do integrando é *falso* para todas as n-tuplas contidas no domínio, caso contrário, é *verdadeiro*.

3.5.4 Expressões parentizadas

Qualquer expressão lógica pode ser cercada entre parênteses, o que a converte sintaticamente em uma expressão lógica primária.

Parênteses podem ser usados em expressões lógicas, como em álgebra, para especificar a ordem desejada na qual as operações devem ser executadas. Quando se usam parênteses, a expressão entre parênteses é avaliada antes que o valor resultante seja usado.

O valor resultante da expressão parentizada é idêntico ao valor da expressão cercada entre parênteses.

3.5.5 Operadores lógicos

Em MathProg existem os seguintes operadores lógicos, que podem ser usados em expressões lógicas:

`not` x, `!`x	negação $\neg\, x$
x `and` y, x `&&` y	conjunção ("e" lógico) $x \,\&\, y$
x `or` y, x `||` y	disjunção ("ou" lógico) $x \vee y$

onde x e y são expressões lógicas.

Se a expressão inclui mais de um operador lógico, todos operadores são executados da esquerda para a direita de acordo com a hierarquia das operações (veja adiante). O valor resultante da expressão que contém operadores lógicos é o resultado da aplicação dos operadores aos seus operandos.

3.5.6 Hierarquia das operações

A lista seguinte mostra a hierarquia das operações em expressões lógicas:

Operation	Hierarchy
Avaliação de operações numéricas	1ª-7ª
Avaliação de operações simbólicas	8ª-9ª
Avaliação de operações de conjunto	10ª-14ª
Operações relacionais (`<`, `<=`, etc.)	15ª
negação (`not`, `!`)	16ª
Conjunção(`and`, `&&`)	17ª
Quantificação-∀ e -∃ (`forall`, `exists`)	18ª
Disjunção (`or`, `||`)	19ª

Esta hierarquia possui o mesmo significado como explicado anteriormente para expressões numéricas (ver Subseção 3.1.10, página 17).

3.6 Expressões lineares

Uma *expressão linear* é uma regra para calcular a chamada *forma linear* ou simplesmente *fórmula*, que é uma função linear (ou afim) de variáveis elementares.

A expressão linear primária pode ser uma variável não-indexada, uma variável indexada, uma expressão linear iterada, uma expressão linear condicional ou outra expressão linear cercada entre parênteses.

Também e permitido usar uma expressão numérica como a expressão linear primária, neste caso, o valor resultante da expressão numérica é automaticamente convertido para uma fórmula que inclui o termo constante apenas.

Exemplos

`z`	(variável não-indexada)
`x[i,j]`	(variável indexada)
`sum{j in J} (a[i,j] * x[i,j] + 3 * y[i-1])`	(expressão linear iterada)
`if i in I then x[i,j] else 1.5 * z + 3.25`	(expressão linear condicional)
`(a[i,j] * x[i,j] + y[i-1] + .1)`	(expressão linear parentizada)

Expressões lineares mais genéricas, contendo duas ou mais expressões lineares primárias, podem ser construídas usando determinados operadores aritméticos.

Exemplos

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

3.6.1 Variáveis não-indexadas

Se a expressão linear primária é uma variável não-indexada (que deve se 0-dimensional), a fórmula resultante formula é aquela variável não-indexada.

3.6.2 Variáveis indexadas

A expressão linear primária que se refere a uma variável indexada possui a seguinte forma sintática:

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

onde *nome* é o nome simbólico da variável do modelo, i_1, i_2, ..., i_n são subíndices.

Cada subíndice deve ser uma expressão numérica ou simbólica. O número de subíndices na lista de subíndices deve ser igual ao da dimensão da variável do modelo com a qual está associada a lista de subíndices.

Os valores correntes das expressões dos subíndices são usados para identificar um membro particular da variável do modelo que determina a fórmula resultante, que é uma variável elementar associada com o membro correspondente.

3.6.3 Expressões iteradas

Uma *expressão linear iterada* é uma expressão linear primária, que tem a seguinte forma sintática:

`sum` *expressão-indexante integrando*

onde *expressão-indexante* é uma expressão indexante, que introduz índices e controla iterações, *integrando* é uma expressão linear que participa da operação.

A expressão linear iterada é avaliada exatamente da mesma forma que a expressão numérica iterada (ver Subseção 3.1.6, página 16), exceto que o integrando participante do somatório é uma fórmula e não um valor numérico.

3.6.4 Expressões condicionais

Uma *expressão linear condicional* é uma expressão linear primária, que possui uma das duas formas sintáticas seguintes:

`if` b `then` f `else` g

`if` b `then` f

onde b é uma expressão lógica, f e g são expressões lineares.

A expressão linear condicional é avaliada exatamente da mesma forma que a expressão condicional numérica (ver Subseção 3.1.7, página 16), exceto que os operandos que participam da operação são fórmulas e não valores numéricos.

3.6.5 Expressões parentizadas

Qualquer expressão linear pode ser cercada entre parênteses, o que a converte sintaticamente em uma expressão linear primária.

Parênteses podem ser usados em expressões lineares, como em álgebra, para especificar a ordem desejada na qual as operações devem ser executadas. Quando se usam parênteses, a expressão entre parênteses é avaliada antes que a fórmula resultante seja usada.

O valor resultante da expressão parentizada é idêntico ao valor da expressão cercada entre parênteses.

3.6.6 Operadores aritméticos

Em MathProg existem os seguintes operadores aritméticos, que podem ser usados em expressões lineares:

`+` f	mais unário
`-` f	menos unário
f `+` g	adição
f `-` g	subtração
x `*` f, f `*` x	multiplicação
f `/` x	divisão

onde f e g são expressões lineares, x é uma expressão numérica (mais precisamente, uma expressão linear contendo apenas o termo constante).

Se a expressão inclui mais de um operador aritmético, todos operadores são executados da esquerda para a direita de acordo com a hierarquia das operações (veja adiante). O valor resultante da expressão, que contém operadores aritméticos, é o resultado de aplicar os operadores aos seus operandos.

3.6.7 Hierarquia das operações

A hierarquia de operações aritméticas usada em expressões lineares é a mesma para expressões numéricas (ver Subseção 3.1.10, página 17).

Capítulo 4

Sentenças

Sentenças são unidades básicas da descrição do modelo. Em MathProg todas as sentenças são divididas em duas categorias: sentenças de declaração e sentenças funcionais.

Sentenças de declaração (sentença *set*, sentença *parameter*, sentença *variable*, sentença *constraint*, sentença *objective*) são usados para declarar objetos de certo tipo do modelo e definir certas propriedades de tais objetos.

Sentenças funcionais (sentença *solve*, sentença *check*, sentença *display*, sentença *printf*, sentença *loop*, sentença *table*) são projetadas para executar ações específicas.

Note que sentenças de declaração podem seguir em qualquer ordem arbitrária, o que não afeta o resultado da tradução. Entretanto, qualquer objeto de modelo deve ser declarado antes de ser referenciado por outras sentenças.

4.1 Sentença set

`set` *nome alias domínio* `,` *atributo* `,` ... `,` *atributo* `;`

nome é um nome simbólico do conjunto;

alias é um literal de cadeia opcional que especifica um pseudônimo para o conjunto;

domínio é uma expressão indexante opcional que especifica o domínio do subíndice do conjunto;

atributo, ..., *atributo* são atributos opcionais do conjunto (as vírgulas que precedem os atributos podem ser omitidas.)

Atributos opcionais

`dimen` *n*
: especifica a dimensão de *n*-tuplas das quais o conjunto é consistido;

`within` *expressão*
: especifica um superconjunto que restringe ao conjunto ou a todos seus membros (conjuntos elementares) a estarem incluídos naquele superconjunto;

`:=` *expressão*
: especifica um conjunto elementar atribuído ao conjunto ou aos seus membros;

`default` *expressão*
: especifica um conjunto elementar atribuído ao conjunto ou aos seus membros sempre que não há dados apropriados disponíveis na seção de dados.

Exemplos

```
set nos;
set arcos within nos cross nos;
set passo{s in 1..maxiter} dimen 2 := if s = 1 then arcos else passo[s-1]
   union setof{k in nos, (i,k) in passo[s-1], (k,j) in passo[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')};
```

A sentença set declara um conjunto. Se o domínio do subíndice não é especificado, o conjunto é um conjunto simples, caso contrário será uma matriz de conjuntos elementares.

O atributo `dimen` especifica a dimensão de n-tuplas da qual é consistida o conjunto (se o conjunto é simples) ou seus membros (se o conjunto é uma matriz de conjuntos elementares), em que n deve ser um inteiro de 1 a 20. Pode-se especificar no máximo um atributo `dimen`. Se o atributo `dimen` não é especificado, a dimensão das n-tuplas é implicitamente determinada por outros atributos (por exemplo, se há uma expressão que segue `:=` ou a palavra-chave `default`, usa-se a dimensão das n-tuplas do conjunto elementar correspondente). Se nenhuma informação de dimensão é fornecida, assume-se `dimen 1`.

O atributo `within` especifica uma expressão de conjunto cujo valor resultante é um superconjunto usado para restringir o conjunto (se o conjunto é simples) ou seus membros (se o conjunto é uma matriz de conjuntos elementares) a estar incluído naquele superconjunto. Um número arbitrário de atributos `within` podem ser especificados na mesma sentença set.

O atributo de atribuição (`:=`) especifica uma expressão de conjunto usada para avaliar conjunto(s) elementar(es) atribuído(s) ao conjunto (se o conjunto é simples) ou seus membros (se o conjunto é uma matriz de conjuntos elementares). Se o atributo de atribuição é especificado, o conjunto é *calculável*, portanto, não há a necessidade de fornecer dados na seção de dados. Se o atributo de atribuição não é especificado, deve-se fornecer os dados na seção de dados. Pode-se especificar no máximo um atributo de atribuição ou `default` para o mesmo conjunto.

O atributo `default` especifica uma expressão de conjunto usado para avaliar conjunto(s) elementar(es) atribuído(s) ao conjunto (se o conjunto é simples) ou seus membros (se o conjunto é uma matriz de conjuntos elementares) sempre que não houver dados apropriados disponíveis na seção de dados. Se não se especifica nem o atributo de atribuição nem o atributo `default`, a falta de dados causará um erro.

4.2 Sentença parameter

`param` *nome alias domínio* `,` *attrib* `,` ... `,` *attrib* `;`

nome é um nome simbólico do parâmetro;

alias é um literal de cadeia opcional que especifica um pseudônimo para o parâmetro;

domínio é uma expressão indexante opcional que especifica o domínio do subíndice do parâmetro;

atributo, ..., *atributo* são atributos opcionais do parâmetro (as vírgulas que precedem os atributos podem ser omitidas.)

Atributos opcionais

`integer`
especifica que o parâmetro é inteiro;

`binary`
especifica que o parâmetro é binário;

`symbolic`
especifica que o parâmetro é simbólico;

expressão de relação
(onde *relação* é algum de: `<`, `<=`, `=`, `==`, `>=`, `>`, `<>`, `!=`)
especifica uma condição que restringe o parâmetro ou seus membros a satisfazer aquela condição;

`in` *expressão*
especifica um superconjunto que restringe o parâmetro ou seus membros a estarem inseridos naquele superconjunto;

`:=` *expressão*
especifica um valor atribuído ao parâmetro ou a seus membros;

`default` *expressão*
especifica um valor atribuído ao parâmetro ou aos seus membros sempre que não houverem dados disponíveis na seção de dados.

Exemplos

```
param unidades{insumo, produto} >= 0;
param lucro{produto, 1..T+1};
param N := 20 integer >= 0 <= 100;
param combinacao 'n escolhe k' {n in 0..N, k in 0..n} :=
   if k = 0 or k = n then 1 else combinacao[n-1,k-1] + combinacao[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 mes symbolic default 'Mai' in {'Mar', 'Abr', 'Mai'};
```

A sentença parameter declara um parâmetro. Se o domínio de subíndice não é especificado, o parâmetro é simples (escalar), caso contrário, é uma matriz n-dimensional.

Os atributos de tipo `integer`, `binary` e `symbolic` qualificam os tipos de valores que podem ser atribuídos ao parâmetro, conforme demonstrado:

Tipo de atributo	Valores atribuídos
(não especificado)	Qualquer valor numérico
`integer`	Apenas valores numéricos inteiros
`binary`	Tanto 0 quanto 1
`symbolic`	Qualquer valor numérico e simbólico

O atributo `symbolic` não pode ser especificado juntamente com outros tipos de atributos. Uma vez especificado, ele deve preceder todos os outros atributos.

O atributo de condição especifica uma condição opcional que restringe os valores atribuídos ao parâmetro para satisfazer aquela condição. Este atributo tem as seguintes formas sintáticas:

`<` v	verifica se $x < v$
`<=` v	verifica se $x \leq v$
`=` v, `==` v	verifica se $x = v$
`>=` v	verifica se $x \geq v$
`>` v	verifica se $x \geq v$
`<>` v, `!=` v	verifica se $x \neq v$

onde x é um valor atribuído ao parâmetro, v é o valor resultante de uma expressão numérica ou simbólica especificado no atributo de condição. Um número arbitrário de atributos de condição pode ser especificado para o mesmo parâmetro. Se, durante a avaliação do modelo, um valor atribuído ao parâmetro viola pelo menos uma das condições especificadas, ocorrerá um erro. (Note que valores simbólicos são ordenados de forma lexicográfica e qualquer valor numérico precede qualquer valor simbólico.)

O atributo `in` é similar ao atributo de condição e especifica uma expressão de conjunto cujo valor resultante é um superconjunto usado para restringir valores numéricos ou simbólicos atribuídos ao parâmetro a estarem incluídos naquele superconjunto. Pode-se especificar um número arbitrário de atributos `in` para o mesmo parâmetro. Se, durante a avaliação do modelo, o valor atribuído ao parâmetro não pertence a pelo menos um dos superconjuntos especificados, ocorrerá um erro.

O atributo de atribuição (`:=`) especifica uma expressão numérica ou simbólica usada para computar um valor atribuído ao parâmetro (se é um parâmetro simples) ou seus membros (se o parâmetro é uma matriz). Se o atributo de atribuição é especificado, o parâmetro é *calculável*, portanto, não há a necessidade de fornecer dados na seção de dados. Se o atributo de atribuição não é especificado, deve-se fornecer os dados para o parâmetro na seção de dados. Pode-se especificar no máximo um atributo de atribuição ou `default` para o mesmo parâmetro.

O atributo `default` especifica uma expressão numérica ou simbólica usada para computar um valor atribuído ao parâmetro ou seus membros sempre que não houver dados apropriados disponíveis na seção de dados. Se não se especifica nem o atributo de atribuição nem o atributo `default`, a falta de dados causará um erro.

4.3 Sentença variable

`var` *nome alias domínio* , *atrib* , . . . , *atrib* ;

nome é um nome simbólico da variável;

alias é um literal de cadeia opcional que especifica um pseudônimo para a variável;

domínio é uma expressão indexante opcional que especifica o domínio do subíndice da variável;

atrib, . . . , *atrib* são atributos opcionais da variável (as vírgulas que precedem os atributos podem ser omitidas.)

Atributos opcionais

`integer`
: restringe a variável a ser inteira;

`binary`
: restringe a variável a ser binária;

`>=` *expressão*
: especifica um limite inferior para a variável;

`<=` *expressão*
: especifica um limite superior para a variável;

`=` *expressão*
: especifica um valor fixo para a variável;

Exemplos

```
var x >= 0;
var y{I,J};
var produzir{p in prod}, integer, >= comprometido[p], <= mercado[p];
var armazenar{insumo, 1..T+1} >= 0;
var z{i in I, j in J} >= i+j;
```

A sentença variable declara uma variável. Se não se especifica o domínio do subíndice, a variável e uma variável simples (escalar), caso contrário é uma matriz n-dimensional de variáveis elementares.

As variáveis elementares associadas com a variável do modelo (se é uma variável simples) ou seus membros (se é uma matriz) corresponde às variáveis na formulação do problema PL/PIM (ver Seção 1.1, página 6). Note que somente variáveis elementares realmente referenciadas em algumas restrições e/ou objetivos serão incluídas na instância do problema PL/PIM a ser gerado.

Os atributos de tipo `integer` e `binary` restringem a variável a ser inteira ou binária, respectivamente. Se nenhum atributo de tipo é especificado, a variável é contínua. Se todas as variáveis no modelo são contínuas, o problema correspondente é da classe PL. Se há pelo menos uma variável inteira ou binária, o problema é da classe PIM.

O atributo de limite inferior (>=) especifica uma expressão numérica para calcular um limite inferior da variável. No máximo um limite inferior pode ser especificado. Por padrão, todas as variáveis (exceto as binárias) não tem limite inferior, assim, se há a necessidade de uma variável ser não-negativa, seu limite inferior zero deve ser explicitamente especificado.

O atributo de limite superior (<=) especifica uma expressão numérica para calcular um limite superior da variável. No máximo um limite superior pode ser especificado.

O atributo de valor fixo (=) especifica uma expressão numérica para calcular um valor no qual a variável é fixada. Este atributo não pode ser especificado junto com os atributos de limite.

4.4 Sentença constraint

`s.t.` *nome alias domínio* `:` *expressão* `,` `=` *expressão* `;`

`s.t.` *nome alias domínio* `:` *expressão* `,` `<=` *expressão* `;`

`s.t.` *nome alias domínio* `:` *expressão* `,` `>=` *expressão* `;`

`s.t.` *nome alias domínio* `:` *expressão* `,` `<=` *expressão* `,` `<=` *expressão* `;`

`s.t.` *nome alias domínio* `:` *expressão* `,` `>=` *expressão* `,` `>=` *expressão* `;`

nome é um nome simbólico da restrição;

alias é um literal de cadeia opcional que especifica um pseudônimo da restrição;

domínio é uma expressão indexante opcional, que especifica o domínio do subíndice da restrição;

expressão é uma expressão linear usada para calcular um componente da restrição (as vírgulas que precedem os atributos podem ser omitidas).

(A palavra-chave `s.t.` pode ser escrita como `subject to`, como `subj to` ou pode ser omitido por completo).

Exemplos

```
s.t. r: x + y + z, >= 0, <= 1;
limite{t in 1..T}: sum{j in produto} produzir[j,t] <= max_prod;
subject to balanco{i in insumo, t in 1..T}:
   estoque[i,t+1] = estoque[i,t] - sum{j in produto} unidades[i,j] * produzir[j,t];
subject to rlim 'limite tempo-regular' {t in tempo}:
   sum{p in produto} pt[p] * rprod[p,t] <= 1.3 * dpp[t] * equipes[t];
```

A sentença de restrição declara uma restrição. Se o domínio do subíndice não é especificado, a restrição é uma restrição simples (escalar), caso contrário, é uma matriz n-dimensional de restrições elementares.

Restrições elementares associadas com a restrição do modelo (se é uma restrição simples) ou seus membros (se é uma matriz) correspondem a restrições lineares na formulação do problema de PL/PIM (ver Seção 1.1, página 6).

Se a restrição possui a forma de igualdade ou desigualdade simples, i.e., inclui duas expressões, uma segue depois dos dois pontos e a outra segue depois do sinal de relação `=`, `<=` ou `>=`, ambas expressões na sentença podem ser expressões lineares. Se a restrição possui a forma de uma desigualdade dupla, i.e., inclui três expressões, a expressão do meio pode ser uma expressão linear, enquanto a da esquerda e a da direita podem ser apenas expressões numéricas.

Gerar o modelo é, a grosso modo, gerar suas restrições, que são sempre avaliadas para todo domínio do subíndice. Avaliar as restrições, por sua vez, leva a avaliação de outros objetos de modelo tais como conjuntos, parâmetros e variáveis.

A construção de uma restrição linear incluída na instância do problema, que corresponde a uma restrição elementar particular, é realizada como segue.

Se a restrição possui a forma de igualdade ou desigualdade simples, a avaliação de ambas expressões lineares resultam em duas formas lineares:

$$f = a_1x_1 + a_2x_2 + \ldots + a_nx_n + a_0,$$
$$g = b_1x_1 + b_2x_2 + \ldots + b_nx_n + b_0,$$

onde x_1, x_2, $\ldots$, x_n são variáveis elementares; a_1, a_2, $\ldots$, a_n, b_1, b_2, $\ldots$, b_n são coeficientes numéricos; a_0 e b_0 são termos constantes. Em seguida, todos os termos lineares de f e g são levados ao lado esquerdo, enquanto que os termos constantes são levados ao lado direito, resultando na restrição elementar final na forma padrão:

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

Se a restrição possui a forma de desigualdade dupla, a avaliação da expressão linear do meio resulta na seguinte forma linear:

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

e a avaliação das expressões numéricas da esquerda e da direita dão dois valores numéricos l e u, respectivamente. Logo, o termo constante da forma linear é levado tanto à esquerda como à direita para gerar a restrição elementar final na forma padrão:

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

4.5 Sentença objective

`minimize` *nome alias domínio* `:` *expressão* `;`

`maximize` *nome alias domínio* `:` *expressão* `;`

nome é um nome simbólico do objetivo;

alias é uma literal de cadeia opcional que especifica um pseudônimo do objetivo;

domínio é uma expressão indexante opcional que especifica um domínio do subíndice do objetivo;

expressão é uma expressão linear usada pra calcular a forma linear do objetivo.

Exemplos

```
minimize obj: x + 1.5 * (y + z);
maximize lucro_total: sum{p in produto} lucro[p] * produzir[p];
```

A sentença objective declara um objetivo. Se o domínio do subíndice não é especificado, o objetivo é um objetivo simples (escalar). Caso contrário, é uma matriz n-dimensional de objetivos elementares.

Objetivos elementares associados com o objetivo do modelo (se é um objetivo simples) ou seus membros (se é uma matriz) correspondem a restrições lineares genéricas na formulação do problema PL/PIM (ver Seção 1.1, página 6). No entanto, diferentemente das restrições, estas formas lineares são livres (ilimitadas).

A construção de uma forma linear incluída na instância do problema, a qual corresponde a uma restrição elementar particular, é realizada como segue. A expressão linear especificada da na sentença objective é avaliada para resultar na seguinte forma linear:

$$f = a_1 x_1 + a_2 x_2 + \ldots + a_n x_n + a_0,$$

onde x_1, x_2, ..., x_n são variáveis elementares; a_1, a_2, ..., a_n são coeficientes numéricos; a_0 é o termo constante. Logo, a forma linear é usada para construir a restrição final elementar na forma padrão:

$$-\infty < a_1 x_1 + a_2 x_2 + \ldots + a_n x_n + a_0 < +\infty.$$

Como via de regra, a descrição do modelo contém apenas uma sentença objective que define a função objetivo usada na instância do problema. No entanto, é permitido declarar uma quantidade arbitrária de objetivos. Neste caso, a função objetivo real será o primeiro objetivo encontrado na descrição do modelo. Outros objetivos também estão incluídos na instância do problema, mas eles não afetam a função objetivo.

4.6 Sentença solve

```
solve ;
```

A sentença solve é opcional e pode ser usada apenas uma vez. Se a sentença solve não é usada, ela é assumida ao final da seção de modelo.

A sentença solve provoca que o modelo seja resolvido, o que significa calcular os valores numéricos de todas as variáveis do modelo. Isto permite usar variáveis em sentenças abaixo da sentença solve como se fossem parâmetros numéricos.

Note que a sentença variable, constraint e objective não podem ser usadas abaixo da sentença solve, i.e., todos os principais componentes do modelo devem ser declarados acima da sentença solve.

4.7 Sentença check

```
check domínio : expressão ;
```

domínio é uma expressão indexante opcional que especifica o domínio do subíndice da sentença check;

expressão é uma expressão lógica que especifica a condição lógica a ser verificada (os dois pontos que precedem a *expressão* podem ser omitidos).

Exemplos

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

A sentença check permite a verificação do valor resultante de uma expressão lógica especificada na sentença. Se o valor é *falso*, um erro é reportado.

Se o domínio do subíndice não é especificado, a verificação é realizada apenas uma vez. Especificar o domínio do subíndice permite a execução de verificações múltiplas para cada n-tupla no conjunto domínio. Neste último caso, a expressão lógica pode incluir índices introduzidos na expressão indexante correspondente.

4.8 Sentença display

```
display domínio : item , ... , item ;
```

domínio é uma expressão indexante opcional que especifica um domínio do subíndice da sentença display;

item, ..., *item* são itens a serem mostrados (os dois pontos que precedem o primeiro item podem ser omitidos).

Exemplos

```
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];
```

A sentença display avalia todos itens especificados na sentença e escreve seus valores em saída padrão (terminal) em formato de texto plano.

Se um domínio de subíndice não é especificado, os itens são avaliados e mostrados apenas uma vez. Ao especificar o domínio do subíndice, itens são avaliados e mostrados para cada n-tupla no conjunto do domínio. No último caso, os itens podem incluir índices introduzidos na expressão indexante correspondente.

Um item a ser mostrado pode ser um objeto de modelo (conjunto, parâmetro, variável, restrição,

objetivo) ou uma expressão.

Se um item é um objeto calculável (i.e., um conjunto ou parâmetro com o atributo de atribuição), o objeto é avaliado por todo domínio e em seguida, seu conteúdo (i.e., o conteúdo da matriz de objetos) é mostrado. Caso contrário, se o item não é um objeto calculável, somente seu seu conteúdo corrente (i.e., os membros realmente gerados durante a avaliação do modelo) é mostrado.

Se o item é uma expressão, a expressão é avaliada e seu valor resultante é mostrado.

4.9 Sentença printf

`printf` *domínio* `:` *formato* `,` *expressão* `,` ... `,` *expressão* `;`

`printf` *domínio* `:` *formato* `,` *expressão* `,` ... `,` *expressão* `>` *nome-do-arquivo* `;`

`printf` *domínio* `:` *formato* `,` *expressão* `,` ... `,` *expressão* `»` *nome-do-arquivo* `;`

domínio é uma expressão indexante opcional que especifica o domínio do subíndice da sentença printf;

formato é uma expressão simbólica cujo valor especifica uma cadeia de controle de formato (os dois pontos que precedem a expressão de formato podem ser omitidos).

expressão, ..., *expressão* são zero ou mais expressões cujos valores devem ser formatados e impressos. Cada expressão deve ser de tipo numérico, simbólico ou lógico.

nome-do-arquivo é uma expressão simbólica cujo valor especifica um nome de um arquivo de texto para onde a saída é redirecionada. O sinal > significa criar um novo aquivo vazio, enquanto o sinal » significa acrescentar a saída a um arquivo existente. Se o nome do arquivo não é especificado, a saída é escrita na saída padrão (terminal).

Exemplos

```
printf 'Ola, mundo!\n';
printf: "x = %.3f; y = %.3f; z = %.3f\n", x, y, z > "resultado.txt";
printf{i in I, j in J}: "fluxo de %s para %s eh %d\n", i, j, x[i,j]
   >> arquivo_resultado & ".txt";
printf{i in I} 'fluxo total de %s eh %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];
```

A sentença printf é similar a sentença display, no entanto, ela permite formatar os dados a serem escritos.

Se um domínio do subíndice não é especificado, a sentença printf é executada apenas uma vez. Especificar um domínio do subíndice gera a execução da sentença printf para cada n-tupla no conjunto do domínio. No último caso, o formato e a expressão podem incluir índices introduzidos nas expressões indexantes correspondentes.

A cadeia de controle de formato é valor da expressão simbólica *formato* especificada na sentença

printf. Ela é composta de zero ou mais diretivas, como segue: tanto caracteres ordinários (exceto `%`), que são copiados sem modificação ao fluxo de saída, quanto especificações de conversão, provocam a avaliação da expressão correspondente especificada na sentença printf, do seu formato e da escrita do valor resultante no fluxo de saída.

As especificações de conversão que podem ser usadas na cadeia de controle de formato são as seguintes: `d`, `i`, `f`, `F`, `e`, `E`, `g`, `G` e `s`. Estas especificações possuem a mesma sintaxe e semântica que na linguagem de programação C.

4.10 Sentença for

`for` *domínio* `:` *sentença* `;`

`for` *domínio* `:` `{` *sentença* ... *sentença* `}` `;`

domínio é uma expressão indexante que especifica um domínio do subíndice da sentença for. (Os dois pontos que seguem a expressão indexante podem ser omitidos).

sentença é uma sentença que deve ser executada sob o controle da sentença for;

sentença, ..., *sentença* é uma sequência de sentenças (cercada entre chaves) que deve ser executada sob o controle da sentença for.

Apenas as sentenças seguintes podem ser usadas dentro da sentença for: check, display, printf e outro for.

Exemplos

```
for {(i,j) in E: i != j}
{  printf "fluxo de %s para %s eh %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("*");
```

A sentença for faz com que a sentença, ou uma sequência de sentenças especificadas como parte da sentença for, seja executada para cada n-tupla no conjunto do domínio. Assim, sentenças dentro da sentença for podem incluir índices introduzidos na expressão indexante correspondente.

4.11 Sentença table

```
table nome alias IN controlador arg ... arg :
      conjunto <- [ cmp , ... , cmp ] , par ~ cmp , ... , par ~ cmp ;
table nome alias domínio OUT controlador arg ... arg :
      expr ~ cmp , ... , expr ~ cmp ;
```

nome é um nome simbólico da tabela;

alias é um literal de cadeia opcional que especifica um pseudônimo da tabela;

domínio é uma expressão indexante que especifica o domínio do subíndice da tabela (de saída);

`IN` significa ler dados de uma tabela de entrada;

`OUT` significa escrever dados em uma tabela de saída;

controlador é uma expressão simbólica que especifica o controlador usado para acessar a tabela (para mais detalhes, ver Apêndice C, página 63);

arg é uma expressão simbólica opcional, que é um argumento passado ao controlador da tabela. Esta expressão simbólica não deveria incluir índices especificados no domínio;

conjunto é o nome de um conjunto simples opcional chamado *conjunto de controle*. Pode ser omitido junto com o delimitador `<-`;

cmp é um nome de campo. Entre colchetes, pelo menos um campo deve ser especificado. O nome do campo, que segue o nome do parâmetro ou de uma expressão, é opcional e pode ser omitido juntamente com o delimitador `~`. Neste caso o nome do objeto de modelo correspondente é usado como nome de campo;

par é um nome simbólico de um parâmetro do modelo;

expr é uma expressão numérica ou simbólica.

Exemplos

```
table dados IN "CSV" "dados.csv": S <- [DE,PARA], d~DISTANCIA,
   c~CUSTO;
table resultado{(d,p) in S} OUT "CSV" "resultado.csv": d~DE, p~PARA,
   x[d,p]~FLUXO;
```

A sentença table permite a leitura de dados de uma tabela para objetos de modelo como conjuntos e parâmetros (não-escalares) assim como escrever dados do modelo para uma tabela.

4.11.1 Estrutura de tabelas

Uma *tabela de dados* é um conjunto (desordenado) de *registros*, onde cada registro consiste do mesmo número de *campos* e cada campo possui um único nome simbólico denominado o *nome do campo*. Por exemplo:

		Primeiro campo ↓	Segundo campo ↓	. . .	Último campo ↓
Cabeçalho da tabela	→	DE	PARA	DISTANCIA	CUSTO
Primeiro registro	→	Seattle	New-York	2.5	0.12
Segundo registro	→	Seattle	Chicago	1.7	0.08
		Seattle	Topeka	1.8	0.09
. . .		San-Diego	New-York	2.5	0.15
		San-Diego	Chicago	1.8	0.10
Último registro	→	San-Diego	Topeka	1.4	0.07

4.11.2 Lendo dados de uma tabela de entrada

A sentença tabela de entrada faz a leitura de dados da tabela especificada, registro por registro.

Uma vez que o registro subsequente foi lido, valores numéricos ou simbólicos dos campos, cujos nomes são cercados entre colchetes na sentença table, são reunidos em um n-tuplo. Se o conjunto de controle é especificado na sentença table, este n-tuplo é adicionado a ele. Além disso, um valor numérico ou simbólico de cada campo associado com um parâmetro do modelo é atribuído ao membro do parâmetro identificado por subíndices, que são componentes da n-tupla que acabou de ser lida.

Por exemplo, a seguinte sentença de tabela de entrada:

```
table dados IN "...": S <- [DE,PARA], d~DISTANCIA, c~CUSTO;
```

faz a leitura de valores de quatro campos chamados `DE`, `PARA`, `DISTANCIA` e `CUSTO` de cada registro da tabela especificada. Os valores dos campos `DE` e `PARA` dão um par (f, t), que é adicionado ao conjunto de controle `S`. O valor do campo `DISTANCIA` é atribuído ao membro do parâmetro `d`$[f, t]$ enquanto que o valor do campo `CUSTO` é atribuído ao membro do parâmetro `c`$[f, t]$.

Note que a tabela de entrada pode conter campos adicionais cujos nomes não sejam especificados na sentença tabela, neste caso, os valores destes campos serão ignorados na leitura da tabela.

4.11.3 Escrevendo dados em uma tabela de saída

A sentença tabela de saída gera a escrita de dados na tabela especificada. Note que alguns controladores (chamados CSV e xBASE) destroem a tabela de saída antes de escrever os dados, i.e., deletam todos os registros existentes.

Cada n-tupla no domínio do conjunto especificado gera um registro escrito na tabela de saída. Os valores dos campos são valores numéricos ou simbólicos das expressões correspondentes especificadas

na sentença table. Estas expressões são avaliadas para cada n-tupla no conjunto do domínio, portanto, podem incluir índices introduzidos na expressão indexante correspondente.

Por exemplo, a seguinte sentença da tabela de saída:

```
table resultado{(f,t) in S} OUT "...": f~DE, t~PARA, x[f,t]~FLUXO;
```

gera a escrita de registros; um registro para cada par (f, t) no conjunto `S` para a tabela de saída, onde cada registro consiste de três campos chamados `DE`, `PARA` e `FLUXO`. Os valores escritos nos campos `DE` e `PARA` são os valores correntes dos índices `f` e `t`. O valor escrito no campo `FLUXO` é um valor do membro `x`$[f, t]$ do correspondente parâmetro ou variável indexada.

Capítulo 5

Dados do modelo

Os *dados do modelo* includem conjuntos elementares, que são "valores" dos conjuntos do modelo, e valores numéricos e simbólicos dos parâmetros do modelo.

Em MathProg existem duas formas diferentes de fornecer valores aos conjuntos e parâmetros do modelo. Uma forma é simplesmente prover os dados necessários usando o atributo de atribuição. No entanto, em muitos casos é mais prático separar o modelo próprio dos dados particulares necessários para o modelo. Para o último caso, em MathProg há uma outra forma, em que a descrição do modelo é dividida em duas partes: a seção de modelo e a seção de dados.

A *seção de modelo* é a principal parte da descrição do modelo. Ela contém todas as declarações de todos objetos do modelo, sendo comum a todos problemas baseados naquele modelo.

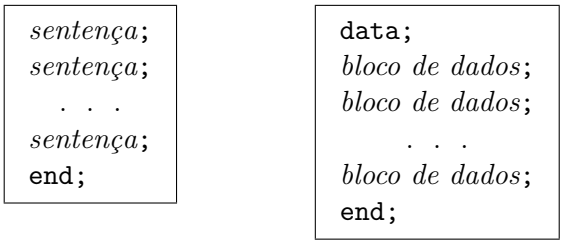
A *seção de dados* é uma parte opcional da descrição do modelo que contém dados específicos para um problema particular.

Em MathProg seções de modelo e de dados podem ser localizadas tanto em um arquivo de texto ou em dois arquivos de texto separados.

1. Se ambas seções de modelo e de dados estão localizados em um arquivo, o arquivo é composto como segue:

sentença;
sentença;
. . .
sentença;
`data`;
bloco de dados;
bloco de dados;
. . .
bloco de dados;
`end`;

2. Se a seção de modelo e dados são posicionados em dois arquivos separados, os arquivos são compostos como segue:



Arquivo de modelo Arquivo de dados

Nota: Se a seção de dados é posicionada em um arquivo separado, a palavra-chave `data` é opcional e pode ser omitida juntamente como o ponto e vírgula que a segue.

5.1 Programando a seção de dados

A *seção de dados* é uma sequência de blocos de dados em vários formatos e são discutidos nas seções seguintes. A ordem na qual os blocos de dados seguem na seção de dados pode ser arbitrária, portanto, não precisa ser necessariamente a mesma ordem que seguem os elementos correspondentes da seção de modelo.

As regras para programar a seção de dados são comumente as mesmas que as regras de programar a descrição do modelo (ver Seção 2, página 9), i.e., blocos de dados são compostos com unidades léxicas básicas, como nomes simbólicos, literais numéricos e de cadeia, palavras-chave, delimitadores e comentários. No entanto, por conveniência e para melhorar legibilidade, há um desvio da regra comum: se um literal de cadeia consiste unicamente de caracteres alfanuméricos (incluindo o caractere sublinhado), os sinais `+` e `-` e/ou o ponto decimal, ele pode ser programado sem aspas limitadoras (simples ou duplas).

Todo material numérico e simbólico provido na seção de dados é programado na forma de números e símbolos, i.e., diferentemente da seção de modelo, não são permitidas expressões na seção de dados. Apesar disso, os sinais `+` e `-` podem preceder literais numéricos para permitir a programação de quantidades numéricas com sinais. Neste caso não deve haver caractere de espaço em branco entre o sinal e o literal numérico seguinte (se houver pelo menos uma espaço em branco, o sinal e o literal numérico seguinte são reconhecidos como duas unidades léxicas diferentes).

5.2 Bloco de dados set

`set` *nome* , *registro* , . . . , *registro* ;

`set` *nome* `[` *símbolo* , . . . , *símbolo* `]` , *registro* , . . . , *registro* ;

nome é um nome simbólico do conjunto;

símbolo, . . . , *símbolo* são subíndices que especificam um membro particular do conjunto (se o conjunto é uma matriz, i.e., um conjunto de conjuntos);

registro, . . . , *registro* são registros.

As vírgulas que precedem os registros podem ser omitidas.

Registros

`:=`
: é um elemento de atribuição de registro não-significativo que pode ser usado livremente para melhorar a legibilidade;

`(` *fatia* `)`
: especifica uma fatia;

dados-simples
: especifica os dados do conjunto em formato simples;

`:` *dados matriciais*
: especifica os dados do conjunto em formato de matriz;

`(tr) :` *dados matriciais*
: especifica os dados do conjunto em formato de matriz transposta. (Neste caso, os dois pontos que seguem a palavra-chave `(tr)` podem ser omitidos).

Exemplos

```
set mes := Jan Fev Mar Abr Mai Jun;
set mes "Jan", "Fev", "Mar", "Abr", "Mai", "Jun";
set A[3,Mar] := (1,2) (2,3) (4,2) (3,1) (2,2) (4,4) (3,4);
set A[3,'Mar'] := 1 2 2 3 4 2 3 1 2 2 4 4 3 4;
set A[3,'Mar'] : 1 2 3 4 :=
               1 - + - -
               2 - + + -
               3 + - - +
               4 - + - + ;
set B := (1,2,3) (1,3,2) (2,3,1) (2,1,3) (1,2,2) (1,1,1) (2,1,1);
set B := (*,*,*) 1 2 3, 1 3 2, 2 3 1, 2 1 3, 1 2 2, 1 1 1, 2 1 1;
set B := (1,*,2) 3 2 (2,*,1) 3 1 (1,2,3) (2,1,3) (1,1,1);
set B := (1,*,*) : 1 2 3 :=
                 1 + - -
                 2 - + +
```

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

(Nestes exemplos `mes` é um conjunto simples de singletos, `A` é uma matriz 2-dimensional de duplas e `B` é um conjunto simples de triplas. Os blocos de dados para o mesmo conjunto são equivalentes no sentido que especificam os mesmos dados em formatos distintos.)

O *bloco de dados do conjunto* é usado para especificar um conjunto elementar completo que é atribuído a um conjunto (se é um conjunto simples) ou a um de seus membros (se o conjunto é uma matriz de conjuntos).[1]

Blocos de dados podem ser especificados somente para conjuntos não-calculáveis, i.e., para conjuntos que possuem o atributo de atribuição (`:=`) na sentença set correspondente.

Se o conjunto é um conjunto simples, somente seus nomes simbólicos devem ser especificados no cabeçalho do bloco de dados. Caso contrário, se o conjunto é uma matriz n-dimensional, seus nomes simbólicos devem ser fornecidos com uma lista completa de subíndices separados por vírgulas e cercados em colchetes para especificar um membro particular da matriz de conjuntos. O número de subíndices deve ser igual ao da dimensão da matriz de conjuntos, onde cada subíndice deve ser um número ou um símbolo.

Um conjunto elementar definido no bloco de dados é programado como uma sequência de registros descritos abaixo.[2]

5.2.1 Registro de atribuição de dados

O *registro de atribuição de dados* (`:=`) é um elemento não-significante. Ele pode ser usado para melhorar a legibilidade de blocos de dados.

5.2.2 Registro em fatia de dados

O *registro em fatia de dados* é um registro de controle que especifica uma *fatia* do conjunto elementar definido no bloco de dados. Ele possui a seguinte forma sintática:

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

onde s_1, s_2, ..., s_n são componentes da fatia.

Cada componente da fatia pode ser um número, símbolo ou asterisco (`*`). O número de componentes na fatia deve ser o mesmo da dimensão n-tuplas do conjunto elementar a ser definido. Por exemplo, se o conjunto elementar contém 4-tuplas (quádruplas), a fatia deve ter quatro componentes. O número de asteriscos na fatia denomina a *dimensão da fatia*.

[1] Há uma outra forma de especificar dados para um conjunto simples com dados para os parâmetros. Esta questão é discutida na próxima seção.

[2] *Registro* é simplesmente um termo técnico. Não significa que os mesmos possuem qualquer formato especial.

O efeito de usar fatias é o seguinte: se uma fatia m-dimensional (i.e., uma fatia contendo m asteriscos) é especificada no bloco de dados, todos registros subsequentes devem especificar tuplas de dimensão m. Sempre que uma m-tupla é encontrada, cada asterisco da fatia é substituído pelos componentes correspondentes da m-tupla, o que resulta na n-tupla, que é incluída no conjunto elementar a ser definido. Por exemplo, se a fatia $(a, *, 1, 2, *)$ está vigente e a dupla $(3, b)$ é encontrada no registro subsequente, a 5-tupla resultante a ser incluída no conjunto elementar é $(a, 3, 1, 2, b)$.

Se a fatia não possui asteriscos, ela própria define uma n-tupla completa que é incluída no conjunto elementar.

Uma vez especificada uma fatia, a mesma está vigente até que apareça uma nova fatia ou até que se encontre o fim do bloco de dados. Note que se uma fatia não é especificada no bloco de dados, assume-se uma cujos componentes são asteriscos em todas as posições.

5.2.3 Registro simples

O *registro simples* define uma n-tupla em um formato simples e possui a seguinte forma sintática:

$$t_1 \ , \ t_2 \ , \ \ldots \ , \ t_n$$

onde t_1, t_2, ..., t_n são componentes da n-tupla. Cada componente pode ser um número ou um símbolo. As vírgulas entre os componentes são opcionais e podem ser omitidas.

5.2.4 Registro de matriz

O *registro de matriz* define diversas 2-tuplas (duplas) em um formato matricial e possui a seguinte forma sintática:

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

onde r_1, r_2, ..., r_m são números e/ou símbolos que correspondem a linhas da matriz; c_1, c_2, ..., c_n são números e/ou símbolos que correspondem a colunas da matriz, a_{11}, a_{12}, ..., a_{mn} são elementos da matriz, que podem ser tanto `+` como `-`. (Neste registro, o delimitador `:` que precede a lista de colunas e o delimitador `:=` que segue após a lista de colunas, não podem ser omitidos.)

Cada elemento a_{ij} do bloco de dados matricial (onde $1 \leq i \leq m$, $1 \leq j \leq n$) correspondem a 2-tuplas (r_i, c_j). Se a_{ij} é o sinal mais (`+`), a 2-tupla correspondente (ou uma n-tupla maior, se uma fatia é usada) é incluída no conjunto elementar. Caso contrário, se a_{ij} é o sinal menos (`-`), aquela 2-tupla não é incluída no conjunto elementar.

Uma vez que o registro de matriz define 2-tuplas, tanto o conjunto elementar quanto a fatia vigente devem ser 2-dimensional.

5.2.5 Registro de matriz transposta

O *registro de matriz transposta* possui a seguinte forma sintática:

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

(Neste caso, o delimitador `:` que segue a palavra-chave `(tr)` é opcional e pode ser omitido.)

Este registro é completamente análogo ao registro de matriz (ver anteriormente) com a única exceção de que neste caso, cada elemento a_{ij} da matriz passa a corresponder a 2-tupla (c_j, r_i) ao invés de (r_i, c_j).

Uma vez especificado, o indicador `(tr)` tem alcance em todos registros subsequentes até que se encontre outra fatia ou o fim do bloco de dados.

5.3 Bloco de dados de parâmetro

`param` *nome* `,` *registro* `,` ... `,` *registro* `;`

`param` *nome* `default` *valor* `,` *registro* `,` ... `,` *registro* `;`

`param :` *dados-tabulação* `;`

`param default` *valor* `:` *dados-tabulação* `;`

nome é um nome simbólico do parâmetro;

valor é um valor opcional padrão do parâmetro;

registro, ..., *registro* são registros;

dados-tabulação especifica os dados do parâmetro em formato tabulação.

As vírgulas que precedem os registros podem ser omitidas.

Registros

`:=`
: é um elemento de atribuição de registro não-significativo que pode ser usado livremente para melhorar a legibilidade;

`[` *fatia* `]`
: especifica uma fatia;

dados-planos
: especifica os dados do parâmetro em formato simples;

`:` *dados-tabulares*
especifica dados do parâmetro em formato tabular;

`(tr)` : *dados-tabulares*
especifica dados do parâmetro no formato tabular transposto. (Neste caso, os dois pontos que seguem a palavra-chave `(tr)` podem ser omitidos).

Exemplos

```
param T := 4;
param mes := 1 Jan 2 Fev 3 Mar 4 Abr 5 Mai;
param mes := [1] 'Jan', [2] 'Fev', [3] 'Mar', [4] 'Abr', [5] 'Mai';
param estoque_inicial := ferro 7.32 niquel 35.8;
param estoque_inicial [*] ferro 7.32, niquel 35.8;
param custo [ferro] .025 [niquel] .03;
param valor := ferro -.1, niquel .02;
param       : estoque_inicial  custo  valor :=
      ferro      7.32     .025   -.1
      niquel    35.8      .03     .02 ;
param : insumo : estoque_inicial  custo  valor :=
        ferro    7.32     .025   -.1
        niquel  35.8      .03     .02 ;
param demanda default 0 (tr)
       :  FRA  DET  LAN  WIN  STL  FRE  LAF :=
   chapa  300   .   100   75   .   225  250
   bobina 500  750  400  250   .   850  500
   placa  100   .    .    50  200   .   250 ;
param custo_transporte :=
   [*,*,chapa]:  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
   [*,*,bobina]:  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
   [*,*,placa]:  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 ;
```

O *bloco de dados do parâmetro* é usado para especificar dados completos para um parâmetro (ou parâmetros, se os dados são especificados no formato tabulação).

Os blocos de dados podem ser especificados apenas para parâmetros não-calculáveis, i.e., para parâmetros que não possuem o atributo de atribuição (`:=`) nas sentenças parameter correspondentes.

Os dados definidos no bloco de dados do parâmetro são programados como uma sequência de registros descritos em seguida. Adicionalmente, o bloco de dados pode vir com o atributo opcional

`default`, que especifica um valor numérico ou simbólico padrão do parâmetro (ou parâmetros). Este valor padrão é atribuído ao parâmetro ou a seus membros quando não se definem valores apropriados no bloco de dados do parâmetro. O atributo `default` não pode ser usado se ele já tiver sido especificado na sentença parameter correspondente.

5.3.1 Registro de atribuição

O *registro de atribuição* (`:=`) é um elemento não-significativo. Ele pode ser usado para melhorar a legibilidade dos blocos de dados;

5.3.2 Registro em fatia

O *registro em fatia* é um registro de controle que especifica uma *fatia* da matriz do parâmetro. Ele tem a seguinte forma sintática:

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

onde s_1, s_2, ..., s_n são componentes da fatia.

Cada componente da fatia pode ser um número, símbolo ou asterisco (`*`). O número de componentes na fatia deve ser o mesmo da dimensão n-tuplas do parâmetro. Por exemplo, se o parâmetro é uma matriz 4-dimensional, a fatia deve ter quatro componentes. O número de asteriscos na fatia denomina a *dimensão da fatia*.

O efeito de usar fatias é o seguinte: se uma fatia m-dimensional (i.e., uma fatia contendo m asteriscos) é especificada no bloco de dados, todos registros subsequentes devem especificar os subíndices do membros do parâmetro, como se o parâmetro fosse m-dimensional, não n-dimensional.

Sempre que m subíndices são encontrados, cada asterisco da fatia é substituído pelos componentes correspondentes que dão n subíndices, que definem o membro corrente do parâmetro. Por exemplo, se a fatia $(a, *, 1, 2, *)$ está vigente e os subíndices 3 e b são encontradas no registro subsequente, a lista completa de subíndices usada para escolher o membro do parâmetro é $(a, 3, 1, 2, b)$.

É permitido especificar uma fatia que não tenha asteriscos. Tal fatia, em si própria, define uma lista completa de subíndice, em cujo caso o próximo registro deve definir apenas um único valor do membro correspondentes do parâmetro.

Uma vez especificada uma fatia, a mesma está vigente até que apareça uma nova fatia ou até que se encontre o fim do bloco de dados. Note que se uma fatia não é especificada no bloco de dados, assume-se uma cujos componentes são asteriscos em todas as posições.

5.3.3 Registro plano

O *registro plano* define uma lista de subíndice e um valor individual no formato plano. Este registro possui a seguinte forma sintática:

$$t_1\ ,\ t_2\ ,\ \ldots\ ,\ t_n\ ,\ v$$

onde t_1, t_2, ..., t_n são subíndices e v é um valor. Cada subíndice, assim como o valor, pode ser um número ou um símbolo. As vírgulas que seguem os subíndices são opcionais e podem ser omitidas.

No caso de um parâmetro ou fatia 0-dimensional, o registro plano não possui subíndice e consiste de um valor individual apenas.

5.3.4 Registro tabular

O *registro tabular* define diversos valores onde cada valor é provido de dois subíndices. Este registro possui a seguinte forma sintática:

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

onde r_1, r_2, ..., r_m são números e/ou símbolos que correspondem a linhas da tabela; enquanto que c_1, c_2, ..., c_n são são números e/ou símbolos que correspondem a colunas da tabela, a_{11}, a_{12}, ..., a_{mn} são elementos da tabela. Cada elemento pode ser um número, símbolo ou o ponto decimal (`.`) individual. (neste registro, o delimitador `:` que precede a lista de colunas e o delimitador `:=` que segue após a lista de colunas, não podem ser omitidos).

Cada elemento a_{ij} do bloco de dados tabulares ($1 \leq i \leq m$, $1 \leq j \leq n$) define dois subíndices, onde o primeiro subíndice é r_i e o segundo é c_j. Estes subíndices são usados juntamente com a fatia vigente para formar a lista completa de subíndices que identifica um membro particular da matriz de parâmetros. Se a_{ij} é um número ou um símbolo, este valor é atribuído ao membro do parâmetro. No entanto, se a_{ij} é um ponto decimal individual, o membro é atribuído ao valor padrão especificado ou no bloco de dados do parâmetro ou na sentença parameter, ou ainda, se nenhum valor padrão é especificado, o membro permanece indefinido.

Uma vez que o registro tabular fornece dois subíndices para cada valor, tanto o parâmetro quanto a fatia vigente em uso devem ser 2-dimensional.

5.3.5 Registro tabular transposto

O *registro tabular transposto* possui a seguinte forma sintática:

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

(Neste caso, o delimitador `:` que segue a palavra-chave `(tr)` é opcional e pode ser omitida.)

Este registro é completamente análogo ao registro tabular (ver anteriormente) com a única exceção que o primeiro subíndice definido pelo elemento a_{ij} é c_j enquanto que o segundo é r_i.

Uma vez especificado, o indicador `(tr)` afeta todos registros subsequentes até que se encontre outra fatia ou o fim do bloco de dados.

5.3.6 Formato de dados em tabulação

O bloco de dados do parâmetro no *formato tabulação* possui a seguinte forma sintática:

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

1. A palavra-chave `default` pode ser omitida juntamente com o valor que a segue.
2. O nome simbólico s pode ser omitido juntamente com os dois pontos que o segue.
3. Todas as vírgulas são opcionais e podem ser omitidas.

O bloco de dados no formato tabulação mostrado acima é exatamente equivalente aos seguintes blocos de dados:

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

$\texttt{param } p_1 \texttt{ default } \mathit{valor} \texttt{ :=}$

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

$\texttt{param } p_2 \texttt{ default } \mathit{valor} \texttt{ :=}$

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

$\quad . \ . \ . \ . \ . \ . \ . \ . \ .$

$\texttt{param } p_r \texttt{ default } \mathit{valor} \texttt{ :=}$

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

Apêndice A

Usando sufixos

Sufixos podem ser usados para recuperar valores adicionais associados com as variáveis, restrições e objetivos do modelo.

Um *sufixo* consiste de um ponto (`.`) seguido por uma palavra-chave não-reservada. Por exemplo, se `x` é uma variável bi-dimensional, `x[i,j].lb` é um valor numérico igual ao limite inferior da variável elementar `x[i,j]`, que (cujo valor) pode ser usado em expressões como um parâmetro numérico.

Para as variáveis do modelo, os sufixos possuem o seguinte significado:

`.lb` limite inferior (lower bound)

`.ub` limite superior (upper bound)

`.status` status na solução:
- 0 — indefinida
- 1 — básica
- 2 — não-básica no limite inferior
- 3 — não-básica no limite superior
- 4 — variável não-básica livre (ilimitada)
- 5 — variável não-básica fixa

`.val` valor primal na solução

`.dual` valor dual (custo reduzido) na solução

Para as restrições e objetivos do modelo, os sufixos têm os seguintes significados:

`.lb` limite inferior (lower bound) da forma linear

`.ub` limite superior (upper bound) da forma linear

`.status` status na solução:
- 0 — indefinida
- 1 — não-limitante
- 2 — limitante no limite inferior
- 3 — limitante no limite superior
- 4 — linha limitante livre (ilimitada)
- 5 — restrição de igualdade limitante

`.val` valor primal da forma linear na solução

`.dual` valor dual (custo reduzido) da forma linear na solução

Note que os sufixos `.status`, `.val` e `.dual` podem ser usados apenas abaixo da sentença solve.

Apêndice B

Funções de data e hora

por Andrew Makhorin <mao@gnu.org>
e Heinrich Schuchardt <heinrich.schuchardt@gmx.de>

B.1 Obtendo o tempo de calendário corrente

Para obter o tempo de calendário corrente em MathProg existe a função `gmtime`. Ela não possui argumentos e retorna o número de segundos transcorridos desde 00:00:00 de 1º de Janeiro de 1970, pelo Tempo Universal Coordenado (UTC). Por exemplo:

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

MathProg não possui uma função para converter o tempo UTC retornado pela função `gmtime` para os tempos de calendário *local*. Assim, para determinar o tempo de calendário local corrente, é preciso que adicione ao tempo UTC retornado a diferença de horas, com respeito a UTC, expressa em segundos. Por exemplo, a hora em Berlim durante o inverno é uma hora à frente do UTC, que corresponde a uma diferença horária de +1 hora = +3600 segundos, assim, o tempo de calendário corrente no inverno em Berlim pode ser determinado como segue:

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

De forma análoga, o horário de verão em Chicago (Zona Horária Central-CDT) é cinco horas atrás da UTC, de modo que o horário corrente do calendário local pode ser determinado como segue:

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

Note que o valor retornado por `gmtime` é volátil, i.e., ao ser chamada diversas vezes, esta função pode retornar diferentes valores.

B.2 Convertendo cadeia de caracteres ao tempo de calendário

A função `str2time(`*s*`,` *f*`)` converte uma cadeia de caractere (impressão da data e hora) especificada pelo seu primeiro argumento *s*, que deve ser uma expressão simbólica, para o tempo de

calendário apropriado para cálculos aritméticos. A conversão é controlada pela cadeia de formato especificado f (o segundo argumento), que também deve ser uma expressão simbólica.

A conversão resultante retornada por `str2time` possui o mesmo significado dos valores retornados pela função `gmtime` (ver Subseção B.1, página 58). Note que `str2time não corrige` o tempo de calendário retornado para zona horária local, i.e., ao se aplicar a 00:00:00 de 1º de Janeiro de 1970, ela sempre retorna 0.

Por exemplo, as sentenças de modelo:

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

produz a seguinte saída:

```
t = 900424020
```

onde o tempo de calendário impresso corresponde a 13:47:00 em 14 de Julho de 1998.

A cadeia de formato passada à função `str2time` consiste de especificadores de conversão e caracteres ordinários. Cada especificador de conversão inicia com um caractere de porcentagem (`%`) seguido de uma letra.

Os seguintes especificadores de conversão podem ser usados na cadeia de formato:

`%b` O nome do mês abreviado (insensível a maiúsculas). Pelo menos as três primeiras letras do mês devem aparecer na cadeia de entrada.

`%d` O dia do mês como número decimal (de 1 até 31). Se permite o zero como primeiro dígito, embora não seja necessário.

`%h` O mesmo que `%b`.

`%H` A hora como um número decimal, usando um relógio de 24-horas (de 0 a 23). Se permite o zero como primeiro dígito, embora não seja necessário.

`%m` O mês como um número decimal (de 1 a 12). Se permite o zero como primeiro dígito, embora não seja necessário.

`%M` O minuto como um número decimal (de 0 a 59). Se permite o zero como primeiro dígito, embora não seja necessário.

`%S` O segundo como um número decimal (de 0 to 60). Se permite o zero como primeiro dígito, embora não seja necessário.

`%y` O ano sem o século, como um número decimal (de 0 to 99). Se permite o zero como primeiro dígito, embora não seja necessário. Valores de entrada de 0 a 68 são considerados dos anos 2000 a 2068 enquanto que os valores 69 até 99 como dos anos 1969 to 1999.

`%z` A diferença horária do GMT no formato ISO 8601.

`%%` Um caractere `%` literal.

Todos os outros caracteres (ordinários) na cadeia de formato devem ter um caractere correspondente com a cadeia de entrada a ser convertida. Exceções são espaços na cadeia de

entrada, a qual pode coincidir com zero ou mais caracteres de espaço na cadeia de formato.

Se algum componente de data e/ou hora estão ausentes no formato e, portanto, na cadeia de entrada, a função `str2time` usa seus valores padrão correspondendo a 00:00:00 de 1º de Janeiro de 1970, ou seja, o valor padrão para o ano é 1970, o valor padrão para o mês é Janeiro, etc.

A função `str2time` é aplicável a todos horários calendário desde 00:00:00 de 1º de Janeiro de 0001 até 23:59:59 de 31 de Dezembro de 4000 do calendário Gregoriano.

B.3 Convertendo tempo de calendário a uma cadeia de caracteres

A função `time2str(`*t*`,` *f*`)` converte o tempo de calendário especificado pelo seu primeiro argumento *t*, que deve ser uma expressão numérica, para uma cadeia de caracteres (valor simbólico). A conversão é controlada pela cadeia de formato *f* (o segundo argumento), que deve ser uma expressão numérica.

O tempo de calendário passado para `time2str` possui o mesmo significado dos valores retornados pela função `gmtime` (ver Subseção B.1, página 58). Note que `time2str` *não corrige* o tempo de calendário especificado para zona horária local, i.e., o tempo de calendário 0 sempre corresponde a 00:00:00 de 1º de Janeiro de 1970.

Por exemplo, as sentenças de modelo:

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

pode produzir a seguinte impressão:

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

que é a impressão da data e hora no formato ISO.

A cadeia de formato passada para a função `time2str` consiste de especificadores de conversão e caracteres ordinários. Cada especificador de conversão começa com um caractere de porcentagem (`%`) seguido de uma letra.

Os seguintes especificadores de conversão podem ser usados na cadeia de formato:

`%a` O nome do dia da semana abreviado(2 caracteres).

`%A` O nome do dia da semana completo.

`%b` O nome do dia do mês abreviado (3 caracteres).

`%B` O nome do mês completo.

`%C` O século do ano, ou seja, o maior inteiro não maior que o ano dividido por 100.

`%d` O dia do mês como um número decimal (de 01 até 31).

`%D` A data usando o formato `%m/%d/%y`.

`%e` O dia do mês, como em `%d`, mas preenchido com espaço em branco ao invés de zero.

`%F` A data usando o formato `%Y-%m-%d`.

`%g` O ano correspondente ao número de semana ISO, mas sem o século (de 00 até 99). Este possui o mesmo formato e valor que `%y`, exceto que se o número de semana ISO (ver `%V`) pertence ao ano anterior ou seguinte, se usa aquele ano em seu lugar.

`%G` O ano correspondente ao número de semana ISO. Este possui o mesmo formato e valor que `%Y`, exceto que se o número de semana ISO (ver `%V`) pertence ao ano anterior ou seguinte, se usa aquele ano em seu lugar.

`%h` O mesmo que `%b`.

`%H` A hora como um número decimal usando um relógio 24 horas (de 00 até 23).

`%I` A hora como um número decimal usando um relógio 12 horas (de 01 até 12).

`%j` O dia do ano como um número decimal (de 001 até 366).

`%k` A hora como um número decimal usando um relógio 24 horas, como `%H`, mas preenchido com espaço em branco ao invés de zero.

`%l` A hora como um número decimal usando um relógio 12 horas, como `%I`, mas preenchido com espaço em branco ao invés de zero.

`%m` O mês como um número decimal (de 01 até 12).

`%M` O minuto como um número decimal (de 00 até 59).

`%p` Tanto `AM` como `PM`, de acordo com o valor da hora fornecido. Meia-noite é tratada como `AM` e meio-dia, como `PM`.

`%P` Tanto `am` como `pm`, de acordo com o valor da hora fornecido. Meia-noite é tratada como `am` e meio-dia, como `pm`.

`%R` A hora e minuto em números decimais usando o formato `%H:%M`.

`%S` O segundo como um número decimal (de 00 até 59).

`%T` A hora do dia em números decimais usando o formato `%H:%M:%S`.

`%u` O dia da semana como número decimal (de 1 até 7) em que Segunda é 1.

`%U` O número da semana do ano corrente como um número decimal (de 00 até 53) iniciando com o primeiro Domingo como o primeiro dia da primeira semana. Os dias que precedem o primeiro Domingo do ano são considerados parte da semana 00.

`%V` O número da semana ISO como um número decimal (de 01 até 53). Semanas ISO iniciam com Segunda e finalizam com Domingo. A semana 01 de um ano é a primeira semana que possui a maioria de seus dias naquele ano. Isto é equivalente à semana contendo 4 de Janeiro. A semana 01 de um ano pode conter dias do ano anterior. A semana anterior à semana 01 de um ano é a última semana (52 ou 53) do ano anterior, mesmo se ela contém dias do novo ano. Em outras palavras, se 1º de Janeiro é Segunda, Terça, Quarta ou Quinta, ele está na semana 01; Se 1º de Janeiro é Sexta, Sábado ou Domingo, ele está na semana 52 ou 53 do ano anterior.

`%w` O dia da semana como um número decimal (de 0 até 6) em que Domingo é 0.

`%W` O número da semana do ano corrente como um número decimal (de 00 até 53), iniciando com a primeira Segunda como o primeiro dia da primeira semana. Dias que precedem a primeira Segunda do ano são considerados parte da semana 00.

`%y` O ano sem o século como um número decimal (de 00 até 99), ou seja, o ano `mod` 100.

`%Y` O ano como um número decimal, usando o calendário Gregoriano.

`%%` Um caractere `%` literal.

Todos os outros caracteres (ordinários) na cadeia de formato são simplesmente copiados à cadeia resultante.

O primeiro argumento (tempo do calendário) passado para a função `time2str` deve estar entre −62135596800 até +64092211199, o que corresponde ao período de 00:00:00 de 1º de Janeiro de 0001 até 23:59:59 de 31 de Dezembro de 4000 do calendário Gregoriano.

Apêndice C

Controladores de tabelas

por Andrew Makhorin <mao@gnu.org>
e Heinrich Schuchardt <heinrich.schuchardt@gmx.de>

O *controlador de tabelas* é um módulo do programa que permite transmitir dados entre objetos de um modelo MathProg e tabela de dados.

Atualmente, o pacote GLPK possui quatro controladores de tabelas:

— controlador interno de tabelas CSV;

— controlador interno de tabelas xBASE;

— controlador de tabelas ODBC;

— controlador de tabelas MySQL.

C.1 Controlador de tabelas CSV

O controlador de tabelas CSV assume que a tabela de dados está representada na forma de arquivo de texto plano, em formato de arquivo CSV (valores serparados por vígula: comma-separated values) como descrito abaixo.

Para escolher o controlador de tabelas CSV, seu nome na sentença table deve ser especificado como `"CSV"` e o único argumento deve especificar o nome do arquivo de texto plano contendo a tabela. Por exemplo:

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

O sufixo do nome do arquivo pode ser arbitrário, no entanto, é recomendado usar o sufixo '`.csv`'.

Ao ler tabelas de entrada o controlador de tabelas CSV fornece um campo implícito chamado `RECNO`, que contém o número do registro corrente. Este campo pode ser especificado na sentença de entrada table, como se realmente houvesse um campo chamado `RECNO` no arquivo CSV. Por exemplo:

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

Formato CSV[1]

O formato CSV (*comma-separated values*) é um formato de arquivo de texto plano definido como segue.

1. Cada registro é localizado em uma linha separada, delimitada por uma quebra de linha. Por exemplo:

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

onde `\n` significa o caractere de controle `LF` (`0x0A`).

2. O último registro no arquivo pode ou não ter a quebra de linha. Por exemplo:

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

3. Deve haver uma linha de cabeçalho na primeira linha do arquivo no mesmo formato das linhas de registros normais. Este cabeçalho deve conter nomes correspondendo aos campos no arquivo. O número de nomes de campos na linha de cabeçalho deve ser o mesmo do número de campos dos registros do arquivo. Por exemplo:

```
nome1,nome2,nome3\n
aaa,bbb,ccc\n
xxx,yyy,zzz\n
```

4. Dentro do cabeçalho e de cada registro, podem haver um ou mais campos separados por vírgulas. Cada linha deve conter o mesmo número de campos por todos arquivo. Espaços são considerados parte de um campo, portanto, não são ignorados. O último campo do registro não deve ser seguido de vírgula. Por exemplo:

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

5. Campos podem ou não estar cercados em aspas duplas. Por exemplo:

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

6. Se um campo é cercado de aspas duplas, cada aspa dupla que faça parte do campo deve ser codificado duas vezes. Por exemplo:

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

Exemplo

```
DE,PARA,DISTANCIA,CUSTO
Seattle,New-York,2.5,0.12
Seattle,Chicago,1.7,0.08
Seattle,Topeka,1.8,0.09
San-Diego,New-York,2.5,0.15
San-Diego,Chicago,1.8,0.10
San-Diego,Topeka,1.4,0.07
```

[1] Este material é baseado no documento RFC 4180.

C.2 Controlador de tabelas xBASE

O controlador de tabelas xBASE assume que a tabela de dados é armazenada no formato de arquivo .dbf.

Para escolher o controlador de tabela xBASE, seu nome na sentença table deve ser especificado como `"xBASE"` e o primeiro argumento deve especificar o nome de um arquivo .dbf contendo a tabela. Para a tabela de saída deve haver um segundo argumento definindo o formato da tabela na forma `"FF...F"`, onde `F` é tanto `C(`n`)`, que especifica um campo de caractere de tamanho n, ou `N(`n`[,`p`])`, que especifica um campo numérico de tamanho n e precisão p (por padrão p é 0).

Adiante está um simples exemplo que ilustra a criação e leitura de um arquivo .dbf:

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

C.3 Controlador de tabelas ODBC

O controlador de tabelas ODBC permite conexões com bancos de dados SQL usando uma implementação da interface ODBC baseada na Call Level Interface (CLI).[2]

Debian GNU/Linux. No Debian GNU/Linux o controlador de tabelas ODBC usa o pacote iODBC, [3] que deve ser instalado antes de montar o pacote GLPK. A instalação pode ser efetuada com o seguinte comando:

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

Note que, ao configurar o pacote GLPK, para habilitar o uso da biblioteca do iODBC a opção '`--enable-odbc`' deve ser passada para o script de configuração.

Para seu uso em todo sistema, as bases de dados individuais devem ser inseridas em `/etc/odbc.ini` e `/etc/odbcinst.ini`. As conexões das bases de dados a serem usadas por um único usuário são especificadas por arquivos do diretório home (`.odbc.ini` e `.odbcinst.ini`).

Microsoft Windows. No Microsoft Windows o controlador de tabelas ODBC usa a biblioteca Microsoft ODBC. Para habilitar esta funcionalidade, o símbolo:

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

deve ser definido no arquivo de configuração do GLPK '`config.h`'.

Fontes de dados podem ser criados via Ferramentas Administrativas do Painel de Controle.

Para escolher do controlador de tabelas ODBC, seu nome na sentença table deve ser especificado

[2] A norma software correspondente é definida na ISO/IEC 9075-3:2003.

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

como `'ODBC'` ou `'iODBC'`.

A lista de argumentos é especificada como segue.

O primeiro argumento é a cadeia de conexão passada para a biblioteca ODBC, por exemplo:

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

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

Diferentes partes da cadeia são separadas por ponto e vírgula. Cada parte consiste de um par *nome-do-campo* e *valor* separados pelo sinar de igualdade. Os nomes de campo permitidos dependem da biblioteca ODBC. Tipicamente os seguintes nomes-de-campo são permitidos:

`DATABASE` base de dados;

`DRIVER` controlador ODBC;

`DSN` nome de uma fonte de dados;

`FILEDSN` nome de um arquivo de fonte de dados;

`PWD` senha de usuário;

`SERVER` base de dados;

`UID` nome de usuário.

O segundo argumento e todos os seguintes são considerados como sentenças SQL.

As sentenças SQL podem ser estendidas sobre múltiplos argumentos. Se o último caractere de um argumento é um ponto e vírgula, este indica o fim de uma sentença SQL.

Os argumentos de uma sentença SQL são concatenados separados por espaço. O eventual ponto e vírgula final será removido.

Todas as sentenças SQL, exceto a última, serão executadas diretamente.

Para tabela-IN, a última sentença SQL pode ser um comando SELECT que se inicia com `'SELECT '` em letras maiúsculas. Se a cadeia não se inicia com `'SELECT '`, se considera que é um nome de uma tabela e uma sentença SELECT é automaticamente gerada.

Para tabela-OUT, a última sentença SQL pode conter um ou múltiplos pontos de interrogação. Se contém um ponto de interrogação, é considerado um gabarito para a rotina de escrita. Caso contrário, a cadeia é considerada um nome de tabela e um gabarito INSERT é automaticamente gerado.

A rotina de escrita usa um gabarito com o pontos de interrogação e o substitui o primeiro ponto de interrogação pelo primeiro parâmetro de saída, o segundo ponto de interrogação, pelo segundo parâmetro e assim por diante. Em seguida, o comando SQL é emitido.

O que segue é um exemplo da sentença table de saída:

```
table ta { l in LOCAIS } OUT
   'ODBC'
   'DSN=glpkdb;UID=glpkuser;PWD=glpkpassword'
   'DROP TABLE IF EXISTS resultado;'
   'CREATE TABLE resultado ( ID INT, LOC VARCHAR(255), QUAN DOUBLE );'
```

```
   'INSERT INTO resultado 'VALUES ( 4, ?, ? )' :
   l ~ LOC, quantidade[l] ~ QUAN;
```

Alternativamente pode se escrever como segue:

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

O uso de gabaritos com '?' não só permite INSERT, como também o UPDATE, DELETE, etc. Por exemplo:

```
table ta { l in LOCAIS } OUT
   'ODBC'
   'DSN=glpkdb;UID=glpkuser;PWD=glpkpassword'
   'UPDATE resultado SET DATA = ' & data & ' WHERE ID = 4;'
   'UPDATE resultado SET QUAN = ? WHERE LOC = ? AND ID = 4' :
   quantidade[l], l;
```

C.4 Controlador de tabelas MySQL

O controlador de tabelas MySQL permite a conexão a base de dados MySQL.

Debian GNU/Linux. No Debian GNU/Linux o controlador de tabelas MySQL usa o pacote MySQL, [4] que deve ser instalado antes da criação do pacote GLPK. A instalação pode ser efetuada com o seguinte comando:

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

Note que ao configurar o pacote GLPK para habilitar o uso da biblioteca MySQL a opção '`--enable-mysql`' deve ser passada ao script de configuração.

Microsoft Windows. No Microsoft Windows o controlador de tabelas MySQL também usa a biblioteca MySQL. Para habilitar esta funcionalidade o símbolo:

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

deve ser definido no arquivo de configuração do GLPK '`config.h`'.

Para escolher o controlador de tabelas MySQL, seu nome na sentença table deve ser especificada como `'MySQL'`.

A lista de argumentos é especificada como segue.

O primeiro argumento especifica como conectar a base de dados no estilo DSN, por exemplo:

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

[4] Para fazer o download de arquivos de desenvolvimento, ver `<http://dev.mysql.com/downloads/mysql/>`.

Diferentes partes da cadeia são separadas por ponto e vírgula. Cada parte consiste de um par *nome-do-campo* e *valor* separado pelo sinal de igualdade. Os seguintes nomes de campo são permitidos:

`Server` servidor rodando a base de dados (localhost por padrão);

`Database` nome da base de dados;

`UID` nome de usuário;

`PWD` senha de usuário;

`Port` porta usada pelo servidor (3306 por padrão).

O segundo argumento e todos os seguintes são considerados sentenças SQL.

Sentenças SQL podem se estender sobre múltiplos argumentos. Se o último caractere de um argumento é um ponto e vírgula, isto indica o fim de uma sentença SQL.

Os argumentos de uma sentença SQL são concatenados e separados por espaço. O eventual ponto e vírgula final será removido.

Todas sentenças SQL, menos a última, serão executadas diretamente.

Para tabela-IN, a última sentença SQL pode ser um comando SELECT iniciado com letras maiúsculas `'SELECT '`. Se a cadeia não inicia com `'SELECT '` é considerado um nome de tabela e a sentença SELECT é automaticamente gerada.

Para tabela-OUT, a última sentença SQL pode conter um ou múltiplos pontos de interrogação. Se contém um ponto de interrogação, é considerado um gabarito para a rotina de escrita. Caso contrário, a cadeia é considerada um nome de tabela e um gabarito INSERT é automaticamente gerado.

A rotina de escrita usa um gabarito com o pontos de interrogação e o substitui o primeiro ponto de interrogação pelo primeiro parâmetro de saída, o segundo ponto de interrogação, pelo segundo parâmetro e assim por diante. Em seguida, o comando SQL é emitido.

O que segue é um exemplo da sentença table de saída:

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

Alternativamente poderia ser escrito como segue:

```
table ta { l in LOCAIS } OUT
   'MySQL'
   'Database=glpkdb;UID=glpkuser;PWD=glpkpassword'
   'DROP TABLE IF EXISTS resultado;'
   'CREATE TABLE resultado ( ID INT, LOC VARCHAR(255), QUAN DOUBLE );'
   'resultado' :
```

```
   l ~ LOC, quantidade[l] ~ QUAN, 4 ~ ID;
```

O uso de gabaritos com '?' não só permite INSERT, como também o UPDATE, DELETE, etc. Por exemplo:

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

Apêndice D

Resolvendo modelos com glpsol

O pacote GLPK [1] inclui o programa `glpsol`, um solver de PL/PIM autônomo. Este programa pode ser chamado por uma linha de comando ou pelo shell para resolver modelos escritos na linguagem de modelagem GNU MathProg.

Para comunicar ao solver que o arquivo de entrada contém uma descrição do modelo, é necessário especificar a opção `--model` na linha de comando. Por exemplo:

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

Às vezes é necessário usar a seção de dados posicionado em um arquivo separado. Neste caso, deve-se usar o seguinte comando:

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

Note que ser o arquivo de modelo também contém a seção de dados, aquela seção é ignorada.

Também é permitido especificar mais de um arquivo contendo a seção de dados, por exemplo:

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

Se a descrição do modelo contém alguma sentença display e/ou printf, o resultado é enviado para o terminal por padrão. Se há a necessidade de redirecionar a saída para um arquivo, deve-se usar o seguinte comando:

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

Se há a necessidade de ver o problema que está sendo gerado pelo tradutor de modelo, deve-ser usar a opção `--wlp` como segue:

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

Neste caso, os dados do problema são escritos no arquivo `foo.lp` no formato CPLEX LP viável para análise visual.

Ás vezes, é necessário checar a descrição do modelo sem ter que resolver a instância do problema gerado. Neste caso, deve-se especificar a opção `--check`, por exemplo:

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

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

Se há a necessidade de escrever uma solução numérica obtida pelo solver para um arquivo, deve-se usar o seguinte comando:

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

neste caso, a solução é escrita no arquivo `foo.sol` em formato de texto plano, viável para análise visual.

A lista completa de opções do `glpsol` pode ser encontrada no manual de referência do GLPK incluída na distribuição do GLPK.

Apêndice E

Exemplo de descrição de modelo

E.1 Descrição de modelo escrito em MathProg

Este é um exemplo completo de descrição de modelo escrito na linguagem de modelagem GNU MathProg.

```
# UM PROBLEMA DE TRANSPORTE
#
# Este problema encontra a logística de custo mínimo que atende das demandas
# de mercado e as ofertas das fábricas.
#
#  Referência:
#              Dantzig G B, "Linear Programming and Extensions."
#              Princeton University Press, Princeton, New Jersey, 1963,
#              Chapter 3-3.

set I;
/* fábricas de enlatados*/

set J;
/* mercados */

param a{i in I};
/* capacidade da fábrica i, em caixas */

param b{j in J};
/* demanda no mercado j, em caixas */

param d{i in I, j in J};
/* distância, em milhares de milhas */
```

```
param f;
/* frete, em dólares por caixa a cada mil milhas */

param c{i in I, j in J} := f * d[i,j] / 1000;
/* custo de transporte, em milhares de dólares por caixa */

var x{i in I, j in J} >= 0;
/* quantidade enviada, em caixas */

minimize custo: sum{i in I, j in J} c[i,j] * x[i,j];
/* custo total de transporte, em milhares de dólares */

s.t. suprimento{i in I}: sum{j in J} x[i,j] <= a[i];
/* observa o limite de suprimento na fábrica i */

s.t. demanda{j in J}: sum{i in I} x[i,j] >= b[j];
/* satisfaz a demanda do mercado j */

data;

set I := Seattle San-Diego;

set J := New-York Chicago Topeka;

param a := Seattle     350
           San-Diego   600;

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

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

param f := 90;

end;
```

E.2 Instância gerada do problema de PL

Este é o resultado da tradução do modelo de exemplo produzido pelo solver `glpsol` e escrito no formato CPLEX LP com a opção `--wlp`.

```
\* Problem: transporte *\

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

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

End
```

E.3 solução ótima do problema de PL

Esta é a solução ótima da instância gerada do problema de PL encontrada pelo solver `glpsol` e escrita em formato de texto plano com a opção`--output`.

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

   No.   Row name   St   Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------ ------------- ------------- -------------
     1 cust         B        153.675
     2 suprimento[Seattle]
                    NU           350                         350         < eps
     3 suprimento[San-Diego]
                    B            550                         600
     4 demanda[New-York]
                    NL           325           325                       0.225
     5 demanda[Chicago]
                    NL           300           300                       0.153
     6 demanda[Topeka]
                    NL           275           275                       0.126
```

```
   No. Column name  St   Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------ ------------- ------------- -------------
     1 x[Seattle,New-York]
                    B             50             0
     2 x[Seattle,Chicago]
                    B            300             0
     3 x[Seattle,Topeka]
                    NL             0             0                       0.036
     4 x[San-Diego,New-York]
                    B            275             0
     5 x[San-Diego,Chicago]
                    NL             0             0                       0.009
     6 x[San-Diego,Topeka]
                    B            275             0

End of output
```

Agradecimentos

Os autores gostariam de agradecer as seguintes pessoas que gentilmente leram, comentaram e corrigiram o rascunho deste documento:

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

Harley Mackenzie <hjm@bigpond.com>

Robbie Morrison <robbie@actrix.co.nz>