Introduction au langage de modélisation GNU MathProg (GLPK)

Romain Apparigliato

GDF-Suez, Direction de la Recherche et de l'Innovation
Pôle Simulation et Optimisation
361 av. du Président Wilson, 93211 St Denis La Plaine

romain.apparigliato@polytechnique.edu

Table des matières

1 Présentation

$i\backslash j$ Le "GNU Linear Programming Kit" (GLPK) est un outil performant pour résoudre des problèmes d'optimisation linéaire de variables continues ou mixtes (entières et continues). Ce kit est composé d'un langage de modélisation **GNU MathProg** et d'une librairie de fonctions C (GLPK) utilisant le solveur **Glpsol**. L'extrême avantage de ce kit est d'être en libre accès et relativement facile à installer et à utiliser.
Ce document est principalement consacré à la présentation du langage de modélisation et a pour objectif d'en montrer les bases pour pouvoir modéliser, résoudre un problème d'optimisation et être capable d'utiliser d'autres outils plus puissants (AMPL,...) au langage très proche.

1.1 GNU Linear Programming Kit

GLPK est une librairie de fonctions C qui utilisent les principales méthodes de résolution de problème d'optimisation (simplex, branch and bound, méthodes de points intérieurs primal-dual, ...) ainsi que tous les outils nécessaires aux créations d'objets, à l'appel des solveurs, à l'affichage des résultats, ...

GLPK n'est pas un programme puisqu'il ne peut pas être lancé et ne possède pas de *main*(). Le programme appelant cette librairie doit donc être conçu et utiliser les fonctions adéquates. Tout le descriptif des fonctions disponibles est inscrit dans le manuel d'utilisation fournit lors du téléchargement. GLPK a un solveur par défaut, glpsol, qui résout le problème généré.

1.2 GNU MathProg

L'utilisation directe des routines GLPK, tout comme l'utilisation d'un solveur comme CPLEX, EXPRESS, MATLAB, ..., n'est pas forcément évidente. Cela nécessite d'avoir des bases en programmation, de maîtriser les outils à utiliser et également d'effectuer des transformations préalables à la formulation du problème. Pour remédier à ce problème, il existe des langages de modélisation. Avec un tel outil, l'utilisateur n'a plus qu'à créer un fichier de données, définir son problème avec une écriture très proche de l'écriture mathématique et le langage de modélisation se charge de la transformation du problème et du lien au solveur. Ainsi, avec quelques bases de programmation, on peut facilement modéliser le problème et le résoudre.

Le langage de modélisation GNU MathProg est un sous-ensemble du langage de modélisation AMPL. AMPL est un langage de modélisation très puissant que l'on peut coupler avec divers solveurs comme CPLEX, EXPRESS, MOSEK,.... La limitation aux problèmes linéaires (PL) de GLPK étant due au solveur utilisé, on se rend compte qu'AMPL, couplé au solveur adéquat, peut être facilement utilisé à des problèmes plus complexes (optimisation non linéaire, optimisation conique, ...). Pour de plus amples informations sur AMPL, un site web est disponible (http://www.ampl.com) sur lequel on peut télécharger une version étudiante gratuite (couplée à une version allégée de CPLEX) qui ne permet de résoudre que des problèmes de moins de 300

variables et 300 contraintes. Cependant, une bonne connaissance d'AMPL permet d'utiliser facilement GLPK et vice-versa (ainsi que bien d'autres langages de modélisation).

1.3 Téléchargement et Installation

GLPK en version UNIX est disponible sur le site http://www.gnu.org/software/glpk/. Lors du téléchargement est fournie une liste de documents donc une doc de référence à partir de laquelle est construit cette introduction. Une version WINDOWS est également disponible à l'adresse http://gnuwin32.sourceforge.net/packages/glpk.htm.

2 Informations préliminaires

2.1 Motivations

L'objectif de ces cours est de parvenir à modéliser des problèmes d'optimisation linéaire purs ou mixtes (avec des variables entières) de la forme :

$$\begin{aligned} \min_x(\max_x) \quad & c_1x_1 + c_2x_2 + \ldots + c_nx_n \\ s.c. \quad & \\ & \ell_1 \leq a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \leq u_1 \\ & \ell_2 \leq a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \leq u_2 \\ & \ldots\ldots\ldots \\ & \ell_m \leq a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n \leq u_m \end{aligned}$$

Les bornes ℓ_i (resp. u_i) peuvent valoir $+\infty$ (resp $-\infty$). Dans le cas d'un problème linéaire pur, les variables x_i sont réelles. Dans le cas MIP, certains x_i peuvent être entières. Les quantités c_i, a_{ij}, ℓ_i, u_i sont des données fournies par l'utilisateur.

2.2 Principe général

La construction d'un modèle est effectuée à partir de briques élémentaires que l'on appelle objets de modélisation. Pour la phase de construction, il en existe 5 types : *parameter*, *set*, *variable*, *Constraint* et *Objective*. Les objets *Parameter* et *Set* permettent de définir toutes les données du problème, *Variable* de définir toutes les variables du problème, *Constraint* de définir les contraintes du problème et enfin *Objective* la fonction objectif du problème.

Chaque brique est composée d'expressions, construites selon des règles que l'on définira plus tard. L'ensemble des possibilités de construction est assez large du moment que le problème demeure linéaire.

2.3 Structuration

La modélisation d'un problème d'optimisation se divise en deux parties :
- La **section Modèle** contient toutes les déclarations, les paramètres calculables et les définitions des contraintes et de l'objectif.
- La **section Données** contient toutes les données fixes (valeurs des paramètres, du contenu des ensembles).

Les deux sections peuvent être déclarées :

- Dans un même fichier composé comme suit :

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

Il est obligatoire de séparer les deux parties en plaçant la partie de données entre `data;` et `end;`. Le fichier devra être sauvegardé avec l'extension `.mod`.

- Dans 2 fichiers séparés :

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

Model file

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

Data file

Dans ce cas, le modèle doit être sauvé avec l'extension `.mod` et les données avec l'extension `.dat`. Le fichier *mod* joue le rôle d'un programme principal en programmation. Du coup, lorsque l'on définit une quantité dans le fichier *dat*, il faut bien signaler au programme principal l'existence de cette donnée. Il faut donc déclarer dans le *mod* toutes les quantités définies dans le *dat*.

En pratique, il est fortement conseillé de séparer l'implémentation modèle et l'attribution des données en 2 fichiers distincts afin d'avoir un modèle générique, pouvant fonctionner pour tout jeu de données cohérent.

