

Chap. III : Programmation

Laurent Poinsot

19 janvier 2009

Sous Maple, comme dans la grande majorité des langages de programmation, il existe trois instructions fondamentales pour la programmation, à savoir, `if`, `for` et `while`. Par ailleurs il existe bien évidemment la possibilité d'écrire des procédures qui permettent de "factoriser" des parties de programmes.

Dans la syntaxe des instructions de programmation, `<objet>` signifie que l'objet entre `<` et `>` est un nom général que l'on devra remplacer afin d'écrire l'instruction, et `[instruction]` signifie que l'instruction entre crochets est facultative. Par ex. `<var>` désigne un nom de variable quelconque que l'utilisateur prendra soin de remplacer par un vrai nom de variable valide sous Maple.

Sous Maple, comme dans la grande majorité des langages de programmation, il existe trois instructions fondamentales pour la programmation, à savoir, `if`, `for` et `while`. Par ailleurs il existe bien évidemment la possibilité d'écrire des procédures qui permettent de "factoriser" des parties de programmes.

Dans la syntaxe des instructions de programmation, `<objet>` signifie que l'objet entre `<` et `>` est un nom général que l'on devra remplacer afin d'écrire l'instruction, et `[instruction]` signifie que l'instruction entre crochets est facultative. Par ex. `<var>` désigne un nom de variable quelconque que l'utilisateur prendra soin de remplacer par un vrai nom de variable valide sous Maple.

Sous Maple, comme dans la grande majorité des langages de programmation, il existe trois instructions fondamentales pour la programmation, à savoir, `if`, `for` et `while`. Par ailleurs il existe bien évidemment la possibilité d'écrire des procédures qui permettent de "factoriser" des parties de programmes.

Dans la syntaxe des instructions de programmation, `<objet>` signifie que l'objet entre `<` et `>` est un nom général que l'on devra remplacer afin d'écrire l'instruction, et `[instruction]` signifie que l'instruction entre crochets est facultative. Par ex. `<var>` désigne un nom de variable quelconque que l'utilisateur prendra soin de remplacer par un vrai nom de variable valide sous Maple.

Sous Maple, comme dans la grande majorité des langages de programmation, il existe trois instructions fondamentales pour la programmation, à savoir, `if`, `for` et `while`. Par ailleurs il existe bien évidemment la possibilité d'écrire des procédures qui permettent de "factoriser" des parties de programmes.

Dans la syntaxe des instructions de programmation, `<objet>` signifie que l'objet entre `<` et `>` est un nom général que l'on devra remplacer afin d'écrire l'instruction, et `[instruction]` signifie que l'instruction entre crochets est facultative. Par ex. `<var>` désigne un nom de variable quelconque que l'utilisateur prendra soin de remplacer par un vrai nom de variable valide sous Maple.

Sous Maple, comme dans la grande majorité des langages de programmation, il existe trois instructions fondamentales pour la programmation, à savoir, `if`, `for` et `while`. Par ailleurs il existe bien évidemment la possibilité d'écrire des procédures qui permettent de "factoriser" des parties de programmes.

Dans la syntaxe des instructions de programmation, `<objet>` signifie que l'objet entre `<` et `>` est un nom général que l'on devra remplacer afin d'écrire l'instruction, et `[instruction]` signifie que l'instruction entre crochets est facultative. Par ex. `<var>` désigne un nom de variable quelconque que l'utilisateur prendra soin de remplacer par un vrai nom de variable valide sous Maple.

Sous Maple, comme dans la grande majorité des langages de programmation, il existe trois instructions fondamentales pour la programmation, à savoir, `if`, `for` et `while`. Par ailleurs il existe bien évidemment la possibilité d'écrire des procédures qui permettent de "factoriser" des parties de programmes.

Dans la syntaxe des instructions de programmation, `<objet>` signifie que l'objet entre `<` et `>` est un nom général que l'on devra remplacer afin d'écrire l'instruction, et `[instruction]` signifie que l'instruction entre crochets est facultative. Par ex. `<var>` désigne un nom de variable quelconque que l'utilisateur prendra soin de remplacer par un vrai nom de variable valide sous Maple.

Sous Maple, comme dans la grande majorité des langages de programmation, il existe trois instructions fondamentales pour la programmation, à savoir, `if`, `for` et `while`. Par ailleurs il existe bien évidemment la possibilité d'écrire des procédures qui permettent de "factoriser" des parties de programmes.

Dans la syntaxe des instructions de programmation, `<objet>` signifie que l'objet entre `<` et `>` est un nom général que l'on devra remplacer afin d'écrire l'instruction, et `[instruction]` signifie que l'instruction entre crochets est facultative. Par ex. `<var>` désigne un nom de variable quelconque que l'utilisateur prendra soin de remplacer par un vrai nom de variable valide sous Maple.

Sous Maple, comme dans la grande majorité des langages de programmation, il existe trois instructions fondamentales pour la programmation, à savoir, `if`, `for` et `while`. Par ailleurs il existe bien évidemment la possibilité d'écrire des procédures qui permettent de "factoriser" des parties de programmes.

Dans la syntaxe des instructions de programmation, `<objet>` signifie que l'objet entre `<` et `>` est un nom général que l'on devra remplacer afin d'écrire l'instruction, et `[instruction]` signifie que l'instruction entre crochets est facultative. Par ex. `<var>` désigne un nom de variable quelconque que l'utilisateur prendra soin de remplacer par un vrai nom de variable valide sous Maple.

Sous `Maple`, comme dans la grande majorité des langages de programmation, il existe trois instructions fondamentales pour la programmation, à savoir, `if`, `for` et `while`. Par ailleurs il existe bien évidemment la possibilité d'écrire des procédures qui permettent de "factoriser" des parties de programmes.

Dans la syntaxe des instructions de programmation, `<objet>` signifie que l'objet entre `<` et `>` est un nom général que l'on devra remplacer afin d'écrire l'instruction, et `[instruction]` signifie que l'instruction entre crochets est facultative. Par ex. `<var>` désigne un nom de variable quelconque que l'utilisateur prendra soin de remplacer par un vrai nom de variable valide sous `Maple`.

Procédures

Chap. III :
Programmation

Laurent
Poinsot

La syntaxe d'une procédure Maple est la suivante :

```
<Nom de la procédure> := proc  
  ([<argument1>, <argument2>, ...])  
  [local <var1>, <var2>, ...;]  
  <instruction1>;  
  <instruction2>;  
  ...  
  [return <résultat>;]  
end proc;
```

Procédures

Chap. III :
Programmation

Laurent
Poinsot

La syntaxe d'une procédure Maple est la suivante :

```
<Nom de la procédure> := proc  
  ([<argument1>, <argument2>, ...])  
  [local <var1>, <var2>, ...;]  
  <instruction1>;  
  <instruction2>;  
  ...  
  [return <résultat>;]  
end proc;
```

Procédures

Chap. III :
Programmation

Laurent
Poinsot

La syntaxe d'une procédure Maple est la suivante :

```
<Nom de la procédure> := proc  
  ([<argument1>, <argument2>, ...])  
  [local <var1>, <var2>, ...;]  
  <instruction1>;  
  <instruction2>;  
  ...  
  [return <résultat>;]  
end proc;
```

Procédures

Chap. III :
Programmation

Laurent
Poinsot

La syntaxe d'une procédure Maple est la suivante :

```
<Nom de la procédure> := proc  
  ([<argument1>, <argument2>, ...])  
  [local <var1>, <var2>, ...;]  
  <instruction1>;  
  <instruction2>;  
  ...  
  [return <résultat>;]  
end proc;
```

Procédures

Chap. III :
Programmation

Laurent
Poinsot

La syntaxe d'une procédure Maple est la suivante :

```
<Nom de la procédure> := proc  
  ([<argument1>, <argument2>, ...])  
  [local <var1>, <var2>, ...;]  
  <instruction1>;  
  <instruction2>;  
  ...  
  [return <résultat>;]  
end proc;
```

Procédures

Chap. III :
Programmation

Laurent
Poinso

La syntaxe d'une procédure Maple est la suivante :

```
<Nom de la procédure> := proc  
  ([<argument1>, <argument2>, ...])  
  [local <var1>, <var2>, ...;]  
  <instruction1>;  
  <instruction2>;  
  ...  
  [return <résultat>;]  
end proc;
```


Procédures

Chap. III :
Programmation

Laurent
Poinso

La syntaxe d'une procédure Maple est la suivante :

```
<Nom de la procédure> := proc  
  ([<argument1>, <argument2>, ...])  
  [local <var1>, <var2>, ...;]  
  <instruction1>;  
  <instruction2>;  
  ...  
  [return <résultat>;]  
end proc;
```

Procédures

Chap. III :
Programmation

Laurent
Poincot

La syntaxe d'une procédure Maple est la suivante :

```
<Nom de la procédure> := proc  
  ([<argument1>, <argument2>, ...])  
  [local <var1>, <var2>, ...;]  
  <instruction1>;  
  <instruction2>;  
  ...  
  [return <résultat>;]  
end proc;
```

- `<Nom de la procédure>` est le nom que l'on donne à la procédure ;
- La procédure utilisera les valeurs données par l'utilisateur pour les arguments **formels** `<argument1>, <argument2>, ...` s'ils sont présents (une procédure sans argument formel se déclare donc par `<Nom de la procédure> := proc ()...` ne pas oublier les parenthèses !);
- Les variables **locales** `<var1>, <var2>, ...`, elles aussi facultatives, doivent être déclarées après le mot clef `local`. Elles ne sont accessibles et manipulables que dans le bloc d'instructions de la procédure à partir de l'endroit où elles sont déclarées et jusqu'à l'instruction finale `"end proc ;"`;

- `<Nom de la procédure>` est le nom que l'on donne à la procédure ;
- La procédure utilisera les valeurs données par l'utilisateur pour les arguments **formels** `<argument1>, <argument2>, ...` s'ils sont présents (une procédure sans argument formel se déclare donc par `<Nom de la procédure> := proc ()...` ne pas oublier les parenthèses !);
- Les variables **locales** `<var1>, <var2>, ...`, elles aussi facultatives, doivent être déclarées après le mot clef `local`. Elles ne sont accessibles et manipulables que dans le bloc d'instructions de la procédure à partir de l'endroit où elles sont déclarées et jusqu'à l'instruction finale `"end proc ;"`;

- `<Nom de la procédure>` est le nom que l'on donne à la procédure ;
- La procédure utilisera les valeurs données par l'utilisateur pour les arguments **formels** `<argument1>`, `<argument2>`, ... s'ils sont présents (une procédure sans argument formel se déclare donc par `<Nom de la procédure> := proc ()...` ne pas oublier les parenthèses !);
- Les variables **locales** `<var1>`, `<var2>`, ..., elles aussi facultatives, doivent être déclarées après le mot clef `local`. Elles ne sont accessibles et manipulables que dans le bloc d'instructions de la procédure à partir de l'endroit où elles sont déclarées et jusqu'à l'instruction finale `"end proc ;"`;

- `<Nom de la procédure>` est le nom que l'on donne à la procédure ;
- La procédure utilisera les valeurs données par l'utilisateur pour les arguments **formels** `<argument1>, <argument2>, ...` s'ils sont présents (une procédure sans argument formel se déclare donc **par** `<Nom de la procédure> := proc () ...` ne pas oublier les parenthèses !);
- Les variables **locales** `<var1>, <var2>, ...`, elles aussi facultatives, doivent être déclarées après le mot clef `local`. Elles ne sont accessibles et manipulables que dans le bloc d'instructions de la procédure à partir de l'endroit où elles sont déclarées et jusqu'à l'instruction finale `"end proc ;"`;

- `<Nom de la procédure>` est le nom que l'on donne à la procédure ;
- La procédure utilisera les valeurs données par l'utilisateur pour les arguments **formels** `<argument1>, <argument2>, ...` s'ils sont présents (une procédure sans argument formel se déclare donc par `<Nom de la procédure> := proc ()...` ne pas oublier les parenthèses !);
- Les variables **locales** `<var1>, <var2>, ...`, elles aussi facultatives, doivent être déclarées après le mot clef `local`. Elles ne sont accessibles et manipulables que dans le bloc d'instructions de la procédure à partir de l'endroit où elles sont déclarées et jusqu'à l'instruction finale `"end proc ;"`;

- `<Nom de la procédure>` est le nom que l'on donne à la procédure ;
- La procédure utilisera les valeurs données par l'utilisateur pour les arguments **formels** `<argument1>, <argument2>, ...` s'ils sont présents (une procédure sans argument formel se déclare donc par `<Nom de la procédure> := proc ()...` ne pas oublier les parenthèses !);
- Les variables **locales** `<var1>, <var2>, ...`, elles aussi facultatives, doivent être déclarées après le mot clef `local`. Elles ne sont accessibles et manipulables que dans le bloc d'instructions de la procédure à partir de l'endroit où elles sont déclarées et jusqu'à l'instruction finale `"end proc ;"`;

- `<Nom de la procédure>` est le nom que l'on donne à la procédure ;
- La procédure utilisera les valeurs données par l'utilisateur pour les arguments **formels** `<argument1>, <argument2>, ...` s'ils sont présents (une procédure sans argument formel se déclare donc par `<Nom de la procédure> := proc ()... ne pas oublier les parenthèses !`);
- Les variables **locales** `<var1>, <var2>, ...`, elles aussi facultatives, doivent être déclarées après le mot clef `local`. Elles ne sont accessibles et manipulables que dans le bloc d'instructions de la procédure à partir de l'endroit où elles sont déclarées et jusqu'à l'instruction finale `"end proc ;"`;

- `<Nom de la procédure>` est le nom que l'on donne à la procédure ;
- La procédure utilisera les valeurs données par l'utilisateur pour les arguments **formels** `<argument1>, <argument2>, ...` s'ils sont présents (une procédure sans argument formel se déclare donc par `<Nom de la procédure> := proc ()...` ne pas oublier les parenthèses !);
- Les variables **locales** `<var1>, <var2>, ...`, elles aussi facultatives, doivent être déclarées après le mot clef `local`. Elles ne sont accessibles et manipulables que dans le bloc d'instructions de la procédure à partir de l'endroit où elles sont déclarées et jusqu'à l'instruction finale `"end proc ;"`;

- `<Nom de la procédure>` est le nom que l'on donne à la procédure ;
- La procédure utilisera les valeurs données par l'utilisateur pour les arguments **formels** `<argument1>`, `<argument2>`, ... s'ils sont présents (une procédure sans argument formel se déclare donc par `<Nom de la procédure> := proc ()...` ne pas oublier les parenthèses !);
- Les variables **locales** `<var1>`, `<var2>`, ..., elles aussi facultatives, doivent être déclarées après le mot clef `local`. Elles ne sont accessibles et manipulables que dans le bloc d'instructions de la procédure à partir de l'endroit où elles sont déclarées et jusqu'à l'instruction finale `"end proc ;"`;

- L'instruction "return" permet de renvoyer le résultat d'un calcul effectué dans la procédure et qui est contenu dans la variable <résultat>;
- La procédure se termine par l'instruction "end proc;".

- L'instruction "return" permet de renvoyer le résultat d'un calcul effectué dans la procédure et qui est contenu dans la variable <résultat>;
- La procédure se termine par l'instruction "end proc;".

- L'instruction "return" permet de renvoyer le résultat d'un calcul effectué dans la procédure et qui est contenu dans la variable <résultat>;
- La procédure se termine par l'instruction "end proc;".

- L'instruction `return` permet de renvoyer le résultat d'un calcul effectué dans la procédure et qui est contenu dans la variable `<résultat>`;
- La procédure se termine par l'instruction `end proc ;`.

Un premier exemple : la procédure suivante effectue l'addition des nombres a et b passés en argument :

```
>addition := proc(a,b)
local res;
res :=a+b;
return res;
end proc;
```

Remarquons que l'on pourrait aussi écrire cette même procédure de la façon suivante :

```
>addition := proc(a,b)
return (a+b);
end proc;
```


Un premier exemple : la procédure suivante effectue l'addition des nombres a et b passés en argument :

```
>addition := proc(a,b)
local res;
res :=a+b;
return res;
end proc;
```

Remarquons que l'on pourrait aussi écrire cette même procédure de la façon suivante :

```
>addition := proc(a,b)
return (a+b);
end proc;
```

Un premier exemple : la procédure suivante effectue l'addition des nombres a et b passés en argument :

```
>addition := proc(a,b)
local res;
res :=a+b;
return res;
end proc;
```

Remarquons que l'on pourrait aussi écrire cette même procédure de la façon suivante :

```
>addition := proc(a,b)
return (a+b);
end proc;
```

Un premier exemple : la procédure suivante effectue l'addition des nombres a et b passés en argument :

```
>addition := proc(a,b)
local res;
res :=a+b;
return res;
end proc;
```

Remarquons que l'on pourrait aussi écrire cette même procédure de la façon suivante :

```
>addition := proc(a,b)
return (a+b);
end proc;
```

Un premier exemple : la procédure suivante effectue l'addition des nombres a et b passés en argument :

```
>addition := proc(a,b)
local res;
res :=a+b;
return res;
end proc;
```

Remarquons que l'on pourrait aussi écrire cette même procédure de la façon suivante :

```
>addition := proc(a,b)
return (a+b);
end proc;
```

Un premier exemple : la procédure suivante effectue l'addition des nombres a et b passés en argument :

```
>addition := proc(a,b)
local res;
res :=a+b;
return res;
end proc;
```

Remarquons que l'on pourrait aussi écrire cette même procédure de la façon suivante :

```
>addition := proc(a,b)
return (a+b);
end proc;
```

Un premier exemple : la procédure suivante effectue l'addition des nombres a et b passés en argument :

```
>addition := proc(a,b)
local res;
res :=a+b;
return res;
end proc;
```

Remarquons que l'on pourrait aussi écrire cette même procédure de la façon suivante :

```
>addition := proc(a,b)
return (a+b);
end proc;
```

Un premier exemple : la procédure suivante effectue l'addition des nombres a et b passés en argument :

```
>addition := proc(a,b)
local res;
res :=a+b;
return res;
end proc;
```

Remarquons que l'on pourrait aussi écrire cette même procédure de la façon suivante :

```
>addition := proc(a,b)
return (a+b);
end proc;
```

Un premier exemple : la procédure suivante effectue l'addition des nombres a et b passés en argument :

```
>addition := proc(a,b)
local res;
res :=a+b;
return res;
end proc;
```

Remarquons que l'on pourrait aussi écrire cette même procédure de la façon suivante :

```
>addition := proc(a,b)
return (a+b);
end proc;
```


Un premier exemple : la procédure suivante effectue l'addition des nombres a et b passés en argument :

```
>addition := proc(a,b)
local res;
res :=a+b;
return res;
end proc;
```

Remarquons que l'on pourrait aussi écrire cette même procédure de la façon suivante :

```
>addition := proc(a,b)
return (a+b);
end proc;
```

Un premier exemple : la procédure suivante effectue l'addition des nombres a et b passés en argument :

```
>addition := proc(a,b)
local res;
res :=a+b;
return res;
end proc;
```

Remarquons que l'on pourrait aussi écrire cette même procédure de la façon suivante :

```
>addition := proc(a,b)
return (a+b);
end proc;
```

Voici comment on utilise la procédure :

```
>addition (10,7) ;
```

17

```
> toto := addition (2,3) ;
```

toto := 5

```
>toto ;
```

5

```
> res ;
```

res

En dehors d'une procédure, on ne peut plus accéder à une variable déclarée localement !

Voici comment on utilise la procédure :

```
>addition (10,7) ;
```

17

```
> toto := addition (2,3) ;
```

toto := 5

```
>toto ;
```

5

```
> res ;
```

res

En dehors d'une procédure, on ne peut plus accéder à une variable déclarée localement !

Voici comment on utilise la procédure :

```
>addition (10,7) ;
```

17

```
> toto := addition (2,3) ;
```

toto := 5

```
>toto ;
```

5

```
> res ;
```

res

En dehors d'une procédure, on ne peut plus accéder à une variable déclarée localement !

Voici comment on utilise la procédure :

```
>addition (10,7) ;
```

17

```
> toto := addition (2,3) ;
```

toto := 5

```
>toto ;
```

5

```
> res ;
```

res

En dehors d'une procédure, on ne peut plus accéder à une variable déclarée localement !

Voici comment on utilise la procédure :

```
>addition (10,7) ;
```

17

```
> toto := addition (2,3) ;
```

toto := 5

```
>toto ;
```

5

```
> res ;
```

res

En dehors d'une procédure, on ne peut plus accéder à une variable déclarée localement !

Voici comment on utilise la procédure :

```
>addition (10,7) ;
```

17

```
> toto := addition (2,3) ;
```

toto := 5

```
>toto ;
```

5

```
> res ;
```

res

En dehors d'une procédure, on ne peut plus accéder à une variable déclarée localement !

Voici comment on utilise la procédure :

```
>addition (10,7) ;
```

17

```
> toto := addition (2,3) ;
```

toto := 5

```
>toto ;
```

5

```
> res ;
```

res

En dehors d'une procédure, on ne peut plus accéder à une variable déclarée localement !

Voici comment on utilise la procédure :

```
>addition (10,7) ;
```

17

```
> toto := addition (2,3) ;
```

toto := 5

```
>toto ;
```

5

```
> res ;
```

res

En dehors d'une procédure, on ne peut plus accéder à une variable déclarée localement !

Voici comment on utilise la procédure :

```
>addition (10,7) ;
```

17

```
> toto := addition (2,3) ;
```

toto := 5

```
>toto ;
```

5

```
> res ;
```

res

En dehors d'une procédure, on ne peut plus accéder à une variable déclarée localement !

Voici comment on utilise la procédure :

```
>addition (10,7) ;
```

17

```
> toto := addition (2,3) ;
```

toto := 5

```
>toto ;
```

5

```
> res ;
```

res

En dehors d'une procédure, on ne peut plus accéder à une variable déclarée localement !

Quelques explications :

Lors de l'appel de la procédure `addition` par l'instruction `"addition (10, 7) ;"`, l'argument formel `a` prend la valeur 10 et l'argument formel `b` prend la valeur 7, et tous les calculs effectués dans `addition` sont faits avec `a` valant 10 et `b` valant 7.

On récupère la valeur calculée par la procédure `addition` dans la variable `toto` grâce à l'instruction `return` (sans cette dernière, on ne peut pas affecter une variable avec le résultat d'une procédure !). Par ailleurs on remarque que la variable `res` est inaccessible à l'extérieur de la procédure `addition` car c'est une variable locale à cette dernière.

Quelques explications :

Lors de l'appel de la procédure `addition` par l'instruction `"addition (10, 7) ;"`, l'argument formel `a` prend la valeur 10 et l'argument formel `b` prend la valeur 7, et tous les calculs effectués dans `addition` sont faits avec `a` valant 10 et `b` valant 7.

On récupère la valeur calculée par la procédure `addition` dans la variable `toto` grâce à l'instruction `return` (sans cette dernière, on ne peut pas affecter une variable avec le résultat d'une procédure !). Par ailleurs on remarque que la variable `res` est inaccessible à l'extérieur de la procédure `addition` car c'est une variable locale à cette dernière.

Quelques explications :

Lors de l'appel de la procédure `addition` par l'instruction `"addition (10, 7) ;"`, l'argument formel `a` prend la valeur **10** et l'argument formel `b` prend la valeur 7, et tous les calculs effectués dans `addition` sont faits avec `a` valant 10 et `b` valant 7.

On récupère la valeur calculée par la procédure `addition` dans la variable `toto` grâce à l'instruction `return` (sans cette dernière, on ne peut pas affecter une variable avec le résultat d'une procédure !). Par ailleurs on remarque que la variable `res` est inaccessible à l'extérieur de la procédure `addition` car c'est une variable locale à cette dernière.

Quelques explications :

Lors de l'appel de la procédure `addition` par l'instruction "`addition (10, 7) ;`", l'argument formel `a` prend la valeur 10 et l'argument formel `b` prend la valeur 7, et tous les calculs effectués dans `addition` sont faits avec `a` valant 10 et `b` valant 7.

On récupère la valeur calculée par la procédure `addition` dans la variable `toto` grâce à l'instruction `return` (sans cette dernière, on ne peut pas affecter une variable avec le résultat d'une procédure !). Par ailleurs on remarque que la variable `res` est inaccessible à l'extérieur de la procédure `addition` car c'est une variable locale à cette dernière.

Quelques explications :

Lors de l'appel de la procédure `addition` par l'instruction "`addition (10, 7) ;`", l'argument formel `a` prend la valeur 10 et l'argument formel `b` prend la valeur 7, et tous les calculs effectués dans `addition` sont faits avec `a` valant 10 et `b` valant 7.

On récupère la valeur calculée par la procédure `addition` dans la variable `toto` grâce à l'instruction `return` (sans cette dernière, on ne peut pas affecter une variable avec le résultat d'une procédure !). Par ailleurs on remarque que la variable `res` est inaccessible à l'extérieur de la procédure `addition` car c'est une variable locale à cette dernière.

Quelques explications :

Lors de l'appel de la procédure `addition` par l'instruction `"addition (10, 7) ;"`, l'argument formel `a` prend la valeur 10 et l'argument formel `b` prend la valeur 7, et tous les calculs effectués dans `addition` sont faits avec `a` valant 10 et `b` valant 7.

On récupère la valeur calculée par la procédure `addition` dans la variable `toto` grâce à l'instruction `return` (sans cette dernière, on ne peut pas affecter une variable avec le résultat d'une procédure !). Par ailleurs on remarque que la variable `res` est inaccessible à l'extérieur de la procédure `addition` car c'est une variable locale à cette dernière.

Quelques explications :

Lors de l'appel de la procédure `addition` par l'instruction `"addition (10, 7) ;"`, l'argument formel `a` prend la valeur 10 et l'argument formel `b` prend la valeur 7, et tous les calculs effectués dans `addition` sont faits avec `a` valant 10 et `b` valant 7.

On récupère la valeur calculée par la procédure `addition` dans la variable `toto` grâce à l'instruction `return` (sans cette dernière, on ne peut pas affecter une variable avec le résultat d'une procédure !). Par ailleurs on remarque que la variable `res` est inaccessible à l'extérieur de la procédure `addition` car c'est une variable locale à cette dernière.

Quelques explications :

Lors de l'appel de la procédure `addition` par l'instruction `"addition (10, 7) ;"`, l'argument formel `a` prend la valeur 10 et l'argument formel `b` prend la valeur 7, et tous les calculs effectués dans `addition` sont faits avec `a` valant 10 et `b` valant 7.

On récupère la valeur calculée par la procédure `addition` dans la variable `toto` grâce à l'instruction `return` (sans cette dernière, on ne peut pas affecter une variable avec le résultat d'une procédure !). Par ailleurs on remarque que la variable `res` est inaccessible à l'extérieur de la procédure `addition` car c'est une variable locale à cette dernière.

Quelques explications :

Lors de l'appel de la procédure `addition` par l'instruction `"addition (10, 7) ;"`, l'argument formel `a` prend la valeur 10 et l'argument formel `b` prend la valeur 7, et tous les calculs effectués dans `addition` sont faits avec `a` valant 10 et `b` valant 7.

On récupère la valeur calculée par la procédure `addition` dans la variable `toto` grâce à l'instruction `return` (sans cette dernière, on ne peut pas affecter une variable avec le résultat d'une procédure !). Par ailleurs on remarque que la variable `res` est inaccessible à l'extérieur de la procédure `addition` car c'est une variable locale à cette dernière.

Quelques explications :

Lors de l'appel de la procédure `addition` par l'instruction `"addition (10, 7) ;"`, l'argument formel `a` prend la valeur 10 et l'argument formel `b` prend la valeur 7, et tous les calculs effectués dans `addition` sont faits avec `a` valant 10 et `b` valant 7.

On récupère la valeur calculée par la procédure `addition` dans la variable `toto` grâce à l'instruction `return` (sans cette dernière, on ne peut pas affecter une variable avec le résultat d'une procédure !). Par ailleurs on remarque que la variable `res` est inaccessible à l'extérieur de la procédure `addition` car c'est une variable locale à cette dernière.

Quelques remarques (très) importantes :

(1) Un argument formel ne peut pas être modifié à l'intérieur d'une procédure. Par ex. :

```
> succ := proc (n)
n :=n+1;
return n;
end proc;
> succ (3) ;
```

Error, in (succ) illegal use of a formal parameter.

Quelques remarques (très) importantes :

(1) Un argument formel ne peut pas être modifié à l'intérieur d'une procédure. Par ex. :

```
> succ := proc (n)
```

```
  n :=n+1;
```

```
  return n;
```

```
end proc;
```

```
> succ (3) ;
```

Error, in (succ) illegal use of a formal parameter.

Quelques remarques (très) importantes :

(1) Un argument formel ne peut pas être modifié à l'intérieur d'une procédure. Par ex. :

```
> succ := proc (n)
```

```
  n :=n+1;
```

```
  return n;
```

```
end proc;
```

```
> succ (3);
```

Error, in (succ) illegal use of a formal parameter.

Quelques remarques (très) importantes :

(1) Un argument formel ne peut pas être modifié à l'intérieur d'une procédure. Par ex. :

```
> succ := proc (n)
```

```
n :=n+1 ;
```

```
return n ;
```

```
end proc ;
```

```
> succ (3) ;
```

Error, in (succ) illegal use of a formal parameter.

Quelques remarques (très) importantes :

(1) Un argument formel ne peut pas être modifié à l'intérieur d'une procédure. Par ex. :

```
> succ := proc (n)
```

```
  n :=n+1 ;
```

```
  return n ;
```

```
end proc ;
```

```
> succ (3) ;
```

Error, in (succ) illegal use of a formal parameter.

Quelques remarques (très) importantes :

(1) Un argument formel ne peut pas être modifié à l'intérieur d'une procédure. Par ex. :

```
> succ := proc (n)
```

```
  n :=n+1 ;
```

```
  return n ;
```

```
end proc ;
```

```
> succ (3) ;
```

Error, in (succ) illegal use of a formal parameter.

Quelques remarques (très) importantes :

(1) Un argument formel ne peut pas être modifié à l'intérieur d'une procédure. Par ex. :

```
> succ := proc (n)
n :=n+1 ;
return n ;
end proc ;
> succ (3) ;
```

Error, in (succ) illegal use of a formal parameter.

Quelques remarques (très) importantes :

(1) Un argument formel ne peut pas être modifié à l'intérieur d'une procédure. Par ex. :

```
> succ := proc (n)
```

```
  n :=n+1 ;
```

```
  return n ;
```

```
end proc ;
```

```
> succ (3) ;
```

Error, in (succ) illegal use of a formal parameter.

Si on veut faire cela, il faut utiliser une variable locale :

```
> succ := proc (n)
  local m;
  m := n+1;
  return m;
end proc;
> succ (3);
```

Si on veut faire cela, il faut utiliser une variable locale :

```
> succ := proc (n)
  local m;
  m := n+1;
  return m;
end proc;
> succ (3);
```


Si on veut faire cela, il faut utiliser une variable locale :

```
> succ := proc (n)
  local m;
  m :=n+1;
  return m;
end proc;
> succ (3) ;
```

Si on veut faire cela, il faut utiliser une variable locale :

```
> succ := proc(n)
local m;
m :=n+1;
return m;
end proc;
> succ (3) ;
```

Si on veut faire cela, il faut utiliser une variable locale :

```
> succ := proc (n)
local m;
m :=n+1;
return m;
end proc;
> succ (3) ;
```

Si on veut faire cela, il faut utiliser une variable locale :

```
> succ := proc(n)
  local m;
  m :=n+1;
  return m;
end proc;
> succ (3) ;
```

Si on veut faire cela, il faut utiliser une variable locale :

```
> succ := proc(n)
  local m;
  m :=n+1;
  return m;
end proc;
> succ (3) ;
```

Si on veut faire cela, il faut utiliser une variable locale :

```
> succ := proc(n)
  local m;
  m :=n+1;
  return m;
end proc;
> succ (3) ;
```

Si on veut faire cela, il faut utiliser une variable locale :

```
> succ := proc(n)
  local m;
  m :=n+1;
  return m;
end proc;
> succ (3) ;
```

(2) On peut définir une procédure sans argument formel :
par exemple une procédure qui ne fait que renvoyer "0" :

```
> zero := proc ()  
return 0 ;  
end proc ;  
>zero () ;
```

0

(2) On peut définir une procédure sans argument formel :
par exemple une procédure qui ne fait que renvoyer "0" :

```
> zero := proc()
```

```
  return 0;
```

```
end proc;
```

```
>zero();
```

0

(2) On peut définir une procédure sans argument formel :
par exemple une procédure qui ne fait que renvoyer "0" :

```
> zero := proc()  
return 0 ;  
end proc ;  
>zero() ;
```

0

(2) On peut définir une procédure sans argument formel :
par exemple une procédure qui ne fait que renvoyer "0" :

```
> zero := proc()  
return 0;  
end proc;  
>zero();
```

0

(2) On peut définir une procédure sans argument formel :
par exemple une procédure qui ne fait que renvoyer "0" :

```
> zero := proc()  
return 0 ;  
end proc ;  
>zero() ;
```

0

(2) On peut définir une procédure sans argument formel :
par exemple une procédure qui ne fait que renvoyer "0" :

```
> zero := proc()  
return 0 ;  
end proc ;  
>zero() ;
```

0

(3) Dès qu'une instruction "return" est exécutée dans une procédure, on sort de celle-ci, et l'exécution du programme reprend après le "end proc ;" final. Par ex. :

```
> fonction := proc(a,b,c)
  return -a;
  return ((a+b)*c);
end proc;
> toto (1,3,7);
```

-1

Dans cet exemple on voit que seul le premier "return" est effectué et pas le second.

(3) Dès qu'une instruction "return" est exécutée dans une procédure, on sort de celle-ci, et l'exécution du programme reprend après le "end proc ;" final. Par ex. :

```
> fonction := proc(a,b,c)
  return -a ;
  return ((a+b)*c) ;
end proc ;
> toto (1,3,7) ;
```

-1

Dans cet exemple on voit que seul le premier "return" est effectué et pas le second.

(3) Dès qu'une instruction "return" est exécutée dans une procédure, on sort de celle-ci, et l'exécution du programme reprend après le "end proc ;" final. Par ex. :

```
> fonction := proc(a,b,c)
  return -a;
  return ((a+b)*c);
end proc;
> toto (1,3,7);
```

-1

Dans cet exemple on voit que seul le premier "return" est effectué et pas le second.

(3) Dès qu'une instruction "return" est exécutée dans une procédure, on sort de celle-ci, et l'exécution du programme reprend après le "end proc ;" final. Par ex. :

```
> fonction := proc(a,b,c)
return -a;
return ((a+b)*c);
end proc;
> toto (1,3,7);
```

-1

Dans cet exemple on voit que seul le premier "return" est effectué et pas le second.

(3) Dès qu'une instruction "return" est exécutée dans une procédure, on sort de celle-ci, et l'exécution du programme reprend après le "end proc ;" final. Par ex. :

```
> fonction := proc(a,b,c)
  return -a;
  return ((a+b)*c);
end proc;
> toto (1,3,7);
```

-1

Dans cet exemple on voit que seul le premier "return" est effectué et pas le second.

(3) Dès qu'une instruction "return" est exécutée dans une procédure, on sort de celle-ci, et l'exécution du programme reprend après le "end proc ;" final. Par ex. :

```
> fonction := proc(a,b,c)
return -a;
return ((a+b)*c);
end proc;
> toto (1,3,7);
```

-1

Dans cet exemple on voit que seul le premier "return" est effectué et pas le second.

(3) Dès qu'une instruction "return" est exécutée dans une procédure, on sort de celle-ci, et l'exécution du programme reprend après le "end proc ;" final. Par ex. :

```
> fonction := proc(a,b,c)
return -a;
return ((a+b)*c);
end proc;
> toto (1,3,7);
```

-1

Dans cet exemple on voit que seul le premier "return" est effectué et pas le second.

(3) Dès qu'une instruction "return" est exécutée dans une procédure, on sort de celle-ci, et l'exécution du programme reprend après le "end proc ;" final. Par ex. :

```
> fonction := proc(a,b,c)
return -a;
return ((a+b)*c);
end proc;
> toto (1,3,7);
```

-1

Dans cet exemple on voit que seul le premier "return" est effectué et pas le second.

(3) Dès qu'une instruction "return" est exécutée dans une procédure, on sort de celle-ci, et l'exécution du programme reprend après le "end proc ;" final. Par ex. :

```
> fonction := proc(a,b,c)
  return -a;
  return ((a+b)*c);
end proc;
> toto (1,3,7);
```

-1

Dans cet exemple on voit que seul le premier "return" est effectué et pas le second.

(3) Dès qu'une instruction "return" est exécutée dans une procédure, on sort de celle-ci, et l'exécution du programme reprend après le "end proc ;" final. Par ex. :

```
> fonction := proc(a,b,c)
  return -a;
  return ((a+b)*c);
end proc;
> toto (1,3,7);
```

-1

Dans cet exemple on voit que seul le premier "return" est effectué et pas le second.

(3) Dès qu'une instruction "return" est exécutée dans une procédure, on sort de celle-ci, et l'exécution du programme reprend après le "end proc ;" final. Par ex. :

```
> fonction := proc(a,b,c)
return -a;
return ((a+b)*c);
end proc;
> toto (1,3,7);
```

-1

Dans cet exemple on voit que seul le premier "return" est effectué et pas le second.

Si l'on souhaite renvoyer deux valeurs, on doit retourner une séquence.

```
> fonction := proc(a,b,c)
return (-a, (a+b)*c);
end proc;
> toto (1,3,7);
```

-1,28

Si l'on souhaite renvoyer deux valeurs, on doit retourner une séquence.

```
> fonction := proc(a,b,c)
return (-a, (a+b)*c) ;
end proc ;
> toto (1,3,7) ;
```

-1,28

Si l'on souhaite renvoyer deux valeurs, on doit retourner une séquence.

```
> fonction := proc(a,b,c)
return (-a, (a+b)*c) ;
end proc ;
> toto (1,3,7) ;
```

-1,28

Si l'on souhaite renvoyer deux valeurs, on doit retourner une séquence.

```
> fonction := proc(a,b,c)
return (-a, (a+b)*c) ;
end proc ;
> toto (1,3,7) ;
```

-1,28

Si l'on souhaite renvoyer deux valeurs, on doit retourner une séquence.

```
> fonction := proc(a,b,c)
return (-a, (a+b)*c) ;
end proc ;
> toto (1,3,7) ;
```

-1,28

Si l'on souhaite renvoyer deux valeurs, on doit retourner une séquence.

```
> fonction := proc(a,b,c)
return (-a, (a+b)*c) ;
end proc ;
> toto (1,3,7) ;
```

-1,28

Si l'on souhaite renvoyer deux valeurs, on doit retourner une séquence.

```
> fonction := proc(a,b,c)
return (-a, (a+b)*c) ;
end proc ;
> toto (1,3,7) ;
```

-1,28

Un autre exemple de procédure : la procédure suivante crée la liste des produits " $i * x$ " où i varie de 0 jusqu'à l'entier n et x est une variable :

```
> liste := proc (n)
local L,i;
L :=[seq(i*x,i=0..n)];
return L;
end proc;
> liste (5);
```

[0, x, 2x, 3x, 4x, 5x]

Un autre exemple de procédure : la procédure suivante crée la liste des produits " $i * x$ " où i varie de 0 jusqu'à l'entier n et x est une variable :

```
> liste := proc (n)
local L,i;
L :=[seq(i*x,i=0..n)];
return L;
end proc;
> liste (5);
```

[0, x, 2x, 3x, 4x, 5x]

Un autre exemple de procédure : la procédure suivante crée la liste des produits " $i * x$ " où i varie de 0 jusqu'à l'entier n et x est une variable :

```
> liste := proc (n)
  local L,i;
  L :=[seq(i*x,i=0..n)];
  return L;
end proc;
> liste (5);
```

[0, x, 2x, 3x, 4x, 5x]

Un autre exemple de procédure : la procédure suivante crée la liste des produits " $i * x$ " où i varie de 0 jusqu'à l'entier n et x est une variable :

```
> liste := proc (n)
local L,i;
L :=[seq(i*x,i=0..n)];
return L;
end proc;
> liste (5);
```

[0, x, 2x, 3x, 4x, 5x]

Un autre exemple de procédure : la procédure suivante crée la liste des produits " $i * x$ " où i varie de 0 jusqu'à l'entier n et x est une variable :

```
> liste := proc (n)
local L,i;
L :=[seq(i*x,i=0..n)] ;
return L;
end proc;
> liste (5);
```

[0, x, 2x, 3x, 4x, 5x]

Un autre exemple de procédure : la procédure suivante crée la liste des produits " $i * x$ " où i varie de 0 jusqu'à l'entier n et x est une variable :

```
> liste := proc (n)
local L,i;
L :=[seq(i*x,i=0..n)] ;
return L;
end proc;
> liste (5);
```

[0, x, 2x, 3x, 4x, 5x]

Un autre exemple de procédure : la procédure suivante crée la liste des produits " $i * x$ " où i varie de 0 jusqu'à l'entier n et x est une variable :

```
> liste := proc (n)
local L,i;
L :=[seq(i*x,i=0..n)] ;
return L;
end proc;
> liste (5);
```

[0, x, 2x, 3x, 4x, 5x]

Un autre exemple de procédure : la procédure suivante crée la liste des produits " $i * x$ " où i varie de 0 jusqu'à l'entier n et x est une variable :

```
> liste := proc (n)
local L,i;
L :=[seq(i*x,i=0..n)] ;
return L ;
end proc ;
> liste (5) ;
```

[0, x, 2x, 3x, 4x, 5x]

Un autre exemple de procédure : la procédure suivante crée la liste des produits " $i * x$ " où i varie de 0 jusqu'à l'entier n et x est une variable :

```
> liste := proc (n)
local L,i;
L :=[seq(i*x,i=0..n)] ;
return L;
end proc;
> liste (5);
```

$[0, x, 2x, 3x, 4x, 5x]$

Instruction conditionnelle `if`

Chap. III :
Programmation

Laurent
Poincot

La syntaxe de l'instruction conditionnelle `if` est la suivante :

```
> if <condition> then  
<instruction1>;  
<instruction2>;  
...  
[else  
<instructionA>;  
<instructionB>;  
...]  
end if;
```

Instruction conditionnelle `if`

Chap. III :
Programmation

Laurent
Poinsot

La syntaxe de l'instruction conditionnelle `if` est la suivante :

```
> if <condition> then  
  <instruction1>;  
  <instruction2>;  
  ...  
  [else  
    <instructionA>;  
    <instructionB>;  
    ...]  
end if;
```

Instruction conditionnelle `if`

Chap. III :
Programmation

Laurent
Poincot

La syntaxe de l'instruction conditionnelle `if` est la suivante :

```
> if <condition> then  
<instruction1>;  
<instruction2>;  
...  
[else  
<instructionA>;  
<instructionB>;  
...]  
end if;
```

Instruction conditionnelle `if`

Chap. III :
Programmation

Laurent
Poinsot

La syntaxe de l'instruction conditionnelle `if` est la suivante :

```
> if <condition> then  
<instruction1>;  
<instruction2>;  
...  
[else  
<instructionA>;  
<instructionB>;  
...]  
end if;
```

Instruction conditionnelle `if`

Chap. III :
Programmation

Laurent
Poincot

La syntaxe de l'instruction conditionnelle `if` est la suivante :

```
> if <condition> then  
<instruction1>;  
<instruction2>;  
...  
[else  
<instructionA>;  
<instructionB>;  
...]  
end if;
```

Instruction conditionnelle `if`

Chap. III :
Programmation

Laurent
Poincot

La syntaxe de l'instruction conditionnelle `if` est la suivante :

```
> if <condition> then  
<instruction1>;  
<instruction2>;  
...  
[else  
<instructionA>;  
<instructionB>;  
...]  
end if;
```

Instruction conditionnelle `if`

Chap. III :
Programmation

Laurent
Poincot

La syntaxe de l'instruction conditionnelle `if` est la suivante :

```
> if <condition> then  
<instruction1>;  
<instruction2>;  
...  
[else  
<instructionA>;  
<instructionB>;  
...]  
end if;
```

Instruction conditionnelle `if`

Chap. III :
Programmation

Laurent
Poincot

La syntaxe de l'instruction conditionnelle `if` est la suivante :

```
> if <condition> then  
<instruction1>;  
<instruction2>;  
...  
[else  
<instructionA>;  
<instructionB>;  
...]  
end if;
```


Instruction conditionnelle `if`

Chap. III :
Programmation

Laurent
Poincot

La syntaxe de l'instruction conditionnelle `if` est la suivante :

```
> if <condition> then  
<instruction1>;  
<instruction2>;  
...  
[else  
<instructionA>;  
<instructionB>;  
...]  
end if;
```

Instruction conditionnelle `if`

Chap. III :
Programmation

Laurent
Poincot

La syntaxe de l'instruction conditionnelle `if` est la suivante :

```
> if <condition> then  
<instruction1>;  
<instruction2>;  
...  
[else  
<instructionA>;  
<instructionB>;  
...]  
end if;
```

Quelques explications

- 1 Si la `<condition>` est vraie (c'est-à-dire qu'elle est égale au booléen `true`), alors les instructions `<instruction1>`, `<instruction2>`, ..., sont exécutées. Puis l'exécution reprend après l'instruction finale `"end if;"`;
- 2 Si la `<condition>` est fausse (autrement dit elle a pour valeur `false`), alors l'exécution se poursuit directement après l'instruction finale `"end if;"`;
- 3 Toutefois, si l'instruction optionnelle `"else"` est spécifiée, et toujours dans le cas où `<condition>` est fausse, alors les instructions `<instructionA>`, `<instructionB>`, ..., sont exécutées. Puis l'exécution du programme reprend après l'instruction finale `"end if;"`;
- 4 L'instruction conditionnelle `if` se termine par l'instruction `"end if;"`.

Quelques explications

- 1 Si la `<condition>` est vraie (c'est-à-dire qu'elle est égale au booléen `true`), alors les instructions `<instruction1>`, `<instruction2>`, ..., sont exécutées. Puis l'exécution reprend après l'instruction finale `"end if;"`;
- 2 Si la `<condition>` est fausse (autrement dit elle a pour valeur `false`), alors l'exécution se poursuit directement après l'instruction finale `"end if;"`;
- 3 Toutefois, si l'instruction optionnelle `"else"` est spécifiée, et toujours dans le cas où `<condition>` est fausse, alors les instructions `<instructionA>`, `<instructionB>`, ..., sont exécutées. Puis l'exécution du programme reprend après l'instruction finale `"end if;"`;
- 4 L'instruction conditionnelle `if` se termine par l'instruction `"end if;"`.

Quelques explications

- 1 Si la `<condition>` est vraie (c'est-à-dire qu'elle est égale au booléen `true`), alors les instructions `<instruction1>`, `<instruction2>`, ..., sont exécutées. Puis l'exécution reprend après l'instruction finale `"end if;"`;
- 2 Si la `<condition>` est fausse (autrement dit elle a pour valeur `false`), alors l'exécution se poursuit directement après l'instruction finale `"end if;"`;
- 3 Toutefois, si l'instruction optionnelle `"else"` est spécifiée, et toujours dans le cas où `<condition>` est fausse, alors les instructions `<instructionA>`, `<instructionB>`, ..., sont exécutées. Puis l'exécution du programme reprend après l'instruction finale `"end if;"`;
- 4 L'instruction conditionnelle `if` se termine par l'instruction `"end if;"`.

Quelques explications

- 1 Si la `<condition>` est vraie (c'est-à-dire qu'elle est égale au booléen `true`), alors les instructions `<instruction1>`, `<instruction2>`, ..., sont exécutées. Puis l'exécution reprend après l'instruction finale `"end if;"`;
- 2 Si la `<condition>` est fausse (autrement dit elle a pour valeur `false`), alors l'exécution se poursuit directement après l'instruction finale `"end if;"`;
- 3 Toutefois, si l'instruction optionnelle `"else"` est spécifiée, et toujours dans le cas où `<condition>` est fausse, alors les instructions `<instructionA>`, `<instructionB>`, ..., sont exécutées. Puis l'exécution du programme reprend après l'instruction finale `"end if;"`;
- 4 L'instruction conditionnelle `if` se termine par l'instruction `"end if;"`.

Quelques explications

- 1 Si la `<condition>` est vraie (c'est-à-dire qu'elle est égale au booléen `true`), alors les instructions `<instruction1>`, `<instruction2>`, ..., sont exécutées. Puis l'exécution reprend après l'instruction finale `"end if ;"`;
- 2 Si la `<condition>` est fausse (autrement dit elle a pour valeur `false`), alors l'exécution se poursuit directement après l'instruction finale `"end if ;"`;
- 3 Toutefois, si l'instruction optionnelle `"else"` est spécifiée, et toujours dans le cas où `<condition>` est fausse, alors les instructions `<instructionA>`, `<instructionB>`, ..., sont exécutées. Puis l'exécution du programme reprend après l'instruction finale `"end if ;"`;
- 4 L'instruction conditionnelle `if` se termine par l'instruction `"end if ;"`.

Quelques explications

- 1 Si la `<condition>` est vraie (c'est-à-dire qu'elle est égale au booléen `true`), alors les instructions `<instruction1>`, `<instruction2>`, ..., sont exécutées. Puis l'exécution reprend après l'instruction finale `"end if ;"`;
- 2 Si la `<condition>` est fausse (autrement dit elle a pour valeur `false`), alors l'exécution se poursuit directement après l'instruction finale `"end if ;"`;
- 3 Toutefois, si l'instruction optionnelle `"else"` est spécifiée, et toujours dans le cas où `<condition>` est fausse, alors les instructions `<instructionA>`, `<instructionB>`, ..., sont exécutées. Puis l'exécution du programme reprend après l'instruction finale `"end if ;"`;
- 4 L'instruction conditionnelle `if` se termine par l'instruction `"end if ;"`.

Quelques explications

- 1** Si la `<condition>` est vraie (c'est-à-dire qu'elle est égale au booléen `true`), alors les instructions `<instruction1>`, `<instruction2>`, ..., sont exécutées. Puis l'exécution reprend après l'instruction finale `"end if;"`;
- 2** Si la `<condition>` est fausse (autrement dit elle a pour valeur `false`), alors l'exécution se poursuit directement après l'instruction finale `"end if;"`;
- 3** Toutefois, si l'instruction optionnelle `"else"` est spécifiée, et toujours dans le cas où `<condition>` est fausse, alors les instructions `<instructionA>`, `<instructionB>`, ..., sont exécutées. Puis l'exécution du programme reprend après l'instruction finale `"end if;"`;
- 4** L'instruction conditionnelle `if` se termine par l'instruction `"end if;"`.

Quelques explications

- 1 Si la `<condition>` est vraie (c'est-à-dire qu'elle est égale au booléen `true`), alors les instructions `<instruction1>`, `<instruction2>`, ..., sont exécutées. Puis l'exécution reprend après l'instruction finale `"end if;"`;
- 2 Si la `<condition>` est fausse (autrement dit elle a pour valeur `false`), alors l'exécution se poursuit directement après l'instruction finale `"end if;"`;
- 3 Toutefois, si l'instruction optionnelle `"else"` est spécifiée, et toujours dans le cas où `<condition>` est fausse, alors les instructions `<instructionA>`, `<instructionB>`, ..., sont exécutées. Puis l'exécution du programme reprend après l'instruction finale `"end if;"`;
- 4 L'instruction conditionnelle `if` se termine par l'instruction `"end if;"`.

Quelques explications

- 1 Si la `<condition>` est vraie (c'est-à-dire qu'elle est égale au booléen `true`), alors les instructions `<instruction1>`, `<instruction2>`, ..., sont exécutées. Puis l'exécution reprend après l'instruction finale `"end if;"`;
- 2 Si la `<condition>` est fausse (autrement dit elle a pour valeur `false`), alors l'exécution se poursuit directement après l'instruction finale `"end if;"`;
- 3 Toutefois, si l'instruction optionnelle `"else"` est spécifiée, et toujours dans le cas où `<condition>` est fausse, alors les instructions `<instructionA>`, `<instructionB>`, ..., sont exécutées. Puis l'exécution du programme reprend après l'instruction finale `"end if;"`;
- 4 L'instruction conditionnelle `if` se termine par l'instruction `"end if;"`.

Quelques explications

- 1 Si la `<condition>` est vraie (c'est-à-dire qu'elle est égale au booléen `true`), alors les instructions `<instruction1>`, `<instruction2>`, ..., sont exécutées. Puis l'exécution reprend après l'instruction finale `"end if ;"`;
- 2 Si la `<condition>` est fausse (autrement dit elle a pour valeur `false`), alors l'exécution se poursuit directement après l'instruction finale `"end if ;"`;
- 3 Toutefois, si l'instruction optionnelle `"else"` est spécifiée, et toujours dans le cas où `<condition>` est fausse, alors les instructions `<instructionA>`, `<instructionB>`, ..., sont exécutées. Puis l'exécution du programme reprend après l'instruction finale `"end if ;"`;
- 4 L'instruction conditionnelle `if` se termine par l'instruction `"end if ;"`.

Quelques explications

- 1 Si la `<condition>` est vraie (c'est-à-dire qu'elle est égale au booléen `true`), alors les instructions `<instruction1>`, `<instruction2>`, ..., sont exécutées. Puis l'exécution reprend après l'instruction finale `"end if ;"`;
- 2 Si la `<condition>` est fausse (autrement dit elle a pour valeur `false`), alors l'exécution se poursuit directement après l'instruction finale `"end if ;"`;
- 3 Toutefois, si l'instruction optionnelle `"else"` est spécifiée, et toujours dans le cas où `<condition>` est fausse, alors les instructions `<instructionA>`, `<instructionB>`, ..., sont exécutées. Puis l'exécution du programme reprend après l'instruction finale `"end if ;"`;
- 4 L'instruction conditionnelle `if` se termine par l'instruction `"end if ;"`.

Test si un entier est pair ou non

Chap. III :
Programmation

Laurent
Poinsot

```
> pair := proc (n)
  local result;
  if n mod 2 = 0 then
    result := true;
  else
    result := false;
  end if;
  return result;
end proc;
```

```
> pair (6);
```

true

```
> pair (15);
```

false

Test si un entier est pair ou non

Chap. III :
Programmation

Laurent
Poinsot

```
> pair := proc (n)
  local result;
  if n mod 2 = 0 then
    result := true;
  else
    result := false;
  end if;
  return result;
end proc;
```

```
> pair (6);
```

true

```
> pair (15);
```

false

Test si un entier est pair ou non

Chap. III :
Programmation

Laurent
Poinsot

```
> pair := proc (n)
local result;
if n mod 2 = 0 then
result := true;
else
result := false;
end if;
return result;
end proc;
```

```
> pair (6);
```

true

```
> pair (15);
```

false

Test si un entier est pair ou non

Chap. III :
Programmation

Laurent
Poinsot

```
> pair := proc (n)
local result;
if n mod 2 = 0 then
result := true;
else
result := false;
end if;
return result;
end proc;
```

```
> pair (6);
```

true

```
> pair (15);
```

false

Test si un entier est pair ou non

Chap. III :
Programmation

Laurent
Poinsot

```
> pair := proc (n)
local result;
if n mod 2 = 0 then
result := true;
else
result := false;
end if;
return result;
end proc;
```

```
> pair (6);
```

true

```
> pair (15);
```

false

Test si un entier est pair ou non

Chap. III :
Programmation

Laurent
Poinsot

```
> pair := proc (n)
local result;
if n mod 2 = 0 then
result := true;
else
result := false;
end if;
return result;
end proc;
```

```
> pair (6);
```

true

```
> pair (15);
```

false

Test si un entier est pair ou non

Chap. III :
Programmation

Laurent
Poinsot

```
> pair := proc (n)
  local result;
  if n mod 2 = 0 then
    result := true;
  else
    result := false;
  end if;
  return result;
end proc;
```

```
> pair (6);
```

true

```
> pair (15);
```

false

Test si un entier est pair ou non

Chap. III :
Programmation

Laurent
Poinsot

```
> pair := proc (n)
local result;
if n mod 2 = 0 then
result := true;
else
result := false;
end if;
return result;
end proc;
```

```
> pair (6);
```

true

```
> pair (15);
```

false

Test si un entier est pair ou non

Chap. III :
Programmation

Laurent
Poinsot

```
> pair := proc (n)
  local result;
  if n mod 2 = 0 then
    result := true;
  else
    result := false;
  end if;
  return result;
end proc;
```

```
> pair (6);
```

true

```
> pair (15);
```

false

Test si un entier est pair ou non

Chap. III :
Programmation

Laurent
Poinsot

```
> pair := proc (n)
  local result;
  if n mod 2 = 0 then
    result := true;
  else
    result := false;
  end if;
  return result;
end proc;
```

```
> pair (6);
```

true

```
> pair (15);
```

false

Test si un entier est pair ou non

```
> pair := proc (n)
local result;
if n mod 2 = 0 then
result := true;
else
result := false;
end if;
return result;
end proc;
> pair (6);
```

true

```
> pair (15);
```

false

Test si un entier est pair ou non

Chap. III :
Programmation

Laurent
Poinsot

```
> pair := proc (n)
  local result;
  if n mod 2 = 0 then
    result := true;
  else
    result := false;
  end if;
  return result;
end proc;
> pair (6);
```

true

```
> pair (15);
```

false

Test si un entier est pair ou non

Chap. III :
Programmation

Laurent
Poinsot

```
> pair := proc (n)
  local result;
  if n mod 2 = 0 then
    result := true;
  else
    result := false;
  end if;
  return result;
end proc;
```

```
> pair (6);
```

true

```
> pair (15);
```

false

Test si un entier est pair ou non

Chap. III :
Programmation

Laurent
Poinsot

```
> pair := proc (n)
  local result;
  if n mod 2 = 0 then
    result := true;
  else
    result := false;
  end if;
  return result;
end proc;
```

```
> pair (6);
```

true

```
> pair (15);
```

false

Boucle `for`

Chap. III :
Programmation

Laurent
Poinsot

La syntaxe de l'instruction d'itération `for` est la suivante :

```
for <compteur> from <début> to <fin> [by  
<pas>] do  
<instruction1>;  
<instruction2>;  
...  
end do;
```

Boucle `for`

Chap. III :
Programmation

Laurent
Poinsot

La syntaxe de l'instruction d'itération `for` est la suivante :

```
for <compteur> from <début> to <fin> [by  
<pas>] do  
  <instruction1>;  
  <instruction2>;  
  ...  
end do;
```

Boucle `for`

Chap. III :
Programmation

Laurent
Poinsot

La syntaxe de l'instruction d'itération `for` est la suivante :

```
for <compteur> from <début> to <fin> [by  
<pas>] do  
<instruction1>;  
<instruction2>;  
...  
end do;
```

Boucle `for`

Chap. III :
Programmation

Laurent
Poinsot

La syntaxe de l'instruction d'itération `for` est la suivante :

```
for <compteur> from <début> to <fin> [by  
<pas>] do  
<instruction1>;  
<instruction2>;  
...  
end do;
```

Boucle `for`

Chap. III :
Programmation

Laurent
Poinsot

La syntaxe de l'instruction d'itération `for` est la suivante :

```
for <compteur> from <début> to <fin> [by  
<pas>] do  
<instruction1>;  
<instruction2>;  
...  
end do;
```


Boucle `for`

La syntaxe de l'instruction d'itération `for` est la suivante :

```
for <compteur> from <début> to <fin> [by  
<pas>] do  
<instruction1>;  
<instruction2>;  
...  
end do;
```

Quelques explications

- 1 La variable entière `<compteur>` est incrémentée (de +1), en partant de l'entier `<début>` et jusqu'à l'entier `<fin>` inclus ;
- 2 Si l'instruction optionnelle `by` est spécifiée, alors la variable `<compteur>` est incrémentée de la valeur `<pas>` ;
- 3 Pour chaque valeur de la variable `<compteur>`, les instructions `<instruction1>`, `<instruction2>`, ... sont exécutées ;
- 4 Une fois que `<compteur>` est égale à `<fin>`, on exécute une **dernière** fois les `<instruction1>`, `<instruction2>`, ... puis la boucle s'arrête et l'exécution du programme reprend après ce qui suit l'instruction finale `"end do ;"`.

Quelques explications

- 1 La variable entière `<compteur>` est incrémentée (de +1), en partant de l'entier `<début>` et jusqu'à l'entier `<fin>` inclus ;
- 2 Si l'instruction optionnelle `by` est spécifiée, alors la variable `<compteur>` est incrémentée de la valeur `<pas>` ;
- 3 Pour chaque valeur de la variable `<compteur>`, les instructions `<instruction1>`, `<instruction2>`, ... sont exécutées ;
- 4 Une fois que `<compteur>` est égale à `<fin>`, on exécute une **dernière** fois les `<instruction1>`, `<instruction2>`, ... puis la boucle s'arrête et l'exécution du programme reprend après ce qui suit l'instruction finale `"end do ;"`.

Quelques explications

- 1** La variable entière `<compteur>` est incrémentée (de +1), en partant de l'entier `<début>` et jusqu'à l'entier `<fin>` inclus ;
- 2** Si l'instruction optionnelle `by` est spécifiée, alors la variable `<compteur>` est incrémentée de la valeur `<pas>` ;
- 3** Pour chaque valeur de la variable `<compteur>`, les instructions `<instruction1>`, `<instruction2>`, ... sont exécutées ;
- 4** Une fois que `<compteur>` est égale à `<fin>`, on exécute une **dernière** fois les `<instruction1>`, `<instruction2>`, ... puis la boucle s'arrête et l'exécution du programme reprend après ce qui suit l'instruction finale `"end do ;"`.

Quelques explications

- 1** La variable entière `<compteur>` est incrémentée (de +1), en partant de l'entier `<début>` et jusqu'à l'entier `<fin>` inclus ;
- 2** Si l'instruction optionnelle `by` est spécifiée, alors la variable `<compteur>` est incrémentée de la valeur `<pas>` ;
- 3** Pour chaque valeur de la variable `<compteur>`, les instructions `<instruction1>`, `<instruction2>`, ... sont exécutées ;
- 4** Une fois que `<compteur>` est égale à `<fin>`, on exécute une **dernière** fois les `<instruction1>`, `<instruction2>`, ... puis la boucle s'arrête et l'exécution du programme reprend après ce qui suit l'instruction finale `"end do ;"`.

Quelques explications

- 1** La variable entière `<compteur>` est incrémentée (de +1), en partant de l'entier `<début>` et jusqu'à l'entier `<fin>` inclus ;
- 2** Si l'instruction optionnelle `by` est spécifiée, alors la variable `<compteur>` est incrémentée de la valeur `<pas>` ;
- 3** Pour chaque valeur de la variable `<compteur>`, les instructions `<instruction1>`, `<instruction2>`, ... sont exécutées ;
- 4** Une fois que `<compteur>` est égale à `<fin>`, on exécute une **dernière** fois les `<instruction1>`, `<instruction2>`, ... puis la boucle s'arrête et l'exécution du programme reprend après ce qui suit l'instruction finale `"end do ;"`.

Quelques explications

- 1** La variable entière `<compteur>` est incrémentée (de +1), en partant de l'entier `<début>` et jusqu'à l'entier `<fin>` inclus ;
- 2** Si l'instruction optionnelle `by` est spécifiée, alors la variable `<compteur>` est incrémentée de la valeur `<pas>` ;
- 3** Pour chaque valeur de la variable `<compteur>`, les instructions `<instruction1>`, `<instruction2>`, ... sont exécutées ;
- 4** Une fois que `<compteur>` est égale à `<fin>`, on exécute une **dernière** fois les `<instruction1>`, `<instruction2>`, ... puis la boucle s'arrête et l'exécution du programme reprend après ce qui suit l'instruction finale `"end do ;"`.

Quelques explications

- 1** La variable entière `<compteur>` est incrémentée (de +1), en partant de l'entier `<début>` et jusqu'à l'entier `<fin>` inclus ;
- 2** Si l'instruction optionnelle `by` est spécifiée, alors la variable `<compteur>` est incrémentée de la valeur `<pas>` ;
- 3** Pour chaque valeur de la variable `<compteur>`, les instructions `<instruction1>`, `<instruction2>`, ... sont exécutées ;
- 4** Une fois que `<compteur>` est égale à `<fin>`, on exécute une **dernière fois** les `<instruction1>`, `<instruction2>`, ... puis la boucle s'arrête et l'exécution du programme reprend après ce qui suit l'instruction finale `"end do ;"`.

Quelques explications

- 1 La variable entière `<compteur>` est incrémentée (de +1), en partant de l'entier `<début>` et jusqu'à l'entier `<fin>` inclus ;
- 2 Si l'instruction optionnelle `by` est spécifiée, alors la variable `<compteur>` est incrémentée de la valeur `<pas>` ;
- 3 Pour chaque valeur de la variable `<compteur>`, les instructions `<instruction1>`, `<instruction2>`, ... sont exécutées ;
- 4 Une fois que `<compteur>` est égale à `<fin>`, on exécute une **dernière** fois les `<instruction1>`, `<instruction2>`, ... puis la boucle s'arrête et l'exécution du programme reprend après ce qui suit l'instruction finale `"end do ;"`.

Exemple

Un petit exemple qui calcule, à l'aide d'une boucle `for`, la somme $\sum_{i=1}^k i * n = 1 * n + 2 * n + \dots + k * n$ où les entiers n et k sont choisis par l'utilisateur :

```
> somme := proc (n,k)
local S,i;
S :=0;
for i from 1 to k do
S :=S+i*n;
end do;
return S;
end proc;
> somme (2,3);
```

Exemple

Un petit exemple qui calcule, à l'aide d'une boucle `for`, la somme $\sum_{i=1}^k i * n = 1 * n + 2 * n + \dots + k * n$ où les entiers n et k sont choisis par l'utilisateur :

```
> somme := proc (n,k)
local S,i;
S :=0;
for i from 1 to k do
S :=S+i*n;
end do;
return S;
end proc;
> somme (2,3);
```

Exemple

Un petit exemple qui calcule, à l'aide d'une boucle `for`, la somme $\sum_{i=1}^k i * n = 1 * n + 2 * n + \dots + k * n$ où les entiers n

et k sont choisis par l'utilisateur :

```
> somme := proc (n,k)
local S,i;
S :=0;
for i from 1 to k do
S :=S+i*n;
end do;
return S;
end proc;
> somme (2,3);
```

Exemple

Un petit exemple qui calcule, à l'aide d'une boucle `for`, la somme $\sum_{i=1}^k i * n = 1 * n + 2 * n + \dots + k * n$ où les entiers n et k sont choisis par l'utilisateur :

```
> somme := proc (n,k)
local S,i;
S :=0;
for i from 1 to k do
S :=S+i*n;
end do;
return S;
end proc;
> somme (2,3);
```

Exemple

Un petit exemple qui calcule, à l'aide d'une boucle `for`, la somme $\sum_{i=1}^k i * n = 1 * n + 2 * n + \dots + k * n$ où les entiers n et k sont choisis par l'utilisateur :

```
> somme := proc (n,k)
local S,i;
S :=0;
for i from 1 to k do
S :=S+i*n;
end do;
return S;
end proc;
> somme (2,3);
```

Exemple

Un petit exemple qui calcule, à l'aide d'une boucle `for`, la somme $\sum_{i=1}^k i * n = 1 * n + 2 * n + \dots + k * n$ où les entiers n et k sont choisis par l'utilisateur :

```
> somme := proc (n,k)
local S,i;
S :=0;
for i from 1 to k do
S :=S+i*n;
end do;
return S;
end proc;
> somme (2,3);
```

Exemple

Un petit exemple qui calcule, à l'aide d'une boucle `for`, la somme $\sum_{i=1}^k i * n = 1 * n + 2 * n + \dots + k * n$ où les entiers n et k sont choisis par l'utilisateur :

```
> somme := proc (n,k)
local S,i;
S :=0;
for i from 1 to k do
S :=S+i*n;
end do;
return S;
end proc;
> somme (2,3);
```


Exemple

Un petit exemple qui calcule, à l'aide d'une boucle `for`, la somme $\sum_{i=1}^k i * n = 1 * n + 2 * n + \dots + k * n$ où les entiers n et k sont choisis par l'utilisateur :

```
> somme := proc (n,k)
local S,i;
S :=0;
for i from 1 to k do
S :=S+i*n;
end do;
return S;
end proc;
> somme (2,3);
```

Exemple

Un petit exemple qui calcule, à l'aide d'une boucle `for`, la somme $\sum_{i=1}^k i * n = 1 * n + 2 * n + \dots + k * n$ où les entiers n et k sont choisis par l'utilisateur :

```
> somme := proc (n,k)
local S,i;
S :=0;
for i from 1 to k do
S :=S+i*n;
end do;
return S;
end proc;
> somme (2,3);
```

Exemple

Un petit exemple qui calcule, à l'aide d'une boucle `for`, la somme $\sum_{i=1}^k i * n = 1 * n + 2 * n + \dots + k * n$ où les entiers n et k sont choisis par l'utilisateur :

```
> somme := proc (n,k)
local S,i;
S :=0;
for i from 1 to k do
S :=S+i*n;
end do;
return S;
end proc;
> somme (2,3);
```

Exemple

Un petit exemple qui calcule, à l'aide d'une boucle `for`, la somme $\sum_{i=1}^k i * n = 1 * n + 2 * n + \dots + k * n$ où les entiers n et k sont choisis par l'utilisateur :

```
> somme := proc (n,k)
local S,i;
S :=0;
for i from 1 to k do
S :=S+i*n;
end do;
return S;
end proc;
> somme (2,3);
```

Exemple

Un petit exemple qui calcule, à l'aide d'une boucle `for`, la somme $\sum_{i=1}^k i * n = 1 * n + 2 * n + \dots + k * n$ où les entiers n et k sont choisis par l'utilisateur :

```
> somme := proc (n,k)
local S,i;
S :=0;
for i from 1 to k do
S :=S+i*n;
end do;
return S;
end proc;
> somme (2,3);
```

Exemple

Un petit exemple qui calcule, à l'aide d'une boucle `for`, la somme $\sum_{i=1}^k i * n = 1 * n + 2 * n + \dots + k * n$ où les entiers n et k sont choisis par l'utilisateur :

```
> somme := proc (n,k)
local S,i;
S :=0;
for i from 1 to k do
S :=S+i*n;
end do;
return S;
end proc;
> somme (2,3);
```

Exemple

Un petit exemple qui calcule, à l'aide d'une boucle `for`, la somme $\sum_{i=1}^k i * n = 1 * n + 2 * n + \dots + k * n$ où les entiers n et k sont choisis par l'utilisateur :

```
> somme := proc (n,k)
local S,i;
S :=0;
for i from 1 to k do
S :=S+i*n;
end do;
return S;
end proc;
> somme (2,3);
```

Détails de l'exécution

Quand on "entre" dans la boucle, S vaut 0 et i prend la valeur 1. Puis on calcule $S + i * n = 0 + 1 * 2 = 2$ que l'on affecte à S (qui vaut donc maintenant 2). Puis on passe au pas suivant : $i := i + 1 = 2$ et on calcule $S := S + i * n = 2 + 2 * 2 = 6$. Puis on passe au pas suivant : $i = 3$ et on calcule $S := S + i * n = 6 + 3 * 2 = 12$. Puisqu'on est arrivé à $i = 3 = k$, on arrête l'exécution de la boucle `for` et on reprend l'exécution du programme après le "end do ;", soit, dans notre exemple, on exécute "return S ;".

Détails de l'exécution

Chap. III :
Programmation

Laurent
Poinsot

Quand on "entre" dans la boucle, S vaut 0 et i prend la valeur 1. Puis on calcule $S + i * n = 0 + 1 * 2 = 2$ que l'on affecte à S (qui vaut donc maintenant 2). Puis on passe au pas suivant : $i := i + 1 = 2$ et on calcule $S := S + i * n = 2 + 2 * 2 = 6$. Puis on passe au pas suivant : $i = 3$ et on calcule $S := S + i * n = 6 + 3 * 2 = 12$. Puisqu'on est arrivé à $i = 3 = k$, on arrête l'exécution de la boucle `for` et on reprend l'exécution du programme après le "end do ;", soit, dans notre exemple, on exécute "return S ;".

Détails de l'exécution

Chap. III :
Programmation

Laurent
Poinsot

Quand on "entre" dans la boucle, S vaut 0 et i prend la valeur 1. Puis on calcule $S + i * n = 0 + 1 * 2 = 2$ que l'on affecte à S (qui vaut donc maintenant 2). Puis on passe au pas suivant : $i := i + 1 = 2$ et on calcule $S := S + i * n = 2 + 2 * 2 = 6$. Puis on passe au pas suivant : $i = 3$ et on calcule $S := S + i * n = 6 + 3 * 2 = 12$. Puisqu'on est arrivé à $i = 3 = k$, on arrête l'exécution de la boucle `for` et on reprend l'exécution du programme après le "end do ;", soit, dans notre exemple, on exécute "return S ;".

Détails de l'exécution

Chap. III :
Programmation

Laurent
Poinsot

Quand on "entre" dans la boucle, S vaut 0 et i prend la valeur 1. Puis on calcule $S + i * n = 0 + 1 * 2 = 2$ que l'on affecte à S (qui vaut donc maintenant 2). Puis on passe au pas suivant : $i := i + 1 = 2$ et on calcule $S := S + i * n = 2 + 2 * 2 = 6$. Puis on passe au pas suivant : $i = 3$ et on calcule $S := S + i * n = 6 + 3 * 2 = 12$. Puisqu'on est arrivé à $i = 3 = k$, on arrête l'exécution de la boucle `for` et on reprend l'exécution du programme après le "end do ;", soit, dans notre exemple, on exécute "return S ;".

Détails de l'exécution

Chap. III :
Programmation

Laurent
Poinsot

Quand on "entre" dans la boucle, S vaut 0 et i prend la valeur 1. Puis on calcule $S + i * n = 0 + 1 * 2 = 2$ que l'on affecte à S (qui vaut donc maintenant 2). Puis on passe au pas suivant : $i := i + 1 = 2$ et on calcule

$S := S + i * n = 2 + 2 * 2 = 6$. Puis on passe au pas suivant : $i = 3$ et on calcule $S := S + i * n = 6 + 3 * 2 = 12$. Puisqu'on est arrivé à $i = 3 = k$, on arrête l'exécution de la boucle `for` et on reprend l'exécution du programme après le "end do ;", soit, dans notre exemple, on exécute "return S ;".

Détails de l'exécution

Chap. III :
Programmation

Laurent
Poinsot

Quand on "entre" dans la boucle, S vaut 0 et i prend la valeur 1. Puis on calcule $S + i * n = 0 + 1 * 2 = 2$ que l'on affecte à S (qui vaut donc maintenant 2). Puis on passe au pas suivant : $i := i + 1 = 2$ et on calcule

$S := S + i * n = 2 + 2 * 2 = 6$. Puis on passe au pas suivant : $i = 3$ et on calcule $S := S + i * n = 6 + 3 * 2 = 12$. Puisqu'on est arrivé à $i = 3 = k$, on arrête l'exécution de la boucle `for` et on reprend l'exécution du programme après le "end do ;", soit, dans notre exemple, on exécute "return S ;".

Détails de l'exécution

Chap. III :
Programmation

Laurent
Poinsot

Quand on "entre" dans la boucle, S vaut 0 et i prend la valeur 1. Puis on calcule $S + i * n = 0 + 1 * 2 = 2$ que l'on affecte à S (qui vaut donc maintenant 2). Puis on passe au pas suivant : $i := i + 1 = 2$ et on calcule $S := S + i * n = 2 + 2 * 2 = 6$. Puis on passe au pas suivant : $i = 3$ et on calcule $S := S + i * n = 6 + 3 * 2 = 12$. Puisqu'on est arrivé à $i = 3 = k$, on arrête l'exécution de la boucle `for` et on reprend l'exécution du programme après le "end do ;", soit, dans notre exemple, on exécute "return S ;".

Détails de l'exécution

Chap. III :
Programmation

Laurent
Poinsot

Quand on "entre" dans la boucle, S vaut 0 et i prend la valeur 1. Puis on calcule $S + i * n = 0 + 1 * 2 = 2$ que l'on affecte à S (qui vaut donc maintenant 2). Puis on passe au pas suivant : $i := i + 1 = 2$ et on calcule $S := S + i * n = 2 + 2 * 2 = 6$. Puis on passe au pas suivant : $i = 3$ et on calcule $S := S + i * n = 6 + 3 * 2 = 12$. Puisqu'on est arrivé à $i = 3 = k$, on arrête l'exécution de la boucle `for` et on reprend l'exécution du programme après le "end do ;", soit, dans notre exemple, on exécute "return S ;".

Détails de l'exécution

Quand on "entre" dans la boucle, S vaut 0 et i prend la valeur 1. Puis on calcule $S + i * n = 0 + 1 * 2 = 2$ que l'on affecte à S (qui vaut donc maintenant 2). Puis on passe au pas suivant : $i := i + 1 = 2$ et on calcule $S := S + i * n = 2 + 2 * 2 = 6$. Puis on passe au pas suivant : $i = 3$ et on calcule $S := S + i * n = 6 + 3 * 2 = 12$. Puisqu'on est arrivé à $i = 3 = k$, on arrête l'exécution de la boucle `for` et on reprend l'exécution du programme après le "end do ;", soit, dans notre exemple, on exécute "return S ;".

Détails de l'exécution

Quand on "entre" dans la boucle, S vaut 0 et i prend la valeur 1. Puis on calcule $S + i * n = 0 + 1 * 2 = 2$ que l'on affecte à S (qui vaut donc maintenant 2). Puis on passe au pas suivant : