

# Chap. III : Les objets structurés

Laurent Poinsot

25 septembre 2009

- 1 Les objets structurés
  - Introduction
  - Séquences
  - Listes
  - Ensembles
  - Chaînes de caractères

- 2 Quelques exercices

- 1** Les objets structurés
  - Introduction
  - Séquences
  - Listes
  - Ensembles
  - Chaînes de caractères
  
- 2** Quelques exercices

- 1** Les objets structurés
  - Introduction
  - Séquences
  - Listes
  - Ensembles
  - Chaînes de caractères

- 2** Quelques exercices

Pour connaître le type d'un objet Maple il faut utiliser la commande `whattype` :

```
> whattype(3) ;  
integer
```

Dans ce cours nous avons déjà rencontré des objets de types "simples", tels que `integer`, `fraction` (ex.  $2/3$ ), `float` (ex.  $2.3$ ,  $\pi$ ) et `complex` (ex.  $2 + 3 * I$ ). En les combinant, on peut construire des objets de types "structurés".

Pour connaître le type d'un objet Maple il faut utiliser la commande `whattype` :

```
> whattype(3) ;
```

*integer*

Dans ce cours nous avons déjà rencontré des objets de types "simples", tels que *integer*, *fraction* (ex.  $2/3$ ), *float* (ex.  $2.3$ ,  $\pi$ ) et *complex* (ex.  $2 + 3 * I$ ). En les combinant, on peut construire des objets de types "structurés".

Pour connaître le type d'un objet Maple il faut utiliser la commande `whattype` :

```
> whattype(3) ;
```

*integer*

Dans ce cours nous avons déjà rencontré des objets de types "simples", tels que `integer`, `fraction` (ex.  $2/3$ ), `float` (ex.  $2.3$ ,  $\pi$ ) et `complex` (ex.  $2 + 3 * I$ ). En les combinant, on peut construire des objets de types "structurés".

Pour connaître le type d'un objet Maple il faut utiliser la commande `whattype` :

```
> whattype(3) ;
```

*integer*

Dans ce cours nous avons déjà rencontré des objets de types "simples", tels que `integer`, `fraction` (ex.  $2/3$ ), `float` (ex.  $2.3$ ,  $\pi$ ) et `complex` (ex.  $2 + 3 * I$ ). En les combinant, on peut construire des objets de types "structurés".

Pour connaître le type d'un objet Maple il faut utiliser la commande `whattype` :

```
> whattype(3) ;
```

*integer*

Dans ce cours nous avons déjà rencontré des objets de types "simples", tels que `integer`, `fraction` (ex.  $2/3$ ), `float` (ex.  $2.3$ ,  $\pi$ ) et `complex` (ex.  $2 + 3 * I$ ). En les combinant, on peut construire des objets de types "structurés".

Pour connaître le type d'un objet Maple il faut utiliser la commande `whattype` :

```
> whattype(3) ;
```

*integer*

Dans ce cours nous avons déjà rencontré des objets de types "simples", tels que `integer`, `fraction` (ex.  $2/3$ ), `float` (ex.  $2.3$ ,  $\pi$ ) et `complex` (ex.  $2 + 3 * I$ ). En les combinant, on peut construire des objets de types "structurés".

Pour connaître le type d'un objet Maple il faut utiliser la commande `whattype` :

```
> whattype(3) ;
```

*integer*

Dans ce cours nous avons déjà rencontré des objets de types "simples", tels que `integer`, `fraction` (ex.  $2/3$ ), `float` (ex.  $2.3$ ,  $\pi$ ) et `complex` (ex.  $2 + 3 * I$ ). En les combinant, on peut construire des objets de types "structurés".

Pour connaître le type d'un objet Maple il faut utiliser la commande `whattype` :

```
> whattype(3) ;
```

*integer*

Dans ce cours nous avons déjà rencontré des objets de types "simples", tels que `integer`, `fraction` (ex.  $2/3$ ), `float` (ex.  $2.3$ ,  $\pi$ ) et `complex` (ex.  $2 + 3 * I$ ). En les combinant, on peut construire des objets de types "structurés".

Pour connaître le type d'un objet Maple il faut utiliser la commande `whattype` :

```
> whattype(3) ;
```

*integer*

Dans ce cours nous avons déjà rencontré des objets de types "simples", tels que `integer`, `fraction` (ex.  $2/3$ ), `float` (ex.  $2.3$ ,  $\pi$ ) et `complex` (ex.  $2 + 3 * I$ ). En les combinant, on peut construire des objets de types "structurés".

Voici la liste des cinq types structurés disponibles sous Maple :

- 1 Les **séquences** (`exprseq`);
- 2 Les **listes** (`list`);
- 3 Les **ensembles** (`set`);
- 4 Les **chaînes de caractères** (`string`);
- 5 Les **tableaux** (on les verra plus tard dans le chapitre sur les matrices).

Voici la liste des cinq types structurés disponibles sous Maple :

- 1 Les **séquences** (`exprseq`);
- 2 Les **listes** (`list`);
- 3 Les **ensembles** (`set`);
- 4 Les **chaînes de caractères** (`string`);
- 5 Les **tableaux** (on les verra plus tard dans le chapitre sur les matrices).

Voici la liste des cinq types structurés disponibles sous Maple :

- 1 Les **séquences** (`exprseq`);
- 2 Les **listes** (`list`);
- 3 Les **ensembles** (`set`);
- 4 Les **chaînes de caractères** (`string`);
- 5 Les **tableaux** (on les verra plus tard dans le chapitre sur les matrices).

Voici la liste des cinq types structurés disponibles sous Maple :

- 1 Les **séquences** (`exprseq`);
- 2 Les **listes** (`list`);
- 3 Les **ensembles** (`set`);
- 4 Les **chaînes de caractères** (`string`);
- 5 Les **tableaux** (on les verra plus tard dans le chapitre sur les matrices).

Voici la liste des cinq types structurés disponibles sous Maple :

- 1 Les **séquences** (`exprseq`);
- 2 Les **listes** (`list`);
- 3 Les **ensembles** (`set`);
- 4 Les **chaînes de caractères** (`string`);
- 5 Les **tableaux** (on les verra plus tard dans le chapitre sur les matrices).

Voici la liste des cinq types structurés disponibles sous Maple :

- 1 Les **séquences** (`exprseq`);
- 2 Les **listes** (`list`);
- 3 Les **ensembles** (`set`);
- 4 Les **chaînes de caractères** (`string`);
- 5 Les **tableaux** (on les verra plus tard dans le chapitre sur les matrices).

# Plan

Chap. III :  
Les objets  
structurés

Laurent  
Poinso

Les objets  
structurés

Quelques  
exercices

## 1 Les objets structurés

- Introduction
- Séquences
- Listes
- Ensembles
- Chaînes de caractères

## 2 Quelques exercices

Une **séquence** est une suite d'objets quelconques, séparés par des virgules :

```
> S := 1, a, 2, X, 3 ;
```

```
S := 1, a, 2, X, 3
```

```
> whattype (S) ;
```

```
exprseq
```

Les éléments dans une séquence sont numérotés de 1 à  $l$  ( $l$  est la longueur de la séquence). Par ex. dans  $S$ , l'élément numéro 2 est  $a$ . Pour sélectionner l'élément numéro  $n$  dans une séquence  $seq$ , on tape  $seq[n]$ . Par ex. pour obtenir l'élément numéro 2 de  $S$  :

```
> S[2] ;
```

```
a
```

Une **séquence** est une suite d'objets quelconques, séparés par des virgules :

```
> S := 1, a, 2, X, 3 ;
```

```
S := 1, a, 2, X, 3
```

```
> whattype (S) ;
```

```
exprseq
```

Les éléments dans une séquence sont numérotés de 1 à  $l$  ( $l$  est la longueur de la séquence). Par ex. dans  $S$ , l'élément numéro 2 est  $a$ . Pour sélectionner l'élément numéro  $n$  dans une séquence  $seq$ , on tape  $seq[n]$ . Par ex. pour obtenir l'élément numéro 2 de  $S$  :

```
> S[2] ;
```

```
a
```

Une **séquence** est une suite d'objets quelconques, séparés par des virgules :

```
> S := 1, a, 2, X, 3 ;
```

```
S := 1, a, 2, X, 3
```

```
> whattype (S) ;
```

```
exprseq
```

Les éléments dans une séquence sont numérotés de 1 à  $l$  ( $l$  est la longueur de la séquence). Par ex. dans  $S$ , l'élément numéro 2 est  $a$ . Pour sélectionner l'élément numéro  $n$  dans une séquence  $seq$ , on tape  $seq[n]$ . Par ex. pour obtenir l'élément numéro 2 de  $S$  :

```
> S[2] ;
```

```
a
```

Une **séquence** est une suite d'objets quelconques, séparés par des virgules :

```
> S := 1, a, 2, X, 3 ;
```

```
S := 1, a, 2, X, 3
```

```
> whattype (S) ;
```

```
exprseq
```

Les éléments dans une séquence sont numérotés de 1 à  $l$  ( $l$  est la longueur de la séquence). Par ex. dans  $S$ , l'élément numéro 2 est  $a$ . Pour sélectionner l'élément numéro  $n$  dans une séquence  $seq$ , on tape  $seq[n]$ . Par ex. pour obtenir l'élément numéro 2 de  $S$  :

```
> S[2] ;
```

```
a
```

Une **séquence** est une suite d'objets quelconques, séparés par des virgules :

```
> S := 1, a, 2, X, 3 ;
```

$S := 1, a, 2, X, 3$

```
> whattype (S) ;
```

exprseq

Les éléments dans une séquence sont numérotés de 1 à  $l$  ( $l$  est la longueur de la séquence). Par ex. dans  $S$ , l'élément numéro 2 est  $a$ . Pour sélectionner l'élément numéro  $n$  dans une séquence  $seq$ , on tape  $seq[n]$ . Par ex. pour obtenir l'élément numéro 2 de  $S$  :

```
> S[2] ;
```

$a$

Une **séquence** est une suite d'objets quelconques, séparés par des virgules :

```
> S := 1, a, 2, X, 3 ;
```

```
S := 1, a, 2, X, 3
```

```
>whattype (S) ;
```

```
exprseq
```

Les éléments dans une séquence sont numérotés de 1 à  $l$  ( $l$  est la longueur de la séquence). Par ex. dans  $S$ , l'élément numéro 2 est  $a$ . Pour sélectionner l'élément numéro  $n$  dans une séquence  $seq$ , on tape  $seq[n]$ . Par ex. pour obtenir l'élément numéro 2 de  $S$  :

```
> S[2] ;
```

```
a
```

Une **séquence** est une suite d'objets quelconques, séparés par des virgules :

```
> S := 1, a, 2, X, 3 ;
```

```
S := 1, a, 2, X, 3
```

```
> whattype (S) ;
```

```
exprseq
```

Les éléments dans une séquence sont numérotés de 1 à  $l$  ( $l$  est la longueur de la séquence). Par ex. dans  $S$ , l'élément numéro 2 est  $a$ . Pour sélectionner l'élément numéro  $n$  dans une séquence  $seq$ , on tape  $seq[n]$ . Par ex. pour obtenir l'élément numéro 2 de  $S$  :

```
> S[2] ;
```

```
a
```

Une **séquence** est une suite d'objets quelconques, séparés par des virgules :

```
> S := 1, a, 2, X, 3 ;
```

```
S := 1, a, 2, X, 3
```

```
>whattype (S) ;
```

```
exprseq
```

Les éléments dans une séquence sont numérotés de 1 à  $l$  ( $l$  est la longueur de la séquence). Par ex. dans  $S$ , l'élément numéro 2 est  $a$ . Pour sélectionner l'élément numéro  $n$  dans une séquence  $seq$ , on tape  $seq[n]$ . Par ex. pour obtenir l'élément numéro 2 de  $S$  :

```
> S[2] ;
```

```
a
```

Une **séquence** est une suite d'objets quelconques, séparés par des virgules :

```
> S := 1, a, 2, X, 3 ;
```

```
S := 1, a, 2, X, 3
```

```
>whattype (S) ;
```

```
exprseq
```

Les éléments dans une séquence sont numérotés de 1 à  $l$  ( $l$  est la longueur de la séquence). Par ex. dans  $S$ , l'élément numéro 2 est  $a$ . Pour sélectionner l'élément numéro  $n$  dans une séquence  $seq$ , on tape  $seq[n]$ . Par ex. pour obtenir l'élément numéro 2 de  $S$  :

```
> S[2] ;
```

```
a
```

Une **séquence** est une suite d'objets quelconques, séparés par des virgules :

```
> S := 1, a, 2, X, 3 ;
```

```
S := 1, a, 2, X, 3
```

```
> whattype (S) ;
```

```
exprseq
```

Les éléments dans une séquence sont numérotés de 1 à  $l$  ( $l$  est la longueur de la séquence). Par ex. dans  $S$ , l'élément numéro 2 est  $a$ . Pour sélectionner l'élément numéro  $n$  dans une séquence  $seq$ , on tape  $seq[n]$ . Par ex. pour obtenir l'élément numéro 2 de  $S$  :

```
> S[2] ;
```

```
a
```

Une **séquence** est une suite d'objets quelconques, séparés par des virgules :

```
> S := 1, a, 2, X, 3 ;
```

$S := 1, a, 2, X, 3$

```
> whattype (S) ;
```

exprseq

Les éléments dans une séquence sont numérotés de 1 à  $l$  ( $l$  est la longueur de la séquence). Par ex. dans  $S$ , l'élément numéro 2 est  $a$ . Pour sélectionner l'élément numéro  $n$  dans une séquence  $seq$ , on tape  $seq[n]$ . Par ex. pour obtenir l'élément numéro 2 de  $S$  :

```
> S[2] ;
```

$a$

On peut changer la valeur de l'élément numéro 3 de  $S$  par :

```
> S; # Affiche la séquence S
```

1, a, 2, X, 3

```
>S[3]; # Affiche l'élément 3 de S
```

2

```
>S[3] :=Pi; # On remplace 3 par  $\pi$  dans S
```

S[3] :=  $\pi$

```
>S; # On affiche de nouveau la séquence S
```

1, a,  $\pi$ , X, 3

On peut changer la valeur de l'élément numéro 3 de  $S$  par :

```
> S; # Affiche la séquence S
```

1, a, 2, X, 3

```
>S[3]; # Affiche l'élément 3 de S
```

2

```
>S[3] :=Pi; # On remplace 3 par  $\pi$  dans S
```

S[3] :=  $\pi$

```
>S; # On affiche de nouveau la séquence S
```

1, a,  $\pi$ , X, 3

On peut changer la valeur de l'élément numéro 3 de  $S$  par :

```
> S; # Affiche la séquence  $S$ 
```

$1, a, 2, X, 3$

```
>S[3]; # Affiche l'élément 3 de  $S$ 
```

2

```
>S[3] :=Pi; # On remplace 3 par  $\pi$  dans  $S$ 
```

$S[3] := \pi$

```
>S; # On affiche de nouveau la séquence  $S$ 
```

$1, a, \pi, X, 3$

On peut changer la valeur de l'élément numéro 3 de  $S$  par :

```
> S; # Affiche la séquence S
```

1, a, 2, X, 3

```
>S[3]; # Affiche l'élément 3 de S
```

2

```
>S[3] :=Pi; # On remplace 3 par  $\pi$  dans S
```

S[3] :=  $\pi$

```
>S; # On affiche de nouveau la séquence S
```

1, a,  $\pi$ , X, 3

On peut changer la valeur de l'élément numéro 3 de  $S$  par :

```
> S; # Affiche la séquence S
```

1, a, 2, X, 3

```
>S[3]; # Affiche l'élément 3 de S
```

2

```
>S[3] :=Pi; # On remplace 3 par  $\pi$  dans S
```

S[3] :=  $\pi$

```
>S; # On affiche de nouveau la séquence S
```

1, a,  $\pi$ , X, 3

On peut changer la valeur de l'élément numéro 3 de  $S$  par :

```
> S; # Affiche la séquence  $S$ 
```

$1, a, 2, X, 3$

```
>S[3]; # Affiche l'élément 3 de  $S$ 
```

2

```
>S[3] :=Pi; # On remplace 3 par  $\pi$  dans  $S$ 
```

$S[3] := \pi$

```
>S; # On affiche de nouveau la séquence  $S$ 
```

$1, a, \pi, X, 3$

On peut changer la valeur de l'élément numéro 3 de  $S$  par :

```
> S ; # Affiche la séquence  $S$ 
```

$1, a, 2, X, 3$

```
>S[3] ; # Affiche l'élément 3 de  $S$ 
```

$2$

```
>S[3] :=Pi ; # On remplace 3 par  $\pi$  dans  $S$ 
```

$S[3] := \pi$

```
>S ; # On affiche de nouveau la séquence  $S$ 
```

$1, a, \pi, X, 3$

On peut changer la valeur de l'élément numéro 3 de  $S$  par :

```
> S; # Affiche la séquence  $S$ 
```

$1, a, 2, X, 3$

```
>S[3]; # Affiche l'élément 3 de  $S$ 
```

$2$

```
>S[3] :=Pi; # On remplace  $3$  par  $\pi$  dans  $S$ 
```

$S[3] := \pi$

```
>S; # On affiche de nouveau la séquence  $S$ 
```

$1, a, \pi, X, 3$

On peut changer la valeur de l'élément numéro 3 de  $S$  par :

```
> S; # Affiche la séquence  $S$ 
```

$1, a, 2, X, 3$

```
>S[3]; # Affiche l'élément 3 de  $S$ 
```

$2$

```
>S[3] :=Pi; # On remplace  $3$  par  $\pi$  dans  $S$ 
```

$S[3] := \pi$

```
>S; # On affiche de nouveau la séquence  $S$ 
```

$1, a, \pi, X, 3$

On peut changer la valeur de l'élément numéro 3 de  $S$  par :

```
> S; # Affiche la séquence  $S$ 
```

$1, a, 2, X, 3$

```
>S[3]; # Affiche l'élément 3 de  $S$ 
```

$2$

```
>S[3] :=Pi; # On remplace  $3$  par  $\pi$  dans  $S$ 
```

$S[3] := \pi$

```
>S; # On affiche de nouveau la séquence  $S$ 
```

$1, a, \pi, X, 3$

On peut changer la valeur de l'élément numéro 3 de  $S$  par :

```
> S ; # Affiche la séquence  $S$ 
```

$1, a, 2, X, 3$

```
>S[3] ; # Affiche l'élément 3 de  $S$ 
```

$2$

```
>S[3] :=Pi ; # On remplace  $3$  par  $\pi$  dans  $S$ 
```

$S[3] := \pi$

```
>S ; # On affiche de nouveau la séquence  $S$ 
```

$1, a, \pi, X, 3$

On peut changer la valeur de l'élément numéro 3 de  $S$  par :

```
> S ; # Affiche la séquence  $S$ 
```

$1, a, 2, X, 3$

```
>S[3] ; # Affiche l'élément 3 de  $S$ 
```

$2$

```
>S[3] :=Pi ; # On remplace  $3$  par  $\pi$  dans  $S$ 
```

$S[3] := \pi$

```
>S ; # On affiche de nouveau la séquence  $S$ 
```

$1, a, \pi, X, 3$

La **séquence vide** se note `NULL` :

```
> T :=NULL ;
```

```
T :=
```

La **séquence vide** se note NULL :

> T :=NULL ;

*T :=*

La **séquence vide** se note `NULL` :

`> T :=NULL ;`

*T* :=

Pour **concaténer** des séquences, c'est-à-dire les juxtaposer les unes à la suite des autres, on écrit séparées par des virgules ",", les deux séquences :

> U := 4, 6, 8 ;

> S, T, U ;

1, a,  $\pi$ , X, 3, 4, 6, 8

Pour **concaténer** des séquences, c'est-à-dire les juxtaposer les unes à la suite des autres, on écrit séparées par des virgules ",", les deux séquences :

> U := 4, 6, 8 :

> S, T, U ;

1, a,  $\pi$ , X, 3, 4, 6, 8

Pour **concaténer** des séquences, c'est-à-dire les juxtaposer les unes à la suite des autres, on écrit séparées par des virgules ”,”, les deux séquences :

> U :=4, 6, 8 :

>S, T, U ;

1, a,  $\pi$ , X, 3, 4, 6, 8

Pour **concaténer** des séquences, c'est-à-dire les juxtaposer les unes à la suite des autres, on écrit séparées par des virgules ”,”, les deux séquences :

> U :=4, 6, 8 :

>S, T, U ;

1, a,  $\pi$ , X, 3, 4, 6, 8

Pour **concaténer** des séquences, c'est-à-dire les juxtaposer les unes à la suite des autres, on écrit séparées par des virgules ”,”, les deux séquences :

> U :=4, 6, 8 :

>S, T, U ;

1, a,  $\pi$ , X, 3, 4, 6, 8

Pour **concaténer** des séquences, c'est-à-dire les juxtaposer les unes à la suite des autres, on écrit séparées par des virgules ”,”, les deux séquences :

> U :=4, 6, 8 :

>S, T, U ;

1, a,  $\pi$ , X, 3, 4, 6, 8

Voici deux façons de créer automatiquement des séquences :

1 En utilisant le symbole "\$" :

```
> x$5 ;
```

$x, x, x, x, x$

2 En utilisant la commande seq :

```
seq(1/n, n=1..6) ;
```

$1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \frac{1}{6}$

Voici deux façons de créer automatiquement des séquences :

**1** En utilisant le symbole "\$" :

```
> x$5 ;
```

$x, x, x, x, x$

**2** En utilisant la commande seq :

```
seq(1/n, n=1..6) ;
```

$1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \frac{1}{6}$

Voici deux façons de créer automatiquement des séquences :

**1** En utilisant le symbole "\$" :

```
> x$5 ;
```

$x, x, x, x, x$

**2** En utilisant la commande seq :

```
seq(1/n, n=1..6) ;
```

$1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \frac{1}{6}$

Voici deux façons de créer automatiquement des séquences :

**1** En utilisant le symbole "\$" :

```
> x$5 ;
```

$x, x, x, x, x$

**2** En utilisant la commande seq :

```
seq(1/n, n=1..6) ;
```

$1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \frac{1}{6}$

Voici deux façons de créer automatiquement des séquences :

**1** En utilisant le symbole "\$" :

```
> x$5 ;
```

$x, x, x, x, x$

**2** En utilisant la commande `seq` :

```
seq(1/n, n=1..6) ;
```

$1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \frac{1}{6}$

Voici deux façons de créer automatiquement des séquences :

**1** En utilisant le symbole "\$" :

```
> x$5 ;
```

$x, x, x, x, x$

**2** En utilisant la commande `seq` :

```
seq(1/n, n=1..6) ;
```

$1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \frac{1}{6}$

Voici deux façons de créer automatiquement des séquences :

**1** En utilisant le symbole "\$" :

```
> x$5 ;
```

$x, x, x, x, x$

**2** En utilisant la commande `seq` :

```
seq(1/n, n=1..6) ;
```

$1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \frac{1}{6}$

# Plan

Chap. III :  
Les objets  
structurés

Laurent  
Poinsot

Les objets  
structurés

Quelques  
exercices

- 1 Les objets structurés
  - Introduction
  - Séquences
  - **Listes**
  - Ensembles
  - Chaînes de caractères

- 2 Quelques exercices

Une **liste** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des crochets "[" et "]" :

```
> L := [1, 5, a, 8] ;
```

```
L := [1, 5, a, 8]
```

```
> whattype (L) ;
```

```
list
```

Une **liste** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des crochets "[" et "]" :

```
> L := [1, 5, a, 8] ;
```

```
L := [1, 5, a, 8]
```

```
> whattype (L) ;
```

```
list
```

Une **liste** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des crochets "[" et "]" :

```
> L := [1, 5, a, 8] ;
```

```
L := [1, 5, a, 8]
```

```
> whattype (L) ;
```

```
list
```

Une **liste** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des crochets "[" et "]" :

```
> L := [1, 5, a, 8] ;
```

```
L := [1, 5, a, 8]
```

```
>whattype(L) ;
```

```
list
```

Une **liste** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des crochets "[" et "]" :

```
> L := [1, 5, a, 8] ;
```

$$L := [1, 5, a, 8]$$

```
>whattype(L) ;
```

*list*

Une **liste** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des crochets "[" et "]" :

```
> L := [1, 5, a, 8] ;
```

$$L := [1, 5, a, 8]$$

```
>whattype (L) ;
```

*list*

Une **liste** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des crochets "[" et "]" :

```
> L := [1, 5, a, 8] ;
```

$$L := [1, 5, a, 8]$$

```
>whattype (L) ;
```

*list*

Pour transformer une liste en une séquence, il faut enlever les crochets par la commande `op` :

```
> S :=op(L) ;
```

```
S := 1, 5, a, 8
```

Pour transformer une liste en une séquence, il suffit de rajouter les crochets :

```
> S :=x, y, z, alpha ;
```

```
S := x, y, z, α
```

```
>L :=[S] ;
```

```
L := [x, y, z, α]
```

Pour transformer une liste en une séquence, il faut enlever les crochets par la commande `op` :

```
> S :=op(L) ;
```

```
S := 1,5,a,8
```

Pour transformer une liste en une séquence, il suffit de rajouter les crochets :

```
> S :=x,y,z,alpha ;
```

```
S := x,y,z,α
```

```
>L :=[S] ;
```

```
L := [x,y,z,α]
```

Pour transformer une liste en une séquence, il faut enlever les crochets par la commande `op` :

```
> S :=op(L) ;
```

```
S := 1, 5, a, 8
```

Pour transformer une liste en une séquence, il suffit de rajouter les crochets :

```
> S :=x, y, z, alpha ;
```

```
S := x, y, z, α
```

```
>L :=[S] ;
```

```
L := [x, y, z, α]
```

Pour transformer une liste en une séquence, il faut enlever les crochets par la commande `op` :

```
> S :=op(L) ;
```

**S := 1, 5, a, 8**

Pour transformer une liste en une séquence, il suffit de rajouter les crochets :

```
> S :=x, y, z, alpha ;
```

**S := x, y, z, α**

```
>L :=[S] ;
```

**L := [x, y, z, α]**

Pour transformer une liste en une séquence, il faut enlever les crochets par la commande `op` :

```
> S :=op(L) ;
```

`S := 1, 5, a, 8`

Pour transformer une liste en une séquence, il suffit de rajouter les crochets :

```
> S :=x, y, z, alpha ;
```

`S := x, y, z, α`

```
>L :=[S] ;
```

`L := [x, y, z, α]`

Pour transformer une liste en une séquence, il faut enlever les crochets par la commande `op` :

```
> S :=op(L) ;
```

```
S := 1,5,a,8
```

Pour transformer une liste en une séquence, il suffit de rajouter les crochets :

```
> S :=x,y,z,alpha ;
```

```
S := x,y,z,α
```

```
>L :=[S] ;
```

```
L := [x,y,z,α]
```

Pour transformer une liste en une séquence, il faut enlever les crochets par la commande `op` :

```
> S :=op(L) ;
```

`S := 1, 5, a, 8`

Pour transformer une liste en une séquence, il suffit de rajouter les crochets :

```
> S :=x, y, z, alpha ;
```

`S := x, y, z, α`

```
>L :=[S] ;
```

`L := [x, y, z, α]`

Pour transformer une liste en une séquence, il faut enlever les crochets par la commande `op` :

```
> S :=op(L) ;
```

**S := 1, 5, a, 8**

Pour transformer une liste en une séquence, il suffit de rajouter les crochets :

```
> S :=x, y, z, alpha ;
```

**S := x, y, z, α**

```
>L :=[S] ;
```

**L := [x, y, z, α]**

Pour transformer une liste en une séquence, il faut enlever les crochets par la commande `op` :

```
> S :=op(L) ;
```

$S := 1, 5, a, 8$

Pour transformer une liste en une séquence, il suffit de rajouter les crochets :

```
> S :=x, y, z, alpha ;
```

$S := x, y, z, \alpha$

```
>L :=[S] ;
```

$L := [x, y, z, \alpha]$

Pour transformer une liste en une séquence, il faut enlever les crochets par la commande `op` :

```
> S :=op(L) ;
```

$S := 1, 5, a, 8$

Pour transformer une liste en une séquence, il suffit de rajouter les crochets :

```
> S :=x, y, z, alpha ;
```

$S := x, y, z, \alpha$

```
>L :=[S] ;
```

$L := [x, y, z, \alpha]$

La **liste vide** se note par des crochets vides "[]" :

```
> M := [] ;
```

```
M := []
```

Le nombre d'éléments de la liste est obtenu par la  
commande `nops` :

```
> L; # Pour afficher la valeur de L
```

```
[x, y, z, a]
```

```
> nops(L) ;
```

```
4
```

La **liste vide** se note par des crochets vides "[]" :

```
> M := [] ;
```

```
M := []
```

Le nombre d'éléments de la liste est obtenu par la  
commande `nops` :

```
> L; # Pour afficher la valeur de L
```

```
[x, y, z, a]
```

```
> nops(L) ;
```

```
4
```

La **liste vide** se note par des crochets vides "[]" :

```
> M := [] ;
```

$$M := []$$

Le nombre d'éléments de la liste est obtenu par la  
commande `nops` :

```
> L; # Pour afficher la valeur de L
```

$$[x, y, z, \alpha]$$

```
> nops(L) ;
```

4

La **liste vide** se note par des crochets vides "[]" :

```
> M := [] ;
```

$$M := []$$

Le nombre d'éléments de la liste est obtenu par la  
commande `nops` :

```
> L; # Pour afficher la valeur de L
```

$$[x, y, z, a]$$

```
> nops(L) ;
```

4

La **liste vide** se note par des crochets vides "[]" :

```
> M := [] ;
```

$$M := []$$

Le nombre d'éléments de la liste est obtenu par la  
commande `nops` :

```
> L ; # Pour afficher la valeur de L
```

$$[x, y, z, \alpha]$$

```
> nops(L) ;
```

4

La **liste vide** se note par des crochets vides "[]" :

```
> M := [] ;
```

$$M := []$$

Le nombre d'éléments de la liste est obtenu par la  
commande `nops` :

```
> L ; # Pour afficher la valeur de L
```

$$[x, y, z, \alpha]$$

```
> nops(L) ;
```

4

La **liste vide** se note par des crochets vides "[]" :

```
> M := [] ;
```

$$M := []$$

Le nombre d'éléments de la liste est obtenu par la  
commande `nops` :

```
> L ; # Pour afficher la valeur de L
```

$$[x, y, z, \alpha]$$

```
> nops(L) ;
```

4

La **liste vide** se note par des crochets vides "[]" :

```
> M := [] ;
```

$$M := []$$

Le nombre d'éléments de la liste est obtenu par la  
commande `nops` :

```
> L ; # Pour afficher la valeur de L
```

$$[x, y, z, \alpha]$$

```
> nops(L) ;
```

4

La **liste vide** se note par des crochets vides "[]" :

```
> M := [] ;
```

$$M := []$$

Le nombre d'éléments de la liste est obtenu par la  
commande `nops` :

```
> L ; # Pour afficher la valeur de L
```

$$[x, y, z, \alpha]$$

```
> nops(L) ;
```

4

Pour accéder à un élément dans une liste il y a deux façons de procéder. Considérons la liste  $L$  précédente. Pour sélectionner par exemple son deuxième élément :

```
1 > L[2] ;
```

$y$

```
2 > op(2, L) ;
```

$y$

On peut alors changer la valeur d'un élément dans une liste. Par exemple,

```
> L[3] := a ;
```

```
> L ;
```

$[x, y, a, \alpha]$

Pour accéder à un élément dans une liste il y a deux façons de procéder. Considérons la liste  $L$  précédente. Pour sélectionner par exemple son deuxième élément :

```
1 > L[2] ;
```

$y$

```
2 > op(2, L) ;
```

$y$

On peut alors changer la valeur d'un élément dans une liste.  
Par exemple,

```
> L[3] := a ;
```

```
> L ;
```

$[x, y, a, \alpha]$

Pour accéder à un élément dans une liste il y a deux façons de procéder. Considérons la liste  $L$  précédente. Pour sélectionner par exemple son deuxième élément :

1 >  $L[2]$  ;

$y$

2 >  $op(2, L)$  ;

$y$

On peut alors changer la valeur d'un élément dans une liste.  
Par exemple,

>  $L[3] := a$  ;

>  $L$  ;

$[x, y, a, \alpha]$

Pour accéder à un élément dans une liste il y a deux façons de procéder. Considérons la liste  $L$  précédente. Pour sélectionner par exemple son deuxième élément :

**1** >  $L[2]$  ;

$y$

**2** >  $op(2, L)$  ;

$y$

On peut alors changer la valeur d'un élément dans une liste.  
Par exemple,

>  $L[3] := a$  ;

>  $L$  ;

$[x, y, a, \alpha]$

Pour accéder à un élément dans une liste il y a deux façons de procéder. Considérons la liste  $L$  précédente. Pour sélectionner par exemple son deuxième élément :

**1** >  $L[2]$  ;

$y$

**2** >  $op(2, L)$  ;

$y$

On peut alors changer la valeur d'un élément dans une liste.  
Par exemple,

>  $L[3] := a$  ;

>  $L$  ;

$[x, y, a, \alpha]$

Pour accéder à un élément dans une liste il y a deux façons de procéder. Considérons la liste  $L$  précédente. Pour sélectionner par exemple son deuxième élément :

**1** >  $L[2]$  ;

$y$

**2** >  $op(2, L)$  ;

$y$

On peut alors changer la valeur d'un élément dans une liste.  
Par exemple,

>  $L[3] := a$  ;

>  $L$  ;

$[x, y, a, \alpha]$

Pour accéder à un élément dans une liste il y a deux façons de procéder. Considérons la liste  $L$  précédente. Pour sélectionner par exemple son deuxième élément :

**1** >  $L[2]$  ;

$y$

**2** >  $op(2, L)$  ;

$y$

On peut alors changer la valeur d'un élément dans une liste.

Par exemple,

>  $L[3] := a$  ;

>  $L$  ;

$[x, y, a, \alpha]$

Pour accéder à un élément dans une liste il y a deux façons de procéder. Considérons la liste  $L$  précédente. Pour sélectionner par exemple son deuxième élément :

**1** >  $L[2]$  ;

$y$

**2** >  $op(2, L)$  ;

$y$

On peut alors changer la valeur d'un élément dans une liste.  
Par exemple,

>  $L[3] := a$  ;

>  $L$  ;

$[x, y, a, \alpha]$

Pour accéder à un élément dans une liste il y a deux façons de procéder. Considérons la liste  $L$  précédente. Pour sélectionner par exemple son deuxième élément :

**1** >  $L[2]$  ;

$y$

**2** >  $op(2, L)$  ;

$y$

On peut alors changer la valeur d'un élément dans une liste.  
Par exemple,

>  $L[3] := a$  :

>  $L$  ;

$[x, y, a, \alpha]$

Pour accéder à un élément dans une liste il y a deux façons de procéder. Considérons la liste  $L$  précédente. Pour sélectionner par exemple son deuxième élément :

**1** >  $L[2]$  ;

$y$

**2** >  $op(2, L)$  ;

$y$

On peut alors changer la valeur d'un élément dans une liste.  
Par exemple,

>  $L[3] := a$  ;

>  $L$  ;

$[x, y, a, \alpha]$

Pour accéder à un élément dans une liste il y a deux façons de procéder. Considérons la liste  $L$  précédente. Pour sélectionner par exemple son deuxième élément :

**1** >  $L[2]$  ;

$y$

**2** >  $op(2, L)$  ;

$y$

On peut alors changer la valeur d'un élément dans une liste.  
Par exemple,

>  $L[3] := a$  ;

>  $L$  ;

$[x, y, a, \alpha]$

Pour transformer une liste en *somme* ou *produit*, on utilise la commande `convert` :

```
> M := [2, 8, 4, x, z^2] ;
```

$$M := [2, 8, 4, x, z^2]$$

```
> convert (M, '+' ) ;
```

$$14 + x + z^2$$

```
> convert (M, '*') ;
```

$$64xz^2$$

Notez bien que le caractère "+" ou "\*" doit être entouré par des accents graves ""

Pour transformer une liste en *somme* ou *produit*, on utilise la commande `convert` :

```
> M := [2, 8, 4, x, z^2] ;
```

$$M := [2, 8, 4, x, z^2]$$

```
> convert(M, '+' ) ;
```

$$14 + x + z^2$$

```
> convert(M, '*') ;
```

$$64xz^2$$

Notez bien que le caractère "+" ou "\*" doit être entouré par des accents graves ""

Pour transformer une liste en *somme* ou *produit*, on utilise la commande `convert` :

```
> M := [2, 8, 4, x, z^2] ;
```

$$M := [2, 8, 4, x, z^2]$$

```
> convert(M, '+' ) ;
```

$$14 + x + z^2$$

```
> convert(M, '*') ;
```

$$64xz^2$$

Notez bien que le caractère "+" ou "\*" doit être entouré par des accents graves "".

Pour transformer une liste en *somme* ou *produit*, on utilise la commande `convert` :

```
> M := [2, 8, 4, x, z^2] ;
```

$$M := [2, 8, 4, x, z^2]$$

```
> convert (M, '+' ) ;
```

$$14 + x + z^2$$

```
> convert (M, '*') ;
```

$$64xz^2$$

Notez bien que le caractère "+" ou "\*" doit être entouré par des accents graves ""

Pour transformer une liste en *somme* ou *produit*, on utilise la commande `convert` :

```
> M := [2, 8, 4, x, z^2] ;
```

$$M := [2, 8, 4, x, z^2]$$

```
> convert (M, '+' ) ;
```

$$14 + x + z^2$$

```
> convert (M, '*') ;
```

$$64xz^2$$

Notez bien que le caractère "+" ou "\*" doit être entouré par des accents graves "".

Pour transformer une liste en *somme* ou *produit*, on utilise la commande `convert` :

```
> M := [2, 8, 4, x, z^2] ;
```

$$M := [2, 8, 4, x, z^2]$$

```
> convert (M, '+' ) ;
```

$$14 + x + z^2$$

```
> convert (M, '*') ;
```

$$64xz^2$$

Notez bien que le caractère "+" ou "\*" doit être entouré par des accents graves ""

Pour transformer une liste en *somme* ou *produit*, on utilise la commande `convert` :

```
> M := [2, 8, 4, x, z^2] ;
```

$$M := [2, 8, 4, x, z^2]$$

```
> convert (M, '+' ) ;
```

$$14 + x + z^2$$

```
> convert (M, '*') ;
```

$$64xz^2$$

Notez bien que le caractère "+" ou "\*" doit être entouré par des accents graves ""

Pour transformer une liste en *somme* ou *produit*, on utilise la commande `convert` :

```
> M := [2, 8, 4, x, z^2] ;
```

$$M := [2, 8, 4, x, z^2]$$

```
> convert (M, '+' ) ;
```

$$14 + x + z^2$$

```
> convert (M, '*') ;
```

$$64xz^2$$

Notez bien que le caractère "+" ou "\*" doit être entouré par des accents graves "".

On peut **sélectionner** (`select`) ou **supprimer** certains éléments d'une liste :

```
> L := [seq(i, i=5..10)] ;
```

```
L := [5, 6, 7, 8, 9, 10]
```

Pour sélectionner les nombres premiers de cette liste, on utilise `select` avec la commande `isprime` (définie par `isprime (n)` est `true` si `n` est premier, et `false` dans le cas contraire) :

```
> select (isprime, L) ;
```

```
[5, 7]
```

On peut **sélectionner** (`select`) ou **supprimer** certains éléments d'une liste :

```
>L :=[seq(i,i=5..10)] ;
```

```
L := [5,6,7,8,9,10]
```

Pour sélectionner les nombres premiers de cette liste, on utilise `select` avec la commande `isprime` (définie par `isprime (n)` est `true` si `n` est premier, et `false` dans le cas contraire) :

```
> select (isprime,L) ;
```

```
[5,7]
```

On peut **sélectionner** (`select`) ou **supprimer** certains éléments d'une liste :

```
>L :=[seq(i,i=5..10)] ;
```

```
L := [5, 6, 7, 8, 9, 10]
```

Pour sélectionner les nombres premiers de cette liste, on utilise `select` avec la commande `isprime` (définie par `isprime (n)` est `true` si `n` est premier, et `false` dans le cas contraire) :

```
> select (isprime,L) ;
```

```
[5, 7]
```

On peut **sélectionner** (`select`) ou **supprimer** certains éléments d'une liste :

```
> L := [seq(i, i=5..10)] ;
```

$$L := [5, 6, 7, 8, 9, 10]$$

Pour sélectionner les nombres premiers de cette liste, on utilise `select` avec la commande `isprime` (définie par `isprime (n)` est `true` si `n` est premier, et `false` dans le cas contraire) :

```
> select (isprime, L) ;
```

$$[5, 7]$$

On peut **sélectionner** (`select`) ou **supprimer** certains éléments d'une liste :

```
> L := [seq(i, i=5..10)] ;
```

$$L := [5, 6, 7, 8, 9, 10]$$

Pour sélectionner les nombres premiers de cette liste, on utilise `select` avec la commande `isprime` (définie par `isprime (n)` est `true` si `n` est premier, et `false` dans le cas contraire) :

```
> select(isprime, L) ;
```

$$[5, 7]$$

On peut **sélectionner** (`select`) ou **supprimer** certains éléments d'une liste :

```
>L :=[seq(i,i=5..10)] ;
```

$$L := [5, 6, 7, 8, 9, 10]$$

Pour sélectionner les nombres premiers de cette liste, on utilise `select` avec la commande `isprime` (définie par `isprime (n)` est `true` si `n` est premier, et `false` dans le cas contraire) :

```
> select (isprime,L) ;
```

$$[5, 7]$$

On peut **sélectionner** (`select`) ou **supprimer** certains éléments d'une liste :

```
>L :=[seq(i,i=5..10)] ;
```

$$L := [5, 6, 7, 8, 9, 10]$$

Pour sélectionner les nombres premiers de cette liste, on utilise `select` avec la commande `isprime` (définie par `isprime (n)` est `true` si `n` est premier, et `false` dans le cas contraire) :

```
> select (isprime,L) ;
```

$$[5, 7]$$

Attention, la liste  $L$  n'a pas été modifiée !

```
> L ;
```

```
L := [5, 6, 7, 8, 9, 10]
```

Pour la modifier on doit écrire :

```
> L :=select (isprime, L) ;
```

```
L := [5, 7]
```

Attention, la liste  $L$  n'a pas été modifiée !

```
> L ;
```

```
L := [5, 6, 7, 8, 9, 10]
```

Pour la modifier on doit écrire :

```
> L :=select(isprime,L) ;
```

```
L := [5, 7]
```

Attention, la liste  $L$  n'a pas été modifiée !

```
> L ;
```

```
L := [5, 6, 7, 8, 9, 10]
```

Pour la modifier on doit écrire :

```
> L :=select(isprime,L) ;
```

```
L := [5, 7]
```

Attention, la liste  $L$  n'a pas été modifiée !

```
> L ;
```

```
L := [5, 6, 7, 8, 9, 10]
```

Pour la modifier on doit écrire :

```
> L :=select(isprime,L) ;
```

```
L := [5, 7]
```

Attention, la liste  $L$  n'a pas été modifiée !

```
> L ;
```

```
L := [5, 6, 7, 8, 9, 10]
```

Pour la modifier on doit écrire :

```
> L :=select(isprime,L) ;
```

```
L := [5, 7]
```

Attention, la liste  $L$  n'a pas été modifiée !

```
> L ;
```

```
L := [5, 6, 7, 8, 9, 10]
```

Pour la modifier on doit écrire :

```
> L :=select(isprime,L) ;
```

```
L := [5, 7]
```

Attention, la liste  $L$  n'a pas été modifiée !

```
> L ;
```

```
L := [5, 6, 7, 8, 9, 10]
```

Pour la modifier on doit écrire :

```
> L :=select(isprime,L) ;
```

```
L := [5, 7]
```

Maintenant pour supprimer les éléments d'une liste, on utilise la commande `remove` :

```
> L := [seq(i, i=5..10)] :
```

```
> remove(isprime, L) ;
```

```
[6, 8, 9, 10]
```

```
> L ;
```

```
[5, 6, 7, 8, 9, 10]
```

```
> L := remove(isprime, L) :
```

```
> L ;
```

```
[6, 8, 9, 10]
```

Maintenant pour supprimer les éléments d'une liste, on utilise la commande `remove` :

```
> L := [seq(i, i=5..10)] :
```

```
> remove(isprime, L) ;
```

```
[6, 8, 9, 10]
```

```
> L ;
```

```
[5, 6, 7, 8, 9, 10]
```

```
> L := remove(isprime, L) :
```

```
> L ;
```

```
[6, 8, 9, 10]
```

Maintenant pour supprimer les éléments d'une liste, on utilise la commande `remove` :

```
> L := [seq(i, i=5..10)] :  
> remove(isprime, L) ;
```

```
[6, 8, 9, 10]
```

```
> L ;
```

```
[5, 6, 7, 8, 9, 10]
```

```
> L := remove(isprime, L) :
```

```
> L ;
```

```
[6, 8, 9, 10]
```

Maintenant pour supprimer les éléments d'une liste, on utilise la commande `remove` :

```
> L := [seq(i, i=5..10)] :  
> remove(isprime, L) ;
```

```
[6, 8, 9, 10]
```

```
> L ;
```

```
[5, 6, 7, 8, 9, 10]
```

```
> L := remove(isprime, L) :
```

```
> L ;
```

```
[6, 8, 9, 10]
```

Maintenant pour supprimer les éléments d'une liste, on utilise la commande `remove` :

```
> L := [seq(i, i=5..10)] :  
> remove(isprime, L) ;
```

```
[6, 8, 9, 10]
```

```
> L ;
```

```
[5, 6, 7, 8, 9, 10]
```

```
> L := remove(isprime, L) :
```

```
> L ;
```

```
[6, 8, 9, 10]
```

Maintenant pour supprimer les éléments d'une liste, on utilise la commande `remove` :

```
> L := [seq(i, i=5..10)] :  
> remove(isprime, L) ;
```

[6, 8, 9, 10]

```
> L ;
```

[5, 6, 7, 8, 9, 10]

```
> L := remove(isprime, L) :
```

```
> L ;
```

[6, 8, 9, 10]

Maintenant pour supprimer les éléments d'une liste, on utilise la commande `remove` :

```
> L := [seq(i, i=5..10)] :  
> remove(isprime, L) ;
```

[6, 8, 9, 10]

```
> L ;
```

[5, 6, 7, 8, 9, 10]

```
> L := remove(isprime, L) :
```

```
> L ;
```

[6, 8, 9, 10]

Maintenant pour supprimer les éléments d'une liste, on utilise la commande `remove` :

```
> L := [seq(i, i=5..10)] :  
> remove(isprime, L) ;
```

[6, 8, 9, 10]

```
> L ;
```

[5, 6, 7, 8, 9, 10]

```
> L := remove(isprime, L) :
```

```
> L ;
```

[6, 8, 9, 10]

Maintenant pour supprimer les éléments d'une liste, on utilise la commande `remove` :

```
> L := [seq(i, i=5..10)] :  
> remove(isprime, L) ;
```

[6, 8, 9, 10]

```
> L ;
```

[5, 6, 7, 8, 9, 10]

```
> L := remove(isprime, L) :
```

```
> L ;
```

[6, 8, 9, 10]

Plus généralement, si  $f : x \mapsto f(x)$  est une **fonction booléenne**, c'est-à-dire  $f(x) \in \{\text{true}, \text{false}\}$ , on peut employer les commandes `select(L, f)` et `remove(L, f)` :

- 1 `select(f, L)` permet de sélectionner les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$  ;
- 2 `remove(f, L)` permet de supprimer les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$ .

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit  $P$  une certaine condition, alors `evalb(P)` renvoie `true` lorsque  $P$  est vraie, et renvoie `false` si  $P$  est fausse.

```
> evalb(3<2) ;  
  
false
```

```
>evalb(0=0) ;  
  
true
```

Plus généralement, si  $f : x \mapsto f(x)$  est une **fonction booléenne**, c'est-à-dire  $f(x) \in \{\text{true}, \text{false}\}$ , on peut employer les commandes `select(L, f)` et `remove(L, f)` :

- 1 `select(f, L)` permet de sélectionner les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$  ;
- 2 `remove(f, L)` permet de supprimer les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$ .

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit  $P$  une certaine condition, alors `evalb(P)` renvoie `true` lorsque  $P$  est vraie, et renvoie `false` si  $P$  est fausse.

```
> evalb(3<2) ;
```

*false*

```
>evalb(0=0) ;
```

*true*

Plus généralement, si  $f : x \mapsto f(x)$  est une **fonction booléenne**, c'est-à-dire  $f(x) \in \{\text{true}, \text{false}\}$ , on peut employer les commandes `select (L, f)` et `remove (L, f)` :

- 1 `select (f, L)` permet de sélectionner les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$  ;
- 2 `remove (f, L)` permet de supprimer les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$ .

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit  $P$  une certaine condition, alors `evalb(P)` renvoie `true` lorsque  $P$  est vraie, et renvoie `false` si  $P$  est fausse.

```
> evalb(3<2) ;
```

*false*

```
>evalb(0=0) ;
```

*true*

Plus généralement, si  $f : x \mapsto f(x)$  est une **fonction booléenne**, c'est-à-dire  $f(x) \in \{\text{true}, \text{false}\}$ , on peut employer les commandes `select (L, f)` et `remove (L, f)` :

- 1 `select (f, L)` permet de sélectionner les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$  ;
- 2 `remove (f, L)` permet de supprimer les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$ .

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit  $P$  une certaine condition, alors `evalb(P)` renvoie `true` lorsque  $P$  est vraie, et renvoie `false` si  $P$  est fausse.

```
> evalb(3<2) ;
```

*false*

```
>evalb(0=0) ;
```

*true*

Plus généralement, si  $f : x \mapsto f(x)$  est une **fonction booléenne**, c'est-à-dire  $f(x) \in \{\text{true}, \text{false}\}$ , on peut employer les commandes `select(L, f)` et `remove(L, f)` :

- 1 `select(f, L)` permet de sélectionner les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$  ;
- 2 `remove(f, L)` permet de supprimer les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$ .

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit  $P$  une certaine condition, alors `evalb(P)` renvoie `true` lorsque  $P$  est vraie, et renvoie `false` si  $P$  est fausse.

```
> evalb(3<2) ;
```

*false*

```
>evalb(0=0) ;
```

*true*

Plus généralement, si  $f : x \mapsto f(x)$  est une **fonction booléenne**, c'est-à-dire  $f(x) \in \{\text{true}, \text{false}\}$ , on peut employer les commandes `select(L, f)` et `remove(L, f)` :

- 1 `select(f, L)` permet de sélectionner les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$  ;
- 2 `remove(f, L)` permet de supprimer les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$ .

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit  $P$  une certaine condition, alors `evalb(P)` renvoie `true` lorsque  $P$  est vraie, et renvoie `false` si  $P$  est fausse.

```
> evalb(3<2) ;  
  
false
```

```
>evalb(0=0) ;  
  
true
```

Plus généralement, si  $f : x \mapsto f(x)$  est une **fonction booléenne**, c'est-à-dire  $f(x) \in \{\text{true}, \text{false}\}$ , on peut employer les commandes `select(L, f)` et `remove(L, f)` :

- 1 `select(f, L)` permet de sélectionner les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$  ;
- 2 `remove(f, L)` permet de supprimer les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$ .

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit  $P$  une certaine condition, alors `evalb(P)` renvoie `true` lorsque  $P$  est vraie, et renvoie `false` si  $P$  est fausse.

```
> evalb(3<2) ;
```

*false*

```
>evalb(0=0) ;
```

*true*

Plus généralement, si  $f : x \mapsto f(x)$  est une **fonction booléenne**, c'est-à-dire  $f(x) \in \{\text{true}, \text{false}\}$ , on peut employer les commandes `select(L, f)` et `remove(L, f)` :

- 1 `select(f, L)` permet de sélectionner les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$  ;
- 2 `remove(f, L)` permet de supprimer les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$ .

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit  $P$  une certaine condition, alors `evalb(P)` renvoie `true` lorsque  $P$  est vraie, et renvoie `false` si  $P$  est fausse.

```
> evalb(3<2) ;
```

*false*

```
>evalb(0=0) ;
```

*true*

Plus généralement, si  $f : x \mapsto f(x)$  est une **fonction booléenne**, c'est-à-dire  $f(x) \in \{\text{true}, \text{false}\}$ , on peut employer les commandes `select(L, f)` et `remove(L, f)` :

- 1 `select(f, L)` permet de sélectionner les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$  ;
- 2 `remove(f, L)` permet de supprimer les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$ .

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit  $P$  une certaine condition, alors `evalb(P)` renvoie `true` lorsque  $P$  est vraie, et renvoie `false` si  $P$  est fausse.

```
> evalb(3<2) ;
```

*false*

```
>evalb(0=0) ;
```

*true*

Plus généralement, si  $f : x \mapsto f(x)$  est une **fonction booléenne**, c'est-à-dire  $f(x) \in \{\text{true}, \text{false}\}$ , on peut employer les commandes `select(L, f)` et `remove(L, f)` :

- 1 `select(f, L)` permet de sélectionner les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$  ;
- 2 `remove(f, L)` permet de supprimer les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$ .

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit  $P$  une certaine condition, alors `evalb(P)` renvoie `true` lorsque  $P$  est vraie, et renvoie `false` si  $P$  est fausse.

```
> evalb(3<2) ;
```

*false*

```
>evalb(0=0) ;
```

*true*

Plus généralement, si  $f : x \mapsto f(x)$  est une **fonction booléenne**, c'est-à-dire  $f(x) \in \{\text{true}, \text{false}\}$ , on peut employer les commandes `select(L, f)` et `remove(L, f)` :

- 1 `select(f, L)` permet de sélectionner les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$  ;
- 2 `remove(f, L)` permet de supprimer les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$ .

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit  $P$  une certaine condition, alors `evalb(P)` renvoie `true` lorsque  $P$  est vraie, et renvoie `false` si  $P$  est fausse.

```
> evalb(3<2) ;
```

*false*

```
>evalb(0=0) ;
```

*true*

Plus généralement, si  $f : x \mapsto f(x)$  est une **fonction booléenne**, c'est-à-dire  $f(x) \in \{\text{true}, \text{false}\}$ , on peut employer les commandes `select(L, f)` et `remove(L, f)` :

- 1 `select(f, L)` permet de sélectionner les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$  ;
- 2 `remove(f, L)` permet de supprimer les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$ .

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit  $P$  une certaine condition, alors `evalb(P)` renvoie `true` lorsque  $P$  est vraie, et renvoie `false` si  $P$  est fausse.

```
> evalb(3<2) ;
```

*false*

```
>evalb(0=0) ;
```

*true*

Plus généralement, si  $f : x \mapsto f(x)$  est une **fonction booléenne**, c'est-à-dire  $f(x) \in \{\text{true}, \text{false}\}$ , on peut employer les commandes `select(L, f)` et `remove(L, f)` :

- 1 `select(f, L)` permet de sélectionner les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$  ;
- 2 `remove(f, L)` permet de supprimer les éléments  $L[i]$  de la liste  $L$  tels que  $f(L[i]) = \text{true}$ .

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit  $P$  une certaine condition, alors `evalb(P)` renvoie `true` lorsque  $P$  est vraie, et renvoie `false` si  $P$  est fausse.

```
> evalb(3<2) ;
```

*false*

```
>evalb(0=0) ;
```

*true*

# Exemple

Soit la liste :

```
> L := [seq(i, i=-5..10)];
```

```
L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x >= 0);
```

*positif* :=  $x \rightarrow \text{evalb}(0 \leq x)$

```
> positif(-3);
```

*false*

```
> positif(4);
```

*true*

# Exemple

Soit la liste :

```
> L := [seq(i, i=-5..10)] ;
```

```
      L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x >= 0) ;
```

```
      positif := x → evalb(0 ≤ x)
```

```
> positif(-3) ;
```

```
      false
```

```
> positif(4) ;
```

```
      true
```

# Exemple

Soit la liste :

```
> L := [seq(i, i=-5..10)] ;
```

```
L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x >= 0) ;
```

```
positif := x → evalb(0 ≤ x)
```

```
> positif(-3) ;
```

```
false
```

```
> positif(4) ;
```

```
true
```

# Exemple

Soit la liste :

```
> L := [seq(i, i=-5..10)] ;
```

```
      L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x >= 0) ;
```

```
      positif := x -> evalb(0 ≤ x)
```

```
> positif(-3) ;
```

```
      false
```

```
> positif(4) ;
```

```
      true
```

# Exemple

Soit la liste :

```
> L := [seq(i, i=-5..10)] ;
```

```
L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x >= 0) ;
```

```
positif := x → evalb(0 ≤ x)
```

```
> positif(-3) ;
```

```
false
```

```
> positif(4) ;
```

```
true
```

# Exemple

Soit la liste :

```
> L := [seq(i, i=-5..10)] ;
```

```
      L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x >= 0) ;
```

```
      positif := x -> evalb(0 ≤ x)
```

```
> positif(-3) ;
```

```
      false
```

```
> positif(4) ;
```

```
      true
```

# Exemple

Soit la liste :

```
> L := [seq(i, i=-5..10)] ;
```

```
L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x >= 0) ;
```

```
positif := x → evalb(0 ≤ x)
```

```
> positif(-3) ;
```

```
false
```

```
> positif(4) ;
```

```
true
```

# Exemple

Soit la liste :

```
> L := [seq(i, i=-5..10)] ;
```

```
L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x >= 0) ;
```

```
positif := x → evalb(0 ≤ x)
```

```
> positif(-3) ;
```

```
false
```

```
> positif(4) ;
```

```
true
```

# Exemple

Soit la liste :

```
> L := [seq(i, i=-5..10)] ;
```

```
L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x >= 0) ;
```

```
positif := x → evalb(0 ≤ x)
```

```
> positif(-3) ;
```

```
false
```

```
> positif(4) ;
```

```
true
```

# Exemple

Soit la liste :

```
> L := [seq(i, i=-5..10)] ;
```

```
L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x >= 0) ;
```

$$\textit{positif} := x \rightarrow \textit{evalb}(0 \leq x)$$

```
> positif(-3) ;
```

*false*

```
> positif(4) ;
```

*true*

# Exemple

Soit la liste :

```
> L := [seq(i, i=-5..10)] ;
```

$$L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x >= 0) ;
```

$$\textit{positif} := x \rightarrow \textit{evalb}(0 \leq x)$$

```
> positif(-3) ;
```

*false*

```
> positif(4) ;
```

*true*

# Exemple

Soit la liste :

```
> L := [seq(i, i=-5..10)] ;
```

$$L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x >= 0) ;
```

$$\textit{positif} := x \rightarrow \textit{evalb}(0 \leq x)$$

```
> positif(-3) ;
```

*false*

```
> positif(4) ;
```

*true*

# Exemple

Soit la liste :

```
> L := [seq(i, i=-5..10)] ;
```

```
L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x >= 0) ;
```

$$\textit{positif} := x \rightarrow \textit{evalb}(0 \leq x)$$

```
> positif(-3) ;
```

*false*

```
> positif(4) ;
```

*true*

# Exemple

Soit la liste :

```
> L := [seq(i, i=-5..10)] ;
```

```
L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x >= 0) ;
```

$$\textit{positif} := x \rightarrow \textit{evalb}(0 \leq x)$$

```
> positif(-3) ;
```

*false*

```
> positif(4) ;
```

*true*

On peut maintenant procéder à la suppression des nombres négatifs de  $L$  :

```
> L ;
```

```
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
> remove(positif, L) ;
```

```
[-5, -4, -3, -2, -1]
```

On peut maintenant procéder à la suppression des nombres négatifs de  $L$  :

```
> L ;
```

```
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
> remove(positif, L) ;
```

```
[-5, -4, -3, -2, -1]
```

On peut maintenant procéder à la suppression des nombres négatifs de  $L$  :

```
> L ;
```

```
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
> remove(positif, L) ;
```

```
[-5, -4, -3, -2, -1]
```

On peut maintenant procéder à la suppression des nombres négatifs de  $L$  :

```
> L ;
```

```
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
> remove(positif, L) ;
```

```
[-5, -4, -3, -2, -1]
```

On peut maintenant procéder à la suppression des nombres négatifs de  $L$  :

```
> L ;
```

```
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
> remove(positif, L) ;
```

```
[-5, -4, -3, -2, -1]
```

# Fonction `map`

Chap. III :  
Les objets  
structurés

Laurent  
Poinot

Les objets  
structurés

Quelques  
exercices

On peut appliquer une même fonction à tous les éléments d'une liste à l'aide de la commande `map` :

```
> succ := x -> x + 1; # Fonction successeur
```

$$SUCC := x \rightarrow x + 1$$

```
> L := [3, -1, 0, 5] :
```

```
> map(succ, L) :
```

```
[4, 0, 1, 6]
```

# Fonction `map`

Chap. III :  
Les objets  
structurés

Laurent  
Poinot

Les objets  
structurés

Quelques  
exercices

On peut appliquer une même fonction à tous les éléments d'une liste à l'aide de la commande `map` :

```
> succ := x -> x + 1; # Fonction successeur
```

$$\text{SUCC} := x \rightarrow x + 1$$

```
> L := [3, -1, 0, 5] :
```

```
> map(succ, L) :
```

$$[4, 0, 1, 6]$$

# Fonction `map`

Chap. III :  
Les objets  
structurés

Laurent  
Poinot

Les objets  
structurés

Quelques  
exercices

On peut appliquer une même fonction à tous les éléments d'une liste à l'aide de la commande `map` :

```
> succ := x -> x + 1; # Fonction successeur
```

$$SUCC := x \rightarrow x + 1$$

```
> L := [3, -1, 0, 5] :
```

```
> map(succ, L) ;
```

```
[4, 0, 1, 6]
```

# Fonction `map`

Chap. III :  
Les objets  
structurés

Laurent  
Poinsot

Les objets  
structurés

Quelques  
exercices

On peut appliquer une même fonction à tous les éléments d'une liste à l'aide de la commande `map` :

```
> succ := x -> x + 1; # Fonction successeur
```

$$SUCC := x \rightarrow x + 1$$

```
> L := [3, -1, 0, 5] :
```

```
> map(succ, L) ;
```

```
[4, 0, 1, 6]
```

# Fonction `map`

Chap. III :  
Les objets  
structurés

Laurent  
Poinsot

Les objets  
structurés

Quelques  
exercices

On peut appliquer une même fonction à tous les éléments d'une liste à l'aide de la commande `map` :

```
> succ := x -> x + 1; # Fonction successeur
```

$$SUCC := x \rightarrow x + 1$$

```
> L := [3, -1, 0, 5] :
```

```
> map(succ, L) ;
```

```
[4, 0, 1, 6]
```

# Fonction `map`

Chap. III :  
Les objets  
structurés

Laurent  
Poinsot

Les objets  
structurés

Quelques  
exercices

On peut appliquer une même fonction à tous les éléments d'une liste à l'aide de la commande `map` :

```
> succ := x -> x + 1; # Fonction successeur
```

$$SUCC := x \rightarrow x + 1$$

```
> L := [3, -1, 0, 5] :
```

```
> map(succ, L) ;
```

[4, 0, 1, 6]

Pour **ajouter un élément à la fin d'une liste** : On convertit la liste en une séquence par la commande `op`. Puis on concatène la séquence obtenue avec le nouvel élément grâce à la virgule ",". Enfin on transforme cette séquence en une liste en l'entourant des crochets :

```
> L := [seq(i, i=1..9)] ;
```

```
L := [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
> L := [op(L), 10] ;
```

```
L := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Pour **ajouter un élément à la fin d'une liste** : On convertit la liste en une séquence par la commande `op`. Puis on concatène la séquence obtenue avec le nouvel élément grâce à la virgule ",". Enfin on transforme cette séquence en une liste en l'entourant des crochets :

```
> L := [seq(i, i=1..9)] ;
```

```
L := [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
> L := [op(L), 10] ;
```

```
L := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Pour **ajouter un élément à la fin d'une liste** : On convertit la liste en une séquence par la commande `op`. Puis on concatène la séquence obtenue avec le nouvel élément grâce à la virgule `,`. Enfin on transforme cette séquence en une liste en l'entourant des crochets :

```
> L := [seq(i, i=1..9)] ;
```

```
L := [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
> L := [op(L), 10] ;
```

```
L := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Pour **ajouter un élément à la fin d'une liste** : On convertit la liste en une séquence par la commande `op`. Puis on concatène la séquence obtenue avec le nouvel élément grâce à la virgule `,`. Enfin on transforme cette séquence en une liste en l'entourant des crochets :

```
> L := [seq(i, i=1..9)] ;
```

```
L := [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
> L := [op(L), 10] ;
```

```
L := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Pour **ajouter un élément à la fin d'une liste** : On convertit la liste en une séquence par la commande `op`. Puis on concatène la séquence obtenue avec le nouvel élément grâce à la virgule ",". Enfin on transforme cette séquence en une liste en l'entourant des crochets :

```
> L := [seq(i, i=1..9)] ;
```

```
L := [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
> L := [op(L), 10] ;
```

```
L := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Pour **ajouter un élément à la fin d'une liste** : On convertit la liste en une séquence par la commande `op`. Puis on concatène la séquence obtenue avec le nouvel élément grâce à la virgule ",". Enfin on transforme cette séquence en une liste en l'entourant des crochets :

```
> L := [seq(i, i=1..9)] ;
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8, 9]$$

```
> L := [op(L), 10] ;
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Pour **ajouter un élément à la fin d'une liste** : On convertit la liste en une séquence par la commande `op`. Puis on concatène la séquence obtenue avec le nouvel élément grâce à la virgule ",". Enfin on transforme cette séquence en une liste en l'entourant des crochets :

```
> L := [seq(i, i=1..9)] ;
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8, 9]$$

```
> L := [op(L), 10] ;
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Pour **ajouter un élément à la fin d'une liste** : On convertit la liste en une séquence par la commande `op`. Puis on concatène la séquence obtenue avec le nouvel élément grâce à la virgule ",". Enfin on transforme cette séquence en une liste en l'entourant des crochets :

```
> L := [seq(i, i=1..9)] ;
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8, 9]$$

```
> L := [op(L), 10] ;
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Chap. III :  
Les objets  
structurés

Laurent  
Poinsot

Les objets  
structurés

Quelques  
exercices

Donner une méthode pour obtenir la longueur d'une  
séquence quelconque.

Étant donné une séquence  $S$ , on obtient une liste en écrivant “  $[S]$  ”. Il suffit alors d'utiliser `nops` : `nops([S])`, ce qui nous donne la longueur de la liste  $[S]$ , égale à la longueur de la séquence  $S$ .

Étant donné une séquence  $S$ , on obtient une liste en écrivant “  $[S]$  ”. Il suffit alors d'utiliser `nops` : `nops([S])`, ce qui nous donne la longueur de la liste  $[S]$ , égale à la longueur de la séquence  $S$ .

Étant donné une séquence  $S$ , on obtient une liste en écrivant “  $[S]$  ”. Il suffit alors d'utiliser `nops` : `nops([S])`, ce qui nous donne la longueur de la liste  $[S]$ , égale à la longueur de la séquence  $S$ .

# Plan

Chap. III :  
Les objets  
structurés

Laurent  
Poinsot

Les objets  
structurés

Quelques  
exercices

## 1 Les objets structurés

- Introduction
- Séquences
- Listes
- **Ensembles**
- Chaînes de caractères

## 2 Quelques exercices

Un **ensemble** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des accolades "{" et "}". Dans un ensemble on ne tient compte ni de l'ordre ni des répétitions !

```
> E := {a, b, c, d};
```

```
E := {a, b, c, d}
```

```
> whattype (E);
```

```
set
```

Un **ensemble** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des accolades "{" et "}". Dans un ensemble on ne tient compte ni de l'ordre ni des répétitions !

```
> E := {a, b, c, d};
```

```
E := {a, b, c, d}
```

```
> whattype (E);
```

```
set
```

Un **ensemble** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des accolades "{" et "}".

Dans un ensemble on ne tient compte ni de l'ordre ni des répétitions !

```
> E := {a, b, c, d};
```

```
E := {a, b, c, d}
```

```
> whattype (E);
```

```
set
```

Un **ensemble** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des accolades "{" et "}". Dans un ensemble on ne tient compte ni de l'ordre ni des répétitions !

```
> E := {a, b, c, d};
```

```
E := {a, b, c, d}
```

```
> whattype (E);
```

```
set
```

Un **ensemble** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des accolades "{" et "}". Dans un ensemble on ne tient compte ni de l'ordre ni des répétitions !

```
> E := {a, b, c, d} ;
```

```
E := {a, b, c, d}
```

```
> whattype (E) ;
```

```
set
```

Un **ensemble** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des accolades "{" et "}". Dans un ensemble on ne tient compte ni de l'ordre ni des répétitions !

```
> E := {a, b, c, d} ;
```

$$E := \{a, b, c, d\}$$

```
> whattype (E) ;
```

*set*

Un **ensemble** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des accolades "{" et "}". Dans un ensemble on ne tient compte ni de l'ordre ni des répétitions !

```
> E := {a, b, c, d} ;
```

$$E := \{a, b, c, d\}$$

```
> whattype (E) ;
```

*set*

Un **ensemble** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des accolades "{" et "}". Dans un ensemble on ne tient compte ni de l'ordre ni des répétitions !

```
> E := {a, b, c, d} ;
```

$$E := \{a, b, c, d\}$$

```
> whattype (E) ;
```

*set*

On transforme un ensemble en une séquence par la commande `op` :

```
> op (E) ;
```

*a, b, c, d*

Pour transformer un ensemble en une liste, on utilise `convert` :

```
> convert (E, list) ;
```

*[a, b, c, d]*

L'ensemble **vide** se note `{}` :

```
> {} ;
```

On transforme un ensemble en une séquence par la  
commande `op` :

```
> op (E) ;
```

*a, b, c, d*

Pour transformer un ensemble en une liste, on utilise  
`convert` :

```
> convert (E, list) ;
```

*[a, b, c, d]*

L'ensemble **vide** se note `{}` :

```
> {} ;
```

On transforme un ensemble en une séquence par la  
commande `op` :

```
> op (E) ;
```

*a, b, c, d*

Pour transformer un ensemble en une liste, on utilise  
`convert` :

```
> convert (E, list) ;
```

*[a, b, c, d]*

L'ensemble **vide** se note `{}` :

```
> {} ;
```

On transforme un ensemble en une séquence par la  
commande `op` :

```
> op (E) ;
```

*a, b, c, d*

Pour transformer un ensemble en une liste, on utilise  
`convert` :

```
> convert (E, list) ;
```

*[a, b, c, d]*

L'ensemble **vide** se note `{}` :

```
> {} ;
```

On transforme un ensemble en une séquence par la  
commande `op` :

```
> op (E) ;
```

*a, b, c, d*

Pour transformer un ensemble en une liste, on utilise  
`convert` :

```
> convert (E, list) ;
```

*[a, b, c, d]*

L'ensemble **vide** se note `{}` :

```
> {} ;
```

On transforme un ensemble en une séquence par la  
commande `op` :

```
> op (E) ;
```

*a, b, c, d*

Pour transformer un ensemble en une liste, on utilise  
`convert` :

```
> convert (E, list) ;
```

*[a, b, c, d]*

L'ensemble **vide** se note `{}` :

```
> {} ;
```

On transforme un ensemble en une séquence par la  
commande `op` :

```
> op (E) ;
```

*a, b, c, d*

Pour transformer un ensemble en une liste, on utilise  
`convert` :

```
> convert (E, list) ;
```

*[a, b, c, d]*

L'ensemble **vide** se note `{}` :

```
> {} ;
```

On transforme un ensemble en une séquence par la  
commande `op` :

```
> op (E) ;
```

*a, b, c, d*

Pour transformer un ensemble en une liste, on utilise  
`convert` :

```
> convert (E, list) ;
```

*[a, b, c, d]*

L'ensemble **vide** se note `{}` :

```
> {} ;
```

Donner une autre méthode pour convertir un ensemble en une liste (et une liste en un ensemble).

Soit  $E$  un ensemble. Avec  $op(E)$  on obtient une séquence, et donc  $[op(E)]$  construit une liste.

Soit  $L$  une liste. Avec  $op(L)$  on obtient une séquence, et donc  $\{op(L)\}$  construit un ensemble.

Soit  $E$  un ensemble. Avec  $op(E)$  on obtient une séquence, et donc  $[op(E)]$  construit une liste.

Soit  $L$  une liste. Avec  $op(L)$  on obtient une séquence, et donc  $\{op(L)\}$  construit un ensemble.

Soit  $E$  un ensemble. Avec  $op(E)$  on obtient une séquence, et donc  $[op(E)]$  construit une liste.

Soit  $L$  une liste. Avec  $op(L)$  on obtient une séquence, et donc  $\{op(L)\}$  construit un ensemble.

Soit  $E$  un ensemble. Avec  $op(E)$  on obtient une séquence, et donc  $[op(E)]$  construit une liste.

Soit  $L$  une liste. Avec  $op(L)$  on obtient une séquence, et donc  $\{op(L)\}$  construit un ensemble.

Soit  $E$  un ensemble. Avec  $op(E)$  on obtient une séquence, et donc  $[op(E)]$  construit une liste.

Soit  $L$  une liste. Avec  $op(L)$  on obtient une séquence, et donc  $\{op(L)\}$  construit un ensemble.

Soit  $E$  un ensemble. Avec  $op(E)$  on obtient une séquence, et donc  $[op(E)]$  construit une liste.

Soit  $L$  une liste. Avec  $op(L)$  on obtient une séquence, et donc  $\{op(L)\}$  construit un ensemble.

Expliquer comment à partir d'un ensemble on peut obtenir deux listes distinctes (soit les éléments sont dans un ordre différent, soit il y a plus d'éléments dans l'une que dans l'autre).

Rappelons qu'en Maple, ni l'ordre ni les répétitions des éléments dans un ensemble n'ont d'importance. Soit donc  $E_1 := \{a, b, c\}$  par exemple. Cet ensemble est égal à  $E_2 := \{a, c, b\}$ , mais aussi à  $E_3 := \{a, a, b, c, a, c, b\}$ . On convertit  $E_1$  en la liste  $[a, b, c]$ ,  $E_2$  en  $[a, c, b]$  et  $E_3$  en  $[a, a, b, c, a, c, b]$ . Les listes sont deux à deux distinctes (car dans une liste on tient évidemment compte de l'ordre et des répétitions !).

Rappelons qu'en Maple, ni l'ordre ni les répétitions des éléments dans un ensemble n'ont d'importance. Soit donc  $E_1 := \{a, b, c\}$  par exemple. Cet ensemble est égal à  $E_2 := \{a, c, b\}$ , mais aussi à  $E_3 := \{a, a, b, c, a, c, b\}$ . On convertit  $E_1$  en la liste  $[a, b, c]$ ,  $E_2$  en  $[a, c, b]$  et  $E_3$  en  $[a, a, b, c, a, c, b]$ . Les listes sont deux à deux distinctes (car dans une liste on tient évidemment compte de l'ordre et des répétitions !).

Rappelons qu'en Maple, ni l'ordre ni les répétitions des éléments dans un ensemble n'ont d'importance. Soit donc  $E_1 := \{a, b, c\}$  par exemple. Cet ensemble est égal à  $E_2 := \{a, c, b\}$ , mais aussi à  $E_3 := \{a, a, b, c, a, c, b\}$ . On convertit  $E_1$  en la liste  $[a, b, c]$ ,  $E_2$  en  $[a, c, b]$  et  $E_3$  en  $[a, a, b, c, a, c, b]$ . Les listes sont deux à deux distinctes (car dans une liste on tient évidemment compte de l'ordre et des répétitions !).

Rappelons qu'en Maple, ni l'ordre ni les répétitions des éléments dans un ensemble n'ont d'importance. Soit donc  $E_1 := \{a, b, c\}$  par exemple. Cet ensemble est égal à  $E_2 := \{a, c, b\}$ , mais aussi à  $E_3 := \{a, a, b, c, a, c, b\}$ . On convertit  $E_1$  en la liste  $[a, b, c]$ ,  $E_2$  en  $[a, c, b]$  et  $E_3$  en  $[a, a, b, c, a, c, b]$ . Les listes sont deux à deux distinctes (car dans une liste on tient évidemment compte de l'ordre et des répétitions !).

Rappelons qu'en Maple, ni l'ordre ni les répétitions des éléments dans un ensemble n'ont d'importance. Soit donc  $E_1 := \{a, b, c\}$  par exemple. Cet ensemble est égal à  $E_2 := \{a, c, b\}$ , mais aussi à  $E_3 := \{a, a, b, c, a, c, b\}$ . On convertit  $E_1$  en la liste  $[a, b, c]$ ,  $E_2$  en  $[a, c, b]$  et  $E_3$  en  $[a, a, b, c, a, c, b]$ . Les listes sont deux à deux distinctes (car dans une liste on tient évidemment compte de l'ordre et des répétitions !).

Rappelons qu'en Maple, ni l'ordre ni les répétitions des éléments dans un ensemble n'ont d'importance. Soit donc  $E_1 := \{a, b, c\}$  par exemple. Cet ensemble est égal à  $E_2 := \{a, c, b\}$ , mais aussi à  $E_3 := \{a, a, b, c, a, c, b\}$ . On convertit  $E_1$  en la liste  $[a, b, c]$ ,  $E_2$  en  $[a, c, b]$  et  $E_3$  en  $[a, a, b, c, a, c, b]$ . Les listes sont deux à deux distinctes (car dans une liste on tient évidemment compte de l'ordre et des répétitions !).

On peut effectuer différentes opérations sur les ensembles :  
**réunion** (union), **intersection** (intersect), **différence**  
**ensembliste** (minus) et le **test d'appartenance** d'un  
élément à un ensemble (member) :

```
> E := {a, b, c};
```

```
E := {a, b, c}
```

```
> F := {d, c, f, b, e};
```

```
F := {d, c, f, b, e}
```

```
> E union F;
```

```
{a, b, c, d, f, e}
```

```
> E intersect F;
```

```
{c, b}
```

```
> E minus F;
```

```
{a}
```

On peut effectuer différentes opérations sur les ensembles :  
**réunion** (union), **intersection** (intersect), **différence**  
**ensembliste** (minus) et le **test d'appartenance** d'un  
élément à un ensemble (member) :

```
> E := {a, b, c};
```

```
E := {a, b, c}
```

```
> F := {d, c, f, b, e};
```

```
F := {d, c, f, b, e}
```

```
> E union F;
```

```
{a, b, c, d, f, e}
```

```
> E intersect F;
```

```
{c, b}
```

```
> E minus F;
```

```
{a}
```

On peut effectuer différentes opérations sur les ensembles :  
**réunion** (union), **intersection** (intersect), **différence**  
**ensembliste** (minus) et le **test d'appartenance** d'un  
élément à un ensemble (member) :

```
> E := {a, b, c};
```

```
E := {a, b, c}
```

```
> F := {d, c, f, b, e};
```

```
F := {d, c, f, b, e}
```

```
> E union F;
```

```
{a, b, c, d, f, e}
```

```
> E intersect F;
```

```
{c, b}
```

```
> E minus F;
```

```
{a}
```

On peut effectuer différentes opérations sur les ensembles :  
**réunion** (union), **intersection** (intersect), **différence  
ensembliste** (minus) et le **test d'appartenance** d'un  
élément à un ensemble (member) :

```
> E := {a, b, c};
```

```
E := {a, b, c}
```

```
> F := {d, c, f, b, e};
```

```
F := {d, c, f, b, e}
```

```
> E union F;
```

```
{a, b, c, d, f, e}
```

```
> E intersect F;
```

```
{c, b}
```

```
> E minus F;
```

```
{a}
```

On peut effectuer différentes opérations sur les ensembles :  
**réunion** (`union`), **intersection** (`intersect`), **différence  
ensembliste** (`minus`) et le **test d'appartenance** d'un  
élément à un ensemble (`member`) :

```
> E := {a, b, c};
```

```
E := {a, b, c}
```

```
> F := {d, c, f, b, e};
```

```
F := {d, c, f, b, e}
```

```
> E union F;
```

```
{a, b, c, d, f, e}
```

```
> E intersect F;
```

```
{c, b}
```

```
> E minus F;
```

```
{a}
```

On peut effectuer différentes opérations sur les ensembles :  
**réunion** (`union`), **intersection** (`intersect`), **différence  
ensembliste** (`minus`) et le **test d'appartenance** d'un  
élément à un ensemble (`member`) :

```
> E := {a, b, c} ;
```

```
E := {a, b, c}
```

```
> F := {d, c, f, b, e} ;
```

```
F := {d, c, f, b, e}
```

```
> E union F ;
```

```
{a, b, c, d, f, e}
```

```
> E intersect F ;
```

```
{c, b}
```

```
> E minus F ;
```

```
{a}
```

On peut effectuer différentes opérations sur les ensembles :  
**réunion** (`union`), **intersection** (`intersect`), **différence  
ensembliste** (`minus`) et le **test d'appartenance** d'un  
élément à un ensemble (`member`) :

```
> E := {a, b, c} ;
```

```
E := {a, b, c}
```

```
> F := {d, c, f, b, e} ;
```

```
F := {d, c, f, b, e}
```

```
> E union F ;
```

```
{a, b, c, d, f, e}
```

```
> E intersect F ;
```

```
{c, b}
```

```
> E minus F ;
```

```
{a}
```

On peut effectuer différentes opérations sur les ensembles :  
**réunion** (`union`), **intersection** (`intersect`), **différence  
ensembliste** (`minus`) et le **test d'appartenance** d'un  
élément à un ensemble (`member`) :

```
> E := {a, b, c} ;
```

$E := \{a, b, c\}$

```
> F := {d, c, f, b, e} ;
```

$F := \{d, c, f, b, e\}$

```
> E union F ;
```

$\{a, b, c, d, f, e\}$

```
> E intersect F ;
```

$\{c, b\}$

```
> E minus F ;
```

$\{a\}$

On peut effectuer différentes opérations sur les ensembles :  
**réunion** (`union`), **intersection** (`intersect`), **différence  
ensembliste** (`minus`) et le **test d'appartenance** d'un  
élément à un ensemble (`member`) :

> E := {a, b, c} ;

$E := \{a, b, c\}$

> F := {d, c, f, b, e} ;

$F := \{d, c, f, b, e\}$

> E union F ;

$\{a, b, c, d, f, e\}$

> E intersect F ;

$\{c, b\}$

> E minus F ;

$\{a\}$

On peut effectuer différentes opérations sur les ensembles :  
**réunion** (`union`), **intersection** (`intersect`), **différence  
ensembliste** (`minus`) et le **test d'appartenance** d'un  
élément à un ensemble (`member`) :

> E := {a, b, c} ;

$E := \{a, b, c\}$

> F := {d, c, f, b, e} ;

$F := \{d, c, f, b, e\}$

> E union F ;

$\{a, b, c, d, f, e\}$

> E intersect F ;

$\{c, b\}$

> E minus F ;

$\{a\}$

On peut effectuer différentes opérations sur les ensembles :  
**réunion** (`union`), **intersection** (`intersect`), **différence  
ensembliste** (`minus`) et le **test d'appartenance** d'un  
élément à un ensemble (`member`) :

> E := {a, b, c} ;

$E := \{a, b, c\}$

> F := {d, c, f, b, e} ;

$F := \{d, c, f, b, e\}$

> E union F ;

$\{a, b, c, d, f, e\}$

> E intersect F ;

$\{c, b\}$

> E minus F ;

$\{a\}$

On peut effectuer différentes opérations sur les ensembles :  
**réunion** (`union`), **intersection** (`intersect`), **différence  
ensembliste** (`minus`) et le **test d'appartenance** d'un  
élément à un ensemble (`member`) :

> E := {a, b, c} ;

$E := \{a, b, c\}$

> F := {d, c, f, b, e} ;

$F := \{d, c, f, b, e\}$

> E union F ;

$\{a, b, c, d, f, e\}$

> E intersect F ;

$\{c, b\}$

> E minus F ;

$\{a\}$

On peut effectuer différentes opérations sur les ensembles :  
**réunion** (`union`), **intersection** (`intersect`), **différence  
ensembliste** (`minus`) et le **test d'appartenance** d'un  
élément à un ensemble (`member`) :

> E := {a, b, c} ;

$E := \{a, b, c\}$

> F := {d, c, f, b, e} ;

$F := \{d, c, f, b, e\}$

> E union F ;

$\{a, b, c, d, f, e\}$

> E intersect F ;

$\{c, b\}$

> E minus F ;

$\{a\}$

On peut effectuer différentes opérations sur les ensembles :  
**réunion** (`union`), **intersection** (`intersect`), **différence  
ensembliste** (`minus`) et le **test d'appartenance** d'un  
élément à un ensemble (`member`) :

> E := {a, b, c} ;

$E := \{a, b, c\}$

> F := {d, c, f, b, e} ;

$F := \{d, c, f, b, e\}$

> E union F ;

$\{a, b, c, d, f, e\}$

> E intersect F ;

$\{c, b\}$

> E minus F ;

$\{a\}$

On peut effectuer différentes opérations sur les ensembles :  
**réunion** (`union`), **intersection** (`intersect`), **différence  
ensembliste** (`minus`) et le **test d'appartenance** d'un  
élément à un ensemble (`member`) :

> E := {a, b, c} ;

$E := \{a, b, c\}$

> F := {d, c, f, b, e} ;

$F := \{d, c, f, b, e\}$

> E union F ;

$\{a, b, c, d, f, e\}$

> E intersect F ;

$\{c, b\}$

> E minus F ;

$\{a\}$

On a aussi :

```
> member (a, E) ;
```

*true*

```
> member (a, F) ;
```

*false*

On a aussi :

```
> member (a, E) ;
```

*true*

```
> member (a, F) ;
```

*false*

On a aussi :

```
> member (a, E) ;
```

*true*

```
> member (a, F) ;
```

*false*

On a aussi :

```
> member (a, E) ;
```

*true*

```
> member (a, F) ;
```

*false*

On a aussi :

```
> member (a, E) ;
```

*true*

```
> member (a, F) ;
```

*false*

# Plan

Chap. III :  
Les objets  
structurés

Laurent  
Poinot

Les objets  
structurés

Quelques  
exercices

- 1 Les objets structurés
  - Introduction
  - Séquences
  - Listes
  - Ensembles
  - Chaînes de caractères

- 2 Quelques exercices

Une **chaîne de caractères** est une suite de caractères encadrée par des guillemets ("").

```
> mot := "bonjour" ;
```

```
mot := " bonjour"
```

```
> whattype (mot) ;
```

```
string
```

Pour extraire une **sous-chaîne** on utilise les crochets "[" et "]" :

```
> mot [2..5] ;
```

```
" onjo"
```

Une **chaîne de caractères** est une suite de caractères encadrée par des guillemets ("").

```
> mot := "bonjour" ;
```

```
mot := " bonjour"
```

```
> whattype (mot) ;
```

```
string
```

Pour extraire une **sous-chaîne** on utilise les crochets "[" et "]" :

```
> mot [2..5] ;
```

```
" onjo"
```

Une **chaîne de caractères** est une suite de caractères encadrée par des guillemets ("").

```
> mot := "bonjour" ;
```

*mot := " bonjour"*

```
> whattype (mot) ;
```

*string*

Pour extraire une **sous-chaîne** on utilise les crochets "[" et "]" :

```
> mot [2..5] ;
```

*" onjo"*

Une **chaîne de caractères** est une suite de caractères encadrée par des guillemets ("").

```
> mot := "bonjour" ;
```

```
mot := " bonjour"
```

```
> whattype (mot) ;
```

```
string
```

Pour extraire une **sous-chaîne** on utilise les crochets "[" et "]" :

```
> mot [2..5] ;
```

```
" onjo"
```

Une **chaîne de caractères** est une suite de caractères encadrée par des guillemets ("").

```
> mot := "bonjour" ;
```

*mot := " bonjour"*

```
> whattype (mot) ;
```

*string*

Pour extraire une **sous-chaîne** on utilise les crochets "[" et "]" :

```
> mot [2..5] ;
```

*" onjo"*

Une **chaîne de caractères** est une suite de caractères encadrée par des guillemets ("").

```
> mot := "bonjour" ;
```

```
mot := " bonjour"
```

```
> whattype (mot) ;
```

```
string
```

Pour extraire une **sous-chaîne** on utilise les crochets "[" et "]" :

```
> mot [2..5] ;
```

```
" onjo"
```

Une **chaîne de caractères** est une suite de caractères encadrée par des guillemets ("").

```
> mot := "bonjour" ;
```

*mot := " bonjour"*

```
> whattype (mot) ;
```

*string*

Pour extraire une **sous-chaîne** on utilise les crochets "[" et "]" :

```
> mot [2..5] ;
```

*" onjo"*

Une **chaîne de caractères** est une suite de caractères encadrée par des guillemets ("").

```
> mot := "bonjour" ;
```

*mot := " bonjour"*

```
> whattype (mot) ;
```

*string*

Pour extraire une **sous-chaîne** on utilise les crochets "[" et "]" :

```
> mot [2..5] ;
```

*" onjo"*

On peut **concaténer** deux chaînes de caractères par la commande `cat` :

```
> mot2 :=cat (mot, " chez vous") ;
```

```
mot2 := "bonjour chez vous"
```

La **longueur** d'une chaîne est donnée par la commande `length` :

```
> length (mot2) ;
```

17

On peut **concaténer** deux chaînes de caractères par la commande `cat` :

```
> mot2 :=cat (mot, " chez vous") ;
```

```
mot2 := "bonjour chez vous"
```

La **longueur** d'une chaîne est donnée par la commande `length` :

```
> length (mot2) ;
```

17

On peut **concaténer** deux chaînes de caractères par la commande `cat` :

```
> mot2 :=cat (mot, " chez vous") ;
```

```
mot2 := "bonjour chez vous"
```

La **longueur** d'une chaîne est donnée par la commande `length` :

```
> length (mot2) ;
```

17

On peut **concaténer** deux chaînes de caractères par la commande `cat` :

```
> mot2 :=cat (mot, " chez vous") ;
```

*mot2 := "bonjour chez vous"*

La **longueur** d'une chaîne est donnée par la commande `length` :

```
> length (mot2) ;
```

17

On peut **concaténer** deux chaînes de caractères par la commande `cat` :

```
> mot2 :=cat (mot, " chez vous") ;
```

*mot2 := "bonjour chez vous"*

La **longueur** d'une chaîne est donnée par la commande `length` :

```
> length (mot2) ;
```

On peut **concaténer** deux chaînes de caractères par la commande `cat` :

```
> mot2 :=cat (mot, " chez vous") ;
```

*mot2 := "bonjour chez vous"*

La **longueur** d'une chaîne est donnée par la commande `length` :

```
> length (mot2) ;
```

17

# Exercice 1

Chap. III :  
Les objets  
structurés

Laurent  
Poinsot

Les objets  
structurés

Quelques  
exercices

À l'aide notamment des fonctions `seq`, `select` et `remove`, construire la séquence des nombres premiers strictement inférieurs à 19 de deux façons différentes (l'une utilisant `select` et l'autre `remove`).

# Correction exercice 1

Chap. III :  
Les objets  
structurés

Laurent  
Poinsot

Les objets  
structurés

Quelques  
exercices

On commence par créer la séquence des nombres entre 1 et 19 :  $S := \text{seq}(i, i=1..19)$  ;

Une première manière de construire la séquence des premiers entre 1 et 19 est

```
 $S := \text{op}(\text{select}(\text{isprime}, [S]))$  ;
```

$$S := [2, 3, 5, 7, 11, 13, 17, 19]$$

Notez que l'on a d'abord transformé la séquence  $S$  en une liste par  $[S]$  (car `select`, comme `remove` ne s'applique qu'aux listes), puis on a sélectionné les nombres premiers, et enfin on a transformé le résultat (qui est une liste) en une séquence par `op`.

# Correction exercice 1

Chap. III :  
Les objets  
structurés

Laurent  
Poinot

Les objets  
structurés

Quelques  
exercices

On commence par créer la séquence des nombres entre 1 et 19 :  $S := \text{seq}(i, i=1..19)$  ;

Une première manière de construire la séquence des premiers entre 1 et 19 est

```
S := op(select(isprime, [S])) ;
```

$$S := [2, 3, 5, 7, 11, 13, 17, 19]$$

Notez que l'on a d'abord transformé la séquence  $S$  en une liste par  $[S]$  (car `select`, comme `remove` ne s'applique qu'aux listes), puis on a sélectionné les nombres premiers, et enfin on a transformé le résultat (qui est une liste) en une séquence par `op`.

# Correction exercice 1

Chap. III :  
Les objets  
structurés

Laurent  
Poinot

Les objets  
structurés

Quelques  
exercices

On commence par créer la séquence des nombres entre 1 et 19 :  $S := \text{seq}(i, i=1..19)$  ;

Une première manière de construire la séquence des premiers entre 1 et 19 est

```
S := op(select(isprime, [S])) ;
```

$$S := [2, 3, 5, 7, 11, 13, 17, 19]$$

Notez que l'on a d'abord transformé la séquence  $S$  en une liste par  $[S]$  (car `select`, comme `remove` ne s'applique qu'aux listes), puis on a sélectionné les nombres premiers, et enfin on a transformé le résultat (qui est une liste) en une séquence par `op`.

# Correction exercice 1

Chap. III :  
Les objets  
structurés

Laurent  
Poinsot

Les objets  
structurés

Quelques  
exercices

On commence par créer la séquence des nombres entre 1 et 19 :  $S := \text{seq}(i, i=1..19)$  ;

Une première manière de construire la séquence des premiers entre 1 et 19 est

$S := \text{op}(\text{select}(\text{isprime}, [S]))$  ;

$S := [2, 3, 5, 7, 11, 13, 17, 19]$

Notez que l'on a d'abord transformé la séquence  $S$  en une liste par  $[S]$  (car `select`, comme `remove` ne s'applique qu'aux listes), puis on a sélectionné les nombres premiers, et enfin on a transformé le résultat (qui est une liste) en une séquence par `op`.

# Correction exercice 1

Chap. III :  
Les objets  
structurés

Laurent  
Poinsot

Les objets  
structurés

Quelques  
exercices

On commence par créer la séquence des nombres entre 1 et 19 :  $S := \text{seq}(i, i=1..19)$  ;

Une première manière de construire la séquence des premiers entre 1 et 19 est

$S := \text{op}(\text{select}(\text{isprime}, [S]))$  ;

$S := [2, 3, 5, 7, 11, 13, 17, 19]$

Notez que l'on a d'abord transformé la séquence  $S$  en une liste par  $[S]$  (car `select`, comme `remove` ne s'applique qu'aux listes), puis on a sélectionné les nombres premiers, et enfin on a transformé le résultat (qui est une liste) en une séquence par `op`.

# Correction exercice 1

Chap. III :  
Les objets  
structurés

Laurent  
Poinsot

Les objets  
structurés

Quelques  
exercices

On commence par créer la séquence des nombres entre 1 et 19 :  $S := \text{seq}(i, i=1..19)$  ;

Une première manière de construire la séquence des premiers entre 1 et 19 est

$S := \text{op}(\text{select}(\text{isprime}, [S]))$  ;

$S := [2, 3, 5, 7, 11, 13, 17, 19]$

Notez que l'on a d'abord transformé la séquence  $S$  en une liste par  $[S]$  (car `select`, comme `remove` ne s'applique qu'aux listes), puis on a sélectionné les nombres premiers, et enfin on a transformé le résultat (qui est une liste) en une séquence par `op`.

# Correction exercice 1

Chap. III :  
Les objets  
structurés

Laurent  
Poinot

Les objets  
structurés

Quelques  
exercices

On commence par créer la séquence des nombres entre 1 et 19 :  $S := \text{seq}(i, i=1..19)$  ;

Une première manière de construire la séquence des premiers entre 1 et 19 est

```
S := op(select(isprime, [S])) ;
```

$$S := [2, 3, 5, 7, 11, 13, 17, 19]$$

Notez que l'on a d'abord transformé la séquence  $S$  en une liste par  $[S]$  (car `select`, comme `remove` ne s'applique qu'aux listes), puis on a sélectionné les nombres premiers, et enfin on a transformé le résultat (qui est une liste) en une séquence par `op`.

# Correction exercice 1

Chap. III :  
Les objets  
structurés

Laurent  
Poinot

Les objets  
structurés

Quelques  
exercices

On commence par créer la séquence des nombres entre 1 et 19 :  $S := \text{seq}(i, i=1..19)$  ;

Une première manière de construire la séquence des premiers entre 1 et 19 est

```
S := op(select(isprime, [S])) ;
```

$$S := [2, 3, 5, 7, 11, 13, 17, 19]$$

Notez que l'on a d'abord transformé la séquence  $S$  en une liste par  $[S]$  (car `select`, comme `remove` ne s'applique qu'aux listes), puis on a sélectionné les nombres premiers, et enfin on a transformé le résultat (qui est une liste) en une séquence par `op`.

On recommence avec `S :=seq(i, i=1..19) ;` On définit  
la fonction booléenne `notprime` par

```
notprime := n -> evalb(not(isprime(n))) ;
```

Puis on retire les éléments non premiers de la séquence `S` :

```
S :=op(remove(notprime, [S])) ;
```

On recommence avec `S := seq(i, i=1..19) ;` On définit  
la fonction booléenne `notprime` par

```
notprime := n -> evalb(not(isprime(n))) ;
```

Puis on retire les éléments non premiers de la séquence `S` :

```
S := op(remove(notprime, [S])) ;
```

On recommence avec `S :=seq(i, i=1..19)` ; On définit  
la fonction booléenne `notprime` par

```
notprime := n -> evalb(not(isprime(n)));
```

Puis on retire les éléments non premiers de la séquence `S` :

```
S :=op(remove(notprime, [S]));
```

On recommence avec `S :=seq(i, i=1..19)` ; On définit  
la fonction booléenne `notprime` par  
`notprime := n -> evalb(not(isprime(n)))` ;  
Puis on retire les éléments non premiers de la séquence `S` :

```
S :=op(remove(notprime, [S])) ;
```

On recommence avec `S := seq(i, i=1..19)` ; On définit  
la fonction booléenne `notprime` par  
`notprime := n -> evalb(not(isprime(n)))` ;  
Puis on retire les éléments non premiers de la séquence `S` :

```
S := op(remove(notprime, [S])) ;
```

# Exercice 2

Chap. III :  
Les objets  
structurés

Laurent  
Poinsot

Les objets  
structurés

Quelques  
exercices

- 1 Écrire une fonction booléenne `pair` qui renvoie `true` lorsque son argument est pair, et `false` dans le cas contraire ;
- 2 Construire l'ensemble de tous les nombres pairs entre 0 et 100 (à l'aide des fonctions `seq` et `select`).

# Correction exercice 2

Chap. III :  
Les objets  
structurés

Laurent  
Poinot

Les objets  
structurés

Quelques  
exercices

```
1 pair := n->evalb(irem(n,2)=0) ;  
2 S :=seq(i,i=0..100) ;  
  S :={op(select(pair,[S]))} ;
```

# Correction exercice 2

Chap. III :  
Les objets  
structurés

Laurent  
Poinot

Les objets  
structurés

Quelques  
exercices

```
1 pair := n->evalb(irem(n,2)=0) ;  
2 S :=seq(i,i=0..100) ;  
   S :={op(select(pair,[S]))} ;
```

# Exercice 3

Chap. III :  
Les objets  
structurés

Laurent  
Poinot

Les objets  
structurés

Quelques  
exercices

- 1 Écrire une fonction booléenne `IsDiv` qui prend deux arguments (des entiers naturels)  $n$ ,  $d$  et qui renvoie `true` si, et seulement si,  $n$  est divisible par  $d$ . (On suppose que  $d > 0$ .)
- 2 Constuire la liste des entiers entre 0 et 1000 qui sont divisibles ni par 3 ni par 5.

# Correction exercice 3

Chap. III :  
Les objets  
structurés

Laurent  
Poinot

Les objets  
structurés

Quelques  
exercices

```
1 IsDiv := (n,d)->evalb(irem(n,d)=0) ;  
2 L :=[seq(i,i=0..1000)] ; IsDiv3_5 :=  
  n->(IsDiv(n,3) and IsDiv(n,5)) ;  
  L :=remove(IsDiv3_5,L) ;
```

# Correction exercice 3

Chap. III :  
Les objets  
structurés

Laurent  
Poinot

Les objets  
structurés

Quelques  
exercices

```
1 IsDiv := (n,d)->evalb(irem(n,d)=0) ;  
2 L :=[seq(i,i=0..1000)] ; IsDiv3_5 :=  
n->(IsDiv(n,3) and IsDiv(n,5)) ;  
L :=remove(IsDiv3_5,L) ;
```

# Exercice 4

Chap. III :  
Les objets  
structurés

Laurent  
Poinot

Les objets  
structurés

Quelques  
exercices

On suppose que l'on a deux listes  $A$  et  $B$  de même longueur (bien que celle-ci nous soit inconnue) constituée d'entier (on suppose également que les listes sont non vides).  
Constuire la liste  $C$  dont la  $i$ ème case contient la somme du  $i$ ème élément de  $A$  avec le  $i$ ème élément de  $B$ .

# Correction exercice 4

Chap. III :  
Les objets  
structurés

Laurent  
Poinsot

Les objets  
structurés

Quelques  
exercices

```
C := [seq(A[i]+B[i], i=1..nops(A))];
```