Exemple : On peut définir une constante $T = 5$ dans le fichier *dat* avec : `param T:=5;` Il faut déclarer cette constante dans le fichier mod en incluant la ligne : `param T;`

2.4 Compilation

2.4.1 Commandes de base

Supposons dans un premier temps que tout est incorporé dans un fichier `modele.mod`. Dans une fenêtre de commande, on se place dans le répertoire où est situé le fichier. L'appel au solveur est effectué à l'aide de la commande :

```
glpsol --model modele.mod
```

Dans le cas général (et conseillé !) où les données et le modèle sont séparés en 2 fichiers `modele.mod` et `donnees.dat`, l'exécution s'effectue avec :

```
glpsol --model modele.mod --data donnees.dat
```

2.4.2 Options

En fait, la formulation générale de l'exécution est :

```
glpsol --options --model modele.mod --data donnees.dat
```

où `--options` est une liste d'options particulières. La liste peut être obtenue facilement à l'aide de la commande :

```
glpsol --help
```

Ces options peuvent être soient des options générales soit des options spécifiques aux algorithmes de résolution. Les principales options générales sont :

Forme générale	**Rôle**
`--display` *filename*	Ecrit tous les affichages du modèle dans un fichier *filename*
`--output` *filename*	Ecrit la solution du problème dans le fichier *filename*
`--tmlim` *nnn*	Limite l'exécution à nnn secondes
`--memlim` *nnn*	Limite la taille mémoire disponible à nnn Mo
`--check`	Ne résoud pas le problème, vérifie juste la cohérence du modèle et des données
`--simplex`	Utilise la méthode du Simplexe (défaut)
`--interior`	Utilise la méthode des Points Intérieurs
`--glp` (resp. `--wglp` *filename*)	Lire (resp. écrire dans *filename*) le problème LP/MIP dans le format GNU LP
`--mps` (resp. `--wmps` *filename*)	Lire (resp. écrire dans *filename*) le problème LP/MIP dans le format MPS
`--freemps` (resp. `--wfreemps` *filename*)	Lire (resp. écrire dans *filename*) le problème LP/MIP dans le format free MPS
`--cpxlp` (resp. `--wcpxlp` *filename*)	Lire (resp. écrire dans *filename*) le problème LP/MIP dans le format CPLEX
`--math`	Lire le problème LP/MIP dans le format GNU MathProg
`--wtxt` *filename*	Ecrire le problème dans *filename* dans un format plain

TAB. 1 – Principales options disponibles pour la compilation de glpsol

D'autres sont disponibles pour la résolution avec le simplexe et pour la résolution MIP (cf. `glpsol --help`).

3 Règles de codage du modèle

La description du modèle est codée en utilisant l'ensemble des caractères ASCII :
– caractères alphabétiques : A B C ... Z a b c ... z (case sensitive) ;
– caractères numériques : 0 1 2 ... 9 ;
– caractères spéciaux : ! " # & ' ( ) * + , - . / : ; < > = [ ] ^ { | }
Les espaces ne sont effectifs qu'à l'intérieur des commentaires et des chaînes de caractères. Ils peuvent être utilisés librement dans le modèle pour améliorer la lisibilité du code.

Les différents outils syntaxiques autorisés pour coder sont :

3.1 Noms symboliques

Les noms symboliques sont constitués de caractères alphabétiques et numériques, le premier étant forcément alphabétique. Tous les noms symboliques doivent être uniques.

Ils sont choisis par l'utilisateur et utilisés pour identifier des objets : paramètres, variables, contraintes, objectif, ensembles.

Exemples :

```
Nombre_Pas_de_Temps, param_J1, _P123_abc_321, ...
```

3.2 Les nombres

Les nombres sont écrits de la forme $xx(Esyy)$, où xx est un nombre réel avec éventuellement une partie décimale séparée par un point, s le signe + ou −, yy un exposant entier. La lettre E peut aussi être écrite e.

Les nombres sont évidemment utilisés pour représenter des quantités numériques et faire des opérations.

Exemples :

```
1234, 3.1415,56.E+5,.784,123.456e-7,...
```

3.3 Les chaînes de caractères

Les chaînes de caratères sont utilisées pour représenter des quantités symboliques et pour effectuer des affichages. La chaîne de caractères est en fait une séquence de caractères enfermés par ' ou " (les 2 formes sont équivalentes). Pour introduire ces mêmes caractères dans une chaîne de

caractères, il faut les doubler.

Exemples :

```
'Ceci est un super cours', "Ceci est un super cours", 'C''est un super cours',
"Elle dit: ""Non"" ", '1+2=3', ...
```

3.4 Mots clés

Les mots clés sont des séquences de caractères alphabétiques (et éventuellement numériques) qui sont reconnus directement et ont un sens pour GLPK. On peut distinguer 2 classes : les mots réservés, ayant une signification fixe, qui ne peuvent pas être utilisés pour définir un nom symbolique et les mots non réservés qui peuvent l'être et qui sont reconnus selon le contexte.

Les mots réservés sont :

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

Les mots non réservés seront présentés dans les sections suivantes.

3.5 Délimiteurs

Les délimiteurs sont des caractères spéciaux simples ou des séquences de 2 caractères spéciaux (dans ce cas, il ne doit pas y avoir d'espace entre les 2) :

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

Chaque délimiteur a une définition fixe qui sera présentée lors de l'utilisation de chacun.

3.6 Commentaires

Il est possible d'introduire dans le modèle des commentaires qui seront ignorés par GLPK et qui permettent de rendre le code plus lisible ou plus compréhensible. Les commentaires peuvent être sur une ligne et dans ce cas commencent après le caractère spécial # et finissent à la fin de la ligne. Ils peuvent être sur plusieurs lignes et sont inscrits entre /* et */. N'importe quoi peut être inscrit dans ces commentaires.

Exemples :

```
var a := 4; #Définition de la variable a
var b := 5; /* Définition de
la variable b */
```

4 Expressions

Les expressions servent à obtenir des résultats utiles à la définition d'un objet de modélisation. La forme générale d'une expression est :

expression_primaire operateur expression_primaire ... expression_primaire

On distingue plusieurs types d'expressions :

4.1 Expressions numériques

Une expression numérique est une règle pour définir ou calculer une valeur numérique. Elle est généralement constituée de 2 ou plusieurs expressions numériques primaires, utilisant des opérateurs arithmétiques.

Exemples :

```
2*j
sum{j in 1..5} a[j]*x[j] + 5*c
```

Voici les différents types d'expressions primaires :

- **Nombre** : Un nombre défini comme dans (3.2).

  Exemple :
  ```
  2*0.5
  ```

- **Constantes** : On peut définir des constantes représentées par des noms symboliques et leur associer une valeur numérique.

  Exemple :
  ```
  param j:=3;
  2*j existe et vaut 6
  ```

- **Paramètres multi-dimensionnels** : On peut définir des paramètres ou des variables représentés par des noms symboliques à multiples dimensions. La syntaxe générale de $a_{i_1,i_2,i_3,\ldots,i_n}$ est :

  $$a[i_1, i_2, i_3, \ldots, i_n].$$

  Chaque indice i_1, i_2, ..., i_n peut être une expression numérique ou une expression symbolique. Il faut être particulièrement vigilant aux tailles.

  Exemples :

```
param A := 1 1 2 3 3 8 4 6 ;
# A[3]*3 existe et vaut 24

param B : 1 2 3 4 :=
        1 2 3 4 5
        2 5 4 9 3
        3 4 5 9 2;

# B[2,4]*B[1,3]*2 existe et vaut 3*4*2=24

param C : PAR  TOU  MAR :=
      PAR  0   700  850
      TOU  700  0   500
      MAR  850 500  0;

# C['PAR','TOU']+C['MAR','TOU'] existe et vaut 700+500=1 200
```

- **Fonctions prédéfinies** :

Fonction GLPK	Rôle
`abs(x)`	Valeur absolue
`atan(x)`	$\arctan x$(radians)
`atan(y,x)`	$\arctan y/x$(radians)
`ceil(x)`	Plus petit entier proche de x
`cos(x)`	$\cos(x)$
`floor(x)`	Plus grand entier proche de x
`exp(x)`	$\exp(x)$
`log(x)`	$\log(x)$
`log10(x)`	$\log_{10}(x)$
`max(x1,x2,...,xn)`	Maximum des valeurs x1,x2,...,xn
`min(x1,x2,...,xn)`	Minimum des valeurs x1,x2,...,xn
`round(x)`	Arrondit x à l'entier le plus proche
`round(x,n)`	Arrondit x avec n décimales
`sin(x)`	$\sin(x)$
`sqrt(x)`	$\sqrt{x}$
`trunc(x)`	Tronque x à l'entier le plus proche
`trunc(x,n)`	Tronque x avec n décimales
`Irand224()`	Retourne un nombre pseudo-aléatoire dans $[0, 2^{24})$ selon une loi uniforme entière
`Uniform01()`	Retourne un nombre pseudo-aléatoire dans $[0, 1)$ selon une loi uniforme
`Uniform(a,b)`	Retourne un nombre pseudo-aléatoire dans $[a, b)$ selon une loi uniforme
`Normal01()`	Loi normale avec moyenne nulle et variance 1
`Normal(m,s)`	Loi normale avec moyenne m et variance s

TAB. 2 – Liste des fonctions disponibles sous GLPK

Tous les arguments des fonctions pré-définies doivent être des expressions numériques. On

peut également créer sa propre fonction en définissant une variable qui se calcule de la façon souhaitée.

Exemple :
```
var fraction {x in X, y in Y} := x/y;
```

On a défini une fonction qui calcule la fraction $\frac{x}{y}$ par l'appel `fraction[x,y]`. On verra ultérieurement comment définir les ensembles X et Y.

- **Expressions itérées** : Il existe 4 opérateurs d'itération qui peuvent être utilisés dans une expression numérique :

Opérateur GLPK		Rôle
`sum{`*ensemble*`}`	sommation :	$\sum_{(i_1,i_2,\ldots,i_n)\in\Delta} x(i_1,\ldots,i_n)$
`prod{`*ensemble*`}`	produit :	$\prod_{(i_1,i_2,\ldots,i_n)\in\Delta} x(i_1,\ldots,i_n)$
`min{`*ensemble*`}`	min :	$\min_{(i_1,i_2,\ldots,i_n)\in\Delta} x(i_1,\ldots,i_n)$
`max{`*ensemble*`}`	max :	$\max_{(i_1,i_2,\ldots,i_n)\in\Delta} x(i_1,\ldots,i_n)$

TAB. 3 – Liste des opérateurs d'itération disponibles dans GLPK

Dans les `{..}` est indiqué l'ensemble sur lequel l'opération s'effectue. La section(4.4) décrit tous les types d'ensembles pouvant être créés.

Exemples :
```
sum{i in 1..5} 2*i; // Retourne 2+4+6+8+10=30
sum{i in 1..J} log(x[i]); // Retourne log(x[1])+log(x[2])+...+log(x[J])
sum{i in 1..2, j in 1..2} x[i,j]; // Retourne x[1,1]+x[1,2]+x[2,1]+x[2,2]
```

- **Expressions conditionnelles** : Il existe 2 syntaxes pour l'expression numérique conditionnelle :

`if` *condition* `then` *solution*1 `else` *solution*2

`if` *condition* `then` *solution*1

C'est la traduction du *Si ... alors ... sinon*. La *condition* est une expression logique (section 4.5) et les *solution*1 et *solution*2 sont des expressions numériques. Dans l'expression complète, la valeur de l'expression conditionnelle est *solution*1 si le test est vrai, sinon elle vaut *solution*2. De même avec la forme réduite avec *solution*2 valant 0.

Exemple :
```
2*(if x>1 then 3 else 2); // Cette expression vaut 6 si x>1 ou 4 si x<= 1
```

Les **opérateurs arithmétiques** suivants peuvent être utilisés dans des expressions numériques :

Opérateur GLPK	Rôle
`+` x	"Plus 1" : Effectue $x + 1$
`-` x	"Moins 1" : Effectue $x - 1$
x `+` y	Addition
x `-` y	Soustraction
x `less` y	Différence positive (si $x < y$ alors 0 sinon $x - y$)
x `*` y	Multiplication
x `/` y	Division
x `div` y	Quotient de la division exacte
x `mod` y	Reste de la division exacte
x `**` y, x `^` y	Puissance

TAB. 4 – Liste des opérateurs arithmétiques disponibles dans GLPK

x et y sont des expressions numériques primaires. Si l'expression inclue plus qu'un opérateur arithmétique, tous les opérateurs sont calculés de la gauche vers la droite (sauf pour la puissance qui est calculée de la droite vers la gauche) selon la hiérarchie des opérations présentée juste après.

Voici la hiérarchie des opérations dans une expression numérique :

Opération	Hiérarchie
Evaluation des fonctions (`abs`, `log`,...)	1^{er}
Puissance (`**`,`^`)	2^{eme}
"Plus 1", "Moins 1" (`+`,`-`)	3^{eme}
Multiplication et division (`*`, `/`, `div`, `mod`)	4^{eme}
Opérations itérées (`sum`, `prod`, `min`, `max`)	5^{eme}
Addition et soustraction (`+`,`-`,`diff`)	6^{eme}
Evaluation conditionnelle (`if ... then ... else`)	7^{eme}

TAB. 5 – Hiérarchie des opérations dans une expression numérique dans GLPK

A noter que toute opération entre parenthèses est effectuée en priorité.

4.2 Expressions symboliques

L'expression symbolique sert à effectuer des opérations sur les chaînes de caractères. L'expression symbolique primaire peut être une chaîne de caractère, un indice, une expression symbolique conditionnelle,... Il est accepté d'utiliser une expression numérique en tant qu'expression

numérique. Dans ce cas, la valeur résultante est automatiquement convertie en symbolique.

Il n'existe qu'un seul opérateur symbolique dans GLPK, l'opérateur de concaténation `&` :

$$x\&y$$

où x et y sont des expressions symboliques. L'opérateur concatène les 2 opérandes (qui sont des chaînes de caractères) en une seule chaîne.

Exemple :

```
'from' & city[i] & 'to' & city[j]
```

La hiérarchie générale des opérations devient ainsi :

Opération	Hiérarchie
Evaluation des opérations numériques	$1^{er} - 7^{eme}$
Concaténation	8^{eme}
Evaluation conditionnelle (`if ... then ... else`)	9^{eme}

TAB. 6 – Hiérarchie des opérations dans GLPK

4.3 Expressions sur les indices

L'expression sur les indices est une construction qui permet d'indiquer l'ensemble de définition d'une variable, d'une contrainte ou spécifier un ensemble notamment pour les opérateurs d'itération. Elle a 2 formes syntaxiques possibles :

$$\{entry_1, entry_2, \ldots, entry_m\}$$
$$\{entry_1, entry_2, \ldots, entry_m : predicat\}$$

où $entry_i$ est une expressions précisant le domaine d'appartenance de l'indice i et *predicat* est une expression logique (section 4.5) qui spécifie des conditions sur les indices. L'indice n'est pas obligatoire s'il n'est pas utilisé dans la définition de l'objet (comme dans le cas d'une déclaration). Le *predicat* est forcément à la fin mais peut dépendre de tous les indices.

Exemple :

```
sum{i in 1..3, j in 1..3 : i==j } x[i,j];
                                  // Retourne x[1,1]+x[2,2]+x[3,3]
sum{i in 1..3, j in 1..3 : i!=j && i!=1 } x[i,j];
                                  // Retourne x[2,1]+x[2,3]+x[3,1]+x[3,2]
```

Quelques règles :

- L'écriture de chaque $entry_i$ peut avoir une des 3 formes suivantes :

$$t \texttt{ in } S$$
$$(t_1, t_2, \ldots, t_k) \texttt{ in } S$$
$$S$$

où $t_1, t_2, \ldots, t_k$ sont des indices et S un ensemble quelconque (défini en 4.4).

Exemple :
```
j in 1..5
(1,'toto') in B
var A{1..J}
```

Evidemment, il faut bien faire attention aux dimensions : le nombre d'indices doit être le même que la dimension de l'ensemble S.

- Quand on écrit une expression de la forme `objet`{i `in` S }, la variable i est une variable locale à l'objet. En dehors de cet objet, la variable i n'existe plus. D'autre part, elle est auto-déclarée et ne nécessite pas une déclaration générale comme pour tous les paramètres et toutes les variables.

Exemple :
```
prod{j in 1..3} x[j]; //Retourne x[1]*x[2]*x[3]
j*2 // Erreur à la compilation. j n'existe pas.
```

On ne peut pas prendre comme indice des noms de variables ou de paramètres déjà existants. Par contre, on peut utiliser dans différents objets les mêmes indices.

Exemple : On peut mettre dans un même code les expressions suivantes sans aucune confusion. Chaque j n'existe que dans son objet `prod` et `sum` (les j sont donc distincts) :
```
prod{j in 1..3:j!=2} x[j];
sum{j in 1..3:j!=1} x[j];
```

- Supposons les 3 ensembles :

$$\begin{aligned} A &= \{4, 7\} \\ B &= \{(1, Jan), (1, Fev), (2, Mar)\} \\ C &= \{a, b, c\} \end{aligned}$$

Si on considère l'expression définie en GLPK,

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

on peut la traduire algorithmiquement par :

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

ce qui donne concrètement la succession d'actions :

$$\begin{array}{llll}
i=4 & j=1 & k=Jan & l=a : action \\
i=4 & j=1 & k=Jan & l=b : action \\
i=4 & j=1 & k=Jan & l=c : action \\
i=4 & j=1 & k=Fev & l=a : action \\
i=4 & j=1 & k=Fev & l=b : action \\
i=4 & j=1 & k=Fev & l=c : action \\
i=4 & j=2 & k=Mar & l=a : action \\
i=4 & j=2 & k=Mar & l=b : action \\
i=4 & j=2 & k=Mar & l=c : action \\
i=7 & j=1 & k=Jan & l=a : action \\
i=7 & j=1 & k=Jan & l=b : action \\
i=7 & j=1 & k=Jan & l=c : action \\
i=7 & j=1 & k=Fev & l=a : action \\
i=7 & j=1 & k=Fev & l=b : action \\
i=7 & j=1 & k=Fev & l=c : action \\
i=7 & j=2 & k=Mar & l=a : action \\
i=7 & j=2 & k=Mar & l=b : action \\
i=7 & j=2 & k=Mar & l=c : action
\end{array}$$

Exemple : Si on suppose la variable p définie par :

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

on peut définir l'expression numérique $\sum_{i \in A,(j,k) \in B,l \in C} (p_{ijkl})^2$ en utilisant l'opérateur d'itération `sum` :

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

- Dans une expression sur les indices, le nombre de composants du n-uplets résultant est le même que le nombre de variables locales déclarées.

  Exemple : Supposons l'expression suivante :

```
{i in A, (i-3,k) in B, l in C }
```

Les seules variables locales déclarées comme indices sont i, k et l. $i-3$ est une expression numérique.
On peut traduire cette expression par l'algorithme :

for all $i \in A$ **do**
  for all $(j,k) \in B$ **and** $j = i-3$ **do**
    for all $l \in C$ **do**
      action;

Le résultat est :

$$\{(4, Jan, a), (4, Jan, b), (4, Jan, c), (4, Fev, a), (4, Fev, b), (4, Fev, c)\}.$$

On n'a donc pas un 4-uplets mais seulement un triplet.

- Le *predicat* est une expression logique qui donne des conditions sur l'ensemble. Cette expression peut être aussi complexe que l'on souhaite, portée sur les indices que l'on souhaite mais doit forcément être située en dernière position.

  Exemple : Supposons l'expression suivante :

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

  On peut traduire cette expression par l'algorithme :

for all $i \in A$ **do**
  for all $(j,k) \in B$ **and** $j = i-3$ **do**
    for all $l \in C$ **do**
      if $i \leq 5$ **and** $k \neq$ 'Mar' **then**
        action;

  Le résultat est :

$$\{(4, 1, Jan, a), (4, 1, Jan, b), (4, 1, Jan, c), (4, 1, Fev, a), (4, 1, Fev, b), (4, 1, Fev, c)\}.$$

4.4 Expressions ensemblistes

Une expression ensembliste est une règle pour calculer un ensemble, c'est à dire une collection de n-uplets où les composantes de ces n-uplets sont des quantités numérique et/ou symbolique. Les différents types d'expression ensembliste primaire sont des ensembles pouvant être formulées par :

- **Une formulation littérale** : Elle peut être de deux formes selon les dimensions :

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

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

  Tous les n-uplets doivent avoir la même dimension. Tous les e_i ou e_{ij} peuvent être des expressions numérique ou symbolique.

  Exemple : `{(123,'aaa'), (i+1,'bbb'), (j-1,'ccc')}`

- **Ensembles indicés ou non** : L'expression primaire peut être un ensemble simple (ex : `I`) ou le contenu d'un ensemble à plusieurs dimensions (ex : S[i-1,j+1]).

- **Ensemble Arithmétique** : L'expression primaire correspondante peut être écrite sous les formes :

$$t_0\texttt{..}t_f\ \texttt{by}\ \delta_t$$

$$t_0\texttt{..}t_f$$

  où t_0, t_f et δ_t sont des expressions numériques. La deuxième expression est équivalente à la première avec $\delta_t = 1$.
  L'ensemble résultant est :

$$\{t_0, t_0 + \delta_t, t_0 + 2\delta_t, \ldots, t_f\}$$

  Exemples :
  `1..5` crée l'ensemble $\{1, 2, 3, 4, 5\}$.
  `1..5 by 2` crée l'ensemble $\{1, 3, 5\}$.

- **Expression itérée** : On peut créer des ensembles à l'aide d'opérations itérées. Pour cela, on utilise la syntaxe :

`setof` *indexing_expression integrand*

  où *indexing_expression* est un ensemble d'indices indiquant le nombre de répétitions à effectuer et *integrand* est soit un nombre, soit une expression symbolique, soit une liste de nombres ou d'expressions symboliques séparée de virgules et enfermées entre parenthèses.

  Exemple :
  `setof{i in 1..2, j in 1..3} (i,j+1)` crée l'ensemble $\{(1,2), (1,3), (1,4), (2,2), (2,3), (2,4)\}$.

- **Expression conditionnelle** : On peut avoir comme expression primaire une expression conditionnelle attribuant un ensemble ou un autre selon un test logique. La syntaxe est :

`if` b `then X else Y`

où `X` et `Y` sont des ensembles et b un test logique.

Les **opérateurs ensemblistes** disponibles sont les suivants :

Formulation	Fonction correspondante
`X union Y`	union $X \cup Y$
`X diff Y`	différence $X \setminus Y$
`X symdiff Y`	différence symétrique $X \oplus Y$
`X inter Y`	intersection $X \cap Y$
`X cross Y`	produit cartésien $X \times Y$

TAB. 7 – Opérateurs ensemblistes

Les opérations sont effectuées de la gauche vers la droite et le résultat est forcément un ensemble.

La hiérarchie générale des opérations devient ainsi :

Opération	Hiérarchie
Evaluation des opérations numériques	$1^{er} - 7^{eme}$
Evaluation des opérations symboliques	$8^{eme} - 9^{eme}$
Evaluation des ensembles itérés et arithmétiques	10^{eme}
Produit cartésien (`cross`)	11^{eme}
Intersection (`inter`)	12^{eme}
Union et difference (`union, diff, symdiff`)	13^{eme}
Evaluation conditionnelle (`if ... then ... else`)	14^{eme}

TAB. 8 – Hiérarchie des opérations dans GLPK

4.5 Expressions logiques

Une expression logique est une règle pour vérifier si un test est vrai ou faux. Les différents types d'expression logique primaire peuvent être formulées à l'aide de :

- **Expressions numériques** : Si l'expression primaire logique est une expression numérique, le résultat est vrai si le résultat est non nul et faux sinon.

- **Expressions relationnelles** : Les différentes opérations relationnelles qui sont utilisées dans des expressions logiques sont :

Formulation	Test correspondant
x `<` y	teste si $x < y$
x `<=` y	teste si $x \leq y$
x `=` y, x `==` y	teste si $x = y$
x `>=` y	teste si $x \geq y$
x `<>` y,x `!=` y	teste si $x \neq y$
x `in Y`	teste si $x \in Y$
$(x_1, \ldots, x_n)$ `in Y`	teste si $(x_1, \ldots, x_n) \in Y$
x `not in Y`, x `!in Y`	teste si $x \notin Y$
$(x_1, \ldots, x_n)$ `not in Y` , $(x_1, \ldots, x_n)$ `!in Y`	teste si $(x_1, \ldots, x_n) \notin Y$
`X within Y`	teste si $X \subseteq Y$
`X not within Y`, `X !within Y`	teste si $X \not\subseteq Y$

TAB. 9 – Tests logiques

où $x, x_1, \ldots, x_n, y$ sont numérique ou symbolique et X et Y des ensembles. Si x et y sont des expressions symboliques, seules les opérateurs `=`,`==`,`<>` et `!=` peuvent être utilisés.

Exemples :
```
a[i,j] < 1.5
(i+1,'Jan') not in I cross J
```

- **Expressions itérées** : Une expression itérée est une expression primaire qui a la forme suivante :

iterated_operator indexing_expression integrand

où *iterated_operator* est un opérateur d'itération, *indexing_expression* est un ensemble d'indices indiquant le nombre d'itérations à effectuer et *integrand* est l'expression logique à itérer.

Il existe 2 opérateurs d'itération :

- `forall` qui a la signification de l'opérateur mathématique $\forall$. Le résultat du test itéré est vrai si le test est vrai pour tous les éléments générés par *indexing_expression* ;
- `exists` qui a la signification de l'opérateur mathématique $\exists$. Le résultat du test itéré est vrai si le test est vrai pour au moins un élément parmi tous les éléments générés par *indexing_expression*.

Exemple :
```
forall{i in I, j in J} a[i,j]<0.5*b[i]
```

Les **opérateurs logiques** disponibles sont :

Formulation	Signification
`not x`, `! x`	negation
`x and y`, `x && y`	ET logique
`x or y`, `x || y`	OU logique

TAB. 10 – Opérateurs logiques

Les opérations sont effectuées de la gauche vers la droite.

La hiérarchie générale des opérations devient ainsi :

Opération	Hiérarchie
Evaluation des opérations numériques	$1^{er} - 7^{eme}$
Evaluation des opérations symboliques	$8^{eme} - 9^{eme}$
Evaluation des opérations ensemblistes	$10^{eme} - 14^{eme}$
Opérations relationnelles	15^{eme}
Négation	16^{eme}
ET logique	17^{eme}
Itération $\forall$ et $\exists$	18^{eme}
OU logique	19^{eme}

TAB. 11 – Hiérarchie des opérations dans GLPK

4.6 Expressions linéaires

Une expression linéaire est une règle pour calculer des formes linéaires ou simplement des formules mathématiques linéaires ou affines. Une expression linéaire peut être composée de :

- **Variables** : Chaque expression primaire peut être basée de variables (ex : T) ou de variables indicées (ex : $x[i, j]$).

- **Expressions itérées** : On peut calculer une expression linéaire avec des opérateurs d'itération conservant la linéarité. Le seul opérateur disponible est :

  `sum` *indexing − expression integrand*

  où *indexing − expression* est l'ensemble d'indices sur lequel porte la somme et *integrand* est une expression linéaire qui est itérée.

  Exemple :
  ```
  sum{j in J} (a[i,j]*x[i,j]+5)
  ```

- **Expressions conditionnelles** : On peut calculer une expression linéaire à l'aide d'une expression conditionnelle de la forme :

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

  où b est une expression logique et f et g des expressions linéaires.

Les **opérateurs arithmétiques** disponibles sont :

Formulation	Signification
`+` f	$f = f + 1$
`-` f	$f = f - 1$
f`+`g	$f + g$
f`-`g	$f - g$
f`*`g	$f * g$
f`/`g	f/g

TAB. 12 – Opérateurs arithmétiques

Les opérations sont effectuées de la gauche vers la droite.

La hiérarchie générale des opérations est la même que celle des expressions numériques.

5 Objets de modélisation

Les objets de modélisation sont les briques de base pour définir un modèle. Il en existe 2 types :
– celles qui permettent de définir un objet ainsi que certaines de ses propriétés : set, parameter, variable, constraint, objective ;
– celles qui permettent de réaliser certaines actions : solve, check, display, printf, loop.
Les différentes expressions définies auparavant peuvent être utilisées dans le paramétrage de ces objets.

5.1 Set : Définition des ensembles

`Set` dans la section Modèle permet de déclarer des ensembles déjà définis dans la section Données ou à définir à l'aide d'expressions.

Forme générale	`set` *name* {*domain* } , *attrib* , . . . , *attrib* ;
Avec	*name* est le nom symbolique de l'ensemble *domain* est optionnel et définit les dimensions et/ou l'ensemble de définition de l'ensemble *attrib, . . . , attrib* est une série d'attributs optionnels
Attributs	`dimen` *n* spécifie la dimension des n-uplets de l'ensemble `within` *expression* qui contraint tous les éléments de l'ensemble à être dans un ensemble plus grand `:=` *expression* qui assigne une valeur fixée ou calculée à l'ensemble `default` *expression* qui spécifie une valeur par défaut à l'ensemble ou à un de ses éléments quand aucune information n'est disponible

`dimen` attribue la dimension des n-uplets d'un ensemble ou de l'élément d'un ensemble. *n* doit être un entier non signé compris entre 1 et 20. Un seul `dimen` peut être mentionné. S'il ne l'est pas, il est calculé automatiquement. Si ce n'est pas possible, la valeur par défaut est 1.

`:=` attribue une valeur à un ensemble ou à un de ses membres. Si `:=` est spécifié, l'ensemble est calculable et aucune donnée n'est nécessaire dans le fichier de données. Par contre, s'il n'est pas mentionné, les données doivent être fournies.

Exemples :

```
set noeuds;

set arcs within (noeuds cross noeuds);

set step{s in 1..maxiter} dimen 2 := if s=1 then arcs else steps[s-1]
      union setof{k in nodes, (i,k) in step[s-1], (k,j) in step[s-1]}(i,j);
```

```
set A{i in I,j in J}, within B[i+1] cross C[j-1], within D diff E, default {('abc',123),
(321,'cba')};
```

5.2 Parameter : Déclaration ou calcul de paramètres

`Parameter` dans la section Modèle permet à l'utilisateur de déclarer des paramètres définis dans la section Données ou de définir des paramètres calculés par des expressions analytiques ou logiques (dans lesquelles les variables du problème ne peuvent intervenir). d'optimisation.

Forme générale	`param` *name* {*domain* } , *attrib* , . . . , *attrib* ;
Avec	*name* est le nom symbolique du paramètre *domain* est optionnel et définit les dimensions et/ou l'ensemble de définition du paramètre *attrib, . . . , attrib* est une série d'attributs optionnels
Attributs optionnels	`integer` pour spécifier que le paramètre est entier `binary` pour spécifier que le paramètre est binaire (0 ou 1) `symbolic` pour spécifier que le paramètre est symbolique *relation avec* `< <= = == >= > <> !=` pour obliger le paramètre à vérifier des conditions formulées via ces opérateurs (sinon erreur !) `in` *expression* pour obliger le paramètre à être dans un certain ensemble `:=` *expression* qui assigne une valeur fixée ou calculée au paramètre `default` *expression* qui spécifie une valeur par défaut au paramètre quand aucune information n'est disponible

Les valeurs étant attribuées par l'utilisateur, les attributs définis ci-dessus permettent essentiellement de vérifier que les données assignées aux paramètres vérifient bien des conditions voulues. Si au moins une d'entre elles ne l'est pas, une erreur apparaîtra lors de l'exécution.

Exemples :

```
param I := 2;

param units{I};

param N := 20, integer, >=0, <=100;

param A {n in 0..N, k in 0..n} := if k=0 or k=n then 1 else A[n-1,k-1]+A[n-1,k];

param p{i in I, j in J}, integer, >=0, <= i+j, in A[i] symdiff B[j],
                         in C[i,j], default 0.5*(i+j);

param month symbolic default 'May' in {'Mar', 'Apr', 'May'};
```

5.3 Variable : Définition des variables du problème

`Variable` permet de définir les variables du problème d'optimisation et certaines conditions sur celles-ci.

Forme générale	`var` *name* {*domain* } , *attrib* , . . . , *attrib* ;
Avec	*name* est le nom symbolique de la variable *domain* est optionnel et définit les dimensions et/ou l'ensemble de définition de la variable *attrib, . . . , attrib* est une série d'attributs optionnels
Attributs optionnels	`integer` pour contraindre la variable à être entière `binary` pour contraindre la variable à être binaire (0 ou 1) `>=` *expression* spécifie une borne inférieure à la variable `<=` *expression* spécifie une borne supérieure à la variable `=` (ou `==`) *expression* spécifie une valeur fixée à la variable

Si le domaine n'est pas specifié, la variable est une simple variable scalaire. Sinon, la variable est un tableau à n dimensions. D'autre part, les conditions doivent être cohérentes entre elles pour ne pas avoir d'erreur à l'exécution (exemple : fixée la variable à une valeur négative et la forcer à être positive).

Exemples :

```
var x = 0;

var y{I,J};

var A{n in I}, integer, >= b[n], <= c[n];

var z{i in I, j in J} >= i+j;
```

5.4 Constraint : Définition des contraintes du problème

`Constraint` permet de définir les contraintes du problème d'optimisation.

Forme générale	`subject to` *name* {*domain* } : *expression* , `=` *expression* ; `subject to` *name* {*domain* } : *expression* , `<=` *expression* ; `subject to` *name* {*domain* } : *expression* , `>=` *expression* ; `subject to` *name* {*domain* } : *expression* , `<=` *expression* , `<=` *expression* ; `subject to` *name* {*domain* } : *expression* , `>=` *expression* , `>=` *expression* ;
Avec	*name* est le nom symbolique de la contrainte *domain* est optionnel et définit le nombre de contraintes de ce type *expressions* sont des expressions linéaires pour calculer les composants de la contrainte (la virgule est facultative)
Remarque	Le mot clé `subject to` peut être réduit à `subj to` ou `s.t.` ou même être supprimé.

Si le domaine n'est pas spécifié, la contrainte est une simple contrainte. Sinon le nombre d'éléments du domaine définit le nombre de contraintes. Les contraintes définies doivent être forcément linéaires.

Exemples :

```
s.t.  C1 : x + y + z >= 0 ;

subject to C2 {t in 1..T, i in 1..I} : x[t] + y[t] <= sqrt[2]*i ;

subj to C3 {t in Ens1, r in Ens2} : sum{k in 1..t} x[k] + y[r] <= 2;
```

5.5 Objective : Définition de la fonction objectif du problème

`Objective` permet de définir l'objectif du problème d'optimisation.

Forme générale	`minimize` *name* {*domain* } : *expression* ; `maximize` *name* {*domain* } : *expression* ;
Avec	*name* est le nom symbolique de l'objectif *domain* est optionnel et définit l'ensemble de définition de l'objectif *expression* est une expression linéaire qui définit l'objectif

Si le domaine n'est pas spécifié, l'objectif est un simple objectif scalaire. Sinon le nombre d'éléments du domaine définit la dimension de l'objectif. L'objectif défini doit être forcément linéaire. D'autre part, on ne peut considérer dans le problème qu'un seul objectif. Si plusieurs objectifs sont définis dans le modèle, le premier rencontré est supposé être l'objectif du problème. Les suivants sont ignorés.

Exemples :

```
minimize obj : x + 1.5*(y+z) ;
maximize profit_total : sum{p in 1..P} profit[p] * produits[p] ;
```

5.6 Solve : Résolution

`solve` lance la résolution du problème d'optimisation.

Forme générale	solve ;
Remarque	`solve` est facultatif et ne peut être utilisé qu'une seule fois. S'il n'est pas mentionné, GLPK résoud tout de même le modèle en considérant qu'il est placé en fin du modèle.

`solve` déclanchant la résolution du problème d'optimisation, tous les paramètres, variables, objectif ou contraintes doivent être définis avant sa déclaration.

5.7 Check : Vérification des données

`Check` permet de vérifier certaines expressions logiques. Si la valeur est fausse, le modèle retourne une erreur.

Forme générale	`check` {*domain* } : *expression* ;
Avec	*domain* est optionnel et définit l'ensemble de vérification *expression* est une expression logique qui doit être vérifiée

Exemples :

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

5.8 Display : Affichage

`Display` permet d'évaluer des expressions et d'écrire leurs valeurs sur l'output standard.

Forme générale	`display` {*domain* } : *item*, ..., *item* ;
Avec	*domain* est optionnel et définit l'ensemble d'affichage *item*, ..., *item* sont les éléments à afficher

Les *item* à afficher peuvent être des ensembles, des paramètres, des variables, des contraintes, un objectif, des expressions,... Si l'*item* est un objet calculable (ensemble, paramètre,...), l'objet est évalué sur le domaine entier et son contenu est entièrement affiché. Si l'objet est une contrainte, GLPK affiche le détail de la contrainte :

Exemple : La contrainte $C_j, j = 1, 2 : \sum_{i=1}^{3} ix_i^j \leq 2$ sera affichée de la forme :

```
C[1] <= Val1:
1 x[1,1]
2 x[2,1]
```

```
3 x[3,1]
C[2] <= Val2:
1 x[1,2]
2 x[2,2]
3 x[3,2]
```

avec $Val1$ et $Val2$ les valeurs de la contrainte estimées par le modèle.

Si l'objet est une expression, l'expression est évaluée et le résultat est affiché.

Si le `display` est situé dans le modèle, seules les quantités de l'*item* calculées avant `display` sont affichées. C'est pourquoi il est préférable de placer la partie `display` après le `solve`.

Exemples :

```
display: 'x=', x, 'y=', y, 'z=', z;
display: sqrt(x**2 + y**2 + z**2);
display{i in I, j in 1..10}: i, j, a[i,j], b[i,j];
```

Exemples : On souhaite faire afficher une variable x[i,j] avec `display` :

```
Volume[1,1] = 56100
Volume[1,2] = 56200
Volume[1,3] = 56300
Volume[2,4] = 56205
Volume[2,5] = 56110
Volume[2,6] = 56015
...
```

5.9 Printf : Affichage formaté

`Printf` permet d'évaluer des expressions et d'écrire leurs valeurs sur l'output standard selon un formalisme choisi.

Forme générale	`printf` {*domain* } : *format*, *expression*, ..., *expression* ;
Avec	*domain* est optionnel et définit l'ensemble d'affichage *format* spécifie le format d'affichage *expression*, ..., *expression* est la liste des éléments à formater et à afficher

Le format est facultatif : soit on ne l'indique pas et les caractères sont copiés tels quels, soit on l'affiche et il définit un format particulier. Les principaux types de format sont définis comme en C par `%d, %f, %e, %E, %g, %G, %s`. On rappelle que :

– `%d` : entier
– `%f` : réel (float)
– `%e` : réel sous format xxx.yyy e+X
– `%E` : réel sous format xxx.yyy E+X
– `%g` : format le plus court entre `%f` et `%e`
– `%G` : format le plus court entre `%f` et `%E`
– `%s` : chaîne de caractères

De même, on rappelle que `\n` permet de passer à la ligne, `\t` d'introduire une tabulation, `\b` un backspace et `\v` un Tab vertibal.

Exemples :

```
printf: 'Hello world! \n';
printf: "x= %.3f; y=%.3f z=%.3f\n", x, y, z;
printf{i in I}: 'flot total de %s est %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];
```

Exemple : On souhaite faire afficher une variable x[i,j] avec `printf` :

```
printf{i in 1..T}:%d %.5f %.5f  \n,i,Volume[1,i],Volume[2,i];
```

```
1 56100.00000 170.00000
2 56205.00000 217.00000
...
```

5.10 For : Répétition d'actions

`For` permet de répéter des actions un nombre déterminé de fois.

Forme générale	`for` {*domain* } : *statement* ; `for` {*domain* } : {*statement* ... *statement* } ;
Avec	*domain* définit l'ensemble de répétition *statement* est l'expression qui est itérée {*statement* ... *statement* } est une séquence d'expressions à itérer
Remarque	Seules des expressions avec `check`, `display`, `printf` et `for` sont autorisées dans un `for`.

Si le domaine n'est pas spécifié, l'objectif est un simple ojectif scalaire. Sinon le nombre d'éléments du domaine définit la dimension de l'objectif. L'objectif défini doit être forcément linéaire. D'autre part, on ne peut considérer dans le problème qu'un seul objectif. Si plusieurs objectifs sont définis dans le modèle, le premier rencontré est supposé être l'objectif du problème. Les suivants sont ignorés.

Exemples :

```
for {1..72} printf("*");



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

6 Données

Dans cette partie, nous allons voir comment définir des données. Il faut bien voir que toute donnée qui est définie dans la section Données doit être préalablement déclarée dans la section modèle. Les règles de codage sont les mêmes que dans la section modèle. Une seule exception est la non-obligation des " "pour définir les chaînes de caractères contenant uniquement des caractères alphanumériques, des + et - et/ou le point décimal.

Exception : Si une donnée doit être calculée, il faut la placer dans la section Modele.

On peut définir 2 types de données : les ensembles et les paramètres.

6.1 Ensembles

Dans la section Données, on ne peut définir des ensembles qu'avec des données figées.

Forme générale	`set` *name* , *record* , . . . , *record* ; `set` [*symbol*, . . . , *symbol*] , *record* , . . . , *record* ;
Avec	*name* est le nom symbolique de l'ensemble *symbol*, . . . , *symbol* sont les indices qui spécifient un élément particulier de l'ensemble *record* , . . . , *record* sont les différentes entrées de données
Records	`:=` est facultatif mais permet une meilleure lisibilité `(` *slice* `)` spécifie un n-uplet *simple-data* définit un ensemble dans un format simple *: matrix data* définit un ensemble dans un format matriciel `(tr)` *: matrix data* définit un ensemble sous la forme de la transposée d'une matrice

Exemples :

```
set month := "Jan" "Fev" "Mar" "Avr" "Mai" "Jui";

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

```
          3 - + -

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

Dans cet exemple, l'ensemble `month` est un ensemble de singletons, `A` est un tableau bi-dimensionnel de couples et `B` un ensemble de triplets.

Supposons le n-uplet $(s_1, s_2, \ldots, s_n)$. Chaque composant peut être un nombre, un symbole ou un `*`. Le nombre de composants doit toujours être identique. L'utilisation d'un `*` est utile dans la saisie. Par exemple, si on considère les couples `(1,2)` et `(1,3)`, on peut définir un format fixe `(1,*)` où seule la deuxième composante varie et peut valoir 2 ou 3. En pratique, on peut remplacer `A := (1,2) (1,3);` par `A := (1,*) 2 3;` . On peut ainsi réduire une longue liste assez facilement

Si le n-uplet est un couple, on peut les définir sous la forme matricielle :

$$\begin{array}{cccccc} : & c_1 & c_2 & \cdots & c_n & := \\ l_1 & a_{11} & a_{12} & \cdots & a_{1n} & \\ l_2 & a_{21} & a_{22} & \cdots & a_{2n} & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \\ l_m & a_{m1} & a_{m2} & \cdots & a_{mn} & \end{array}$$

où l_i est le numéro/symbole correspondant à une ligne de la matrice, c_i le numéro/symbole correspondant à une colonne de la matrice et chaque a_{ij} est un élément possible de la matrice (correspondant au couple (l_i, c_j)), qui peut être `+` et `-`. Si a_{ij} est le symbole `+`, le couple correspondant appartient à l'ensemble. Sinon, c'est un `-` et il n'y appartient pas.

On peut définir de la même sorte la transposée de la matrice :

$$\begin{array}{cccccc} \texttt{(tr)} & : & c_1 & c_2 & \cdots & c_n & := \\ & l_1 & a_{11} & a_{12} & \cdots & a_{1n} & \\ & l_2 & a_{21} & a_{22} & \cdots & a_{2n} & \\ & \cdots & \cdots & \cdots & \cdots & \cdots & \\ & l_m & a_{m1} & a_{m2} & \cdots & a_{mn} & \end{array}$$

La saisie est la même que précédemment hormis le fait que l'élément a_{ij} correspond désormais au couple (c_j, l_i) au lieu de (l_i, c_j).

6.2 Paramètres

Forme générale	`param` *name* , *record* , . . . , *record* ; `param` *name* `default` *value* , *record* , . . . , *record* ; `param` : *tabbing-data* ; `param default` *value* : *tabbing-data* ;
Avec	*name* est le nom symbolique du paramètre *value* est une valeur par défaut du paramètre *record* , . . . , *record* sont les différentes entrées de données *tabbing-data* représente l'entrée des données sous la forme d'un tableau
Records	`:=` est facultatif mais permet une meilleure lisibilité `[` *slice* `]` spécifie un n-uplet *plain-data* définit un ensemble dans un format plain $(indice_1, \ldots, indice_n, valeur)$ *: tabular data* définit un ensemble dans un tableau `(tr)` *: tabular data* définit un paramètre sous la forme de la transposée d'une matrice

Supposons le n-uplet $[s_1, s_2, \ldots, s_n]$. Chaque composant peut être un nombre, un symbole ou un `*`. Le nombre de composants doit toujours être identique. L'utilisation d'un `*` est utile dans la saisie. Par exemple, si on considère les couples `[1,2]` et `[1,3]`, on peut définir un format fixe `[1,*]` où seule la deuxième composante varie et peut valoir 2 ou 3. En pratique, on peut remplacer `A := [1,2] [1,3];` par `A := [1,*] 2 3;` . On peut ainsi réduire une longue liste assez facilement.

Quand le paramètre est un simple tableau, on peut le remplir dans le format plain, c'est à dire de la forme : `param T := 1, 1, 2, 2, 3, 3;` ou `param T := 1 1 2 2 3 3;` ou même

```
param T:= 1 1
          2 2
          3 3;
```

Les espaces n'étant pas pris en compte, la disposition peut être adaptée comme on le souhaite.

Tout comme pour les ensembles, si le n-uplet est un doublet, on peut les définir sous la forme matricielle :

$$\begin{array}{ccccc:c} : & c_1 & c_2 & \cdots & c_n & := \\ l_1 & a_{11} & a_{12} & \cdots & a_{1n} & \\ l_2 & a_{21} & a_{22} & \cdots & a_{2n} & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \\ l_m & a_{m1} & a_{m2} & \cdots & a_{mn} & \end{array}$$

où l_i est le numéro/symbole correspondant à une ligne de la matrice, c_i le numéro/symbole correspondant à une colonne de la matrice et chaque a_{ij} est la veleur correspondante au doublet (l_i, c_j)), qui peut être un nombre, un symbole ou le point décimal. Si a_{ij} est un nombre ou un symbole, cette quantité est assignée à l'élément [l_i,r_j] du paramètre. Si c'est un point décimal, la valeur par défaut définie est attribuée. Si aucune valeur par défaut n'a été attribuée, l'élément

$[l_i,r_j]$ du paramètre n'est pas défini (dans ce cas, faire bien attention de ne pas faire d'opération avec cet élément n'existant pas).

Tout comme pour les ensembles, on peut aussi définir les données en remplissant la transposée :

$$\begin{array}{lccccc} \texttt{(tr)} & : & c_1 & c_2 & \cdots & c_n & := \\ & l_1 & a_{11} & a_{12} & \cdots & a_{1n} & \\ & l_2 & a_{21} & a_{22} & \cdots & a_{2n} & \\ & \cdots & \cdots & \cdots & \cdots & \cdots & \\ & l_m & a_{m1} & a_{m2} & \cdots & a_{mn} & \end{array}$$

Dans ce cas, chaque a_{ij} correspond correspond à l'élément d'indices $[r_j,l_i]$ au lieu de $[l_i,r_j]$.

Un moyen très pratique de définir des paramètres de mêmes dimensions est le *tabbing data format*. Au lieu de définir chaque paramètre un à un, on peut les définir dans un tableau d'un seul coup :

$$\begin{array}{llll} \texttt{param default value :s} & : & p_1 \quad p_2 \quad \cdots \quad p_r & := \\ r_{11}\ , \quad r_{12}\ , \quad \cdots\ , \quad r_{1n}\ , & l_1 & a_{11} \quad a_{12} \quad \cdots \quad a_{1r} & \\ r_{21}\ , \quad r_{22}\ , \quad \cdots\ , \quad r_{2n}\ , & l_2 & a_{21} \quad a_{22} \quad \cdots \quad a_{2r} & \\ \cdots \quad \cdots \quad \cdots \quad \cdots & \cdots & \cdots \quad \cdots \quad \cdots \quad \cdots & \\ r_{m1}\ , \quad r_{m2}\ , \quad \cdots\ , \quad r_{mn}\ , & l_m & a_{m1} \quad a_{m2} \quad \cdots \quad a_{mr} & \end{array}$$

Le nom symbolique `:s` est facultatif, il permet de créer un ensemble. Les virgules le sont aussi.

Cette écriture générale correspond en fait à la série d'instructions :

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

Exemples :

```
param T := 4;

param month := 1 Jan 2 Fev 3 Mar 4 Avr 5 Mai;
param month := [1] 'Jan', [2] 'Fev', [3] 'Mar', [4] 'Avr', [5] 'Mai';
# Pour ces 3 versions, il faut préalablement déclarer le param month dans le fichier .mod er
# param month{1..5} symbolic;

param stock_init := acier 7.32 nickel 35.8 ;
param cout := [acier] .025 [nickel] .03 ;
param quant := acier 20, nickel 12;

# On peut définir tous les paramètres d'un bloc (s'ils ont les mêmes dimensions):
param        : stock_init  cout   quant :=
      acier       7.32     .025    20
      nickel      35.8     .03     12 ;

#Définition d'un ensemble
param : bilan : stock_init  cout   quant :=
      acier       7.32     .025    20
      nickel      35.8     .03     12 ;

param demand default 0 (tr)
         : FRA DAN GBR ALL SUE ITA ESP :=
acier      300  .  100 75   .  225 250
fer        500 750 400 250  .  850 500
nickel     100  .   .   50  .  200  .  ;

param couts_trans :=
 [*,*,acier]: FRA DAN GBR ALL SUE ITA ESP :=
 usine1        30  10   8  10  11  71   6
 usine2        22   7  23  14  17  28  26
 usine3        10  12   9  18  14  51  16

[*,*,fer]:    FRA DAN GBR ALL SUE ITA ESP :=
 usine1        40  20  18  20  21  81  16
 usine2        32  17  33  24  27  38  36
 usine3        20  22  19  28  24  61  26

[*,*,nickel]: FRA DAN GBR ALL SUE ITA ESP :=
 usine1        20  10   8  10   1  61   6
 usine2        12  17  13   4   7  18  16
 usine3         1   2   9   8   4  41   6
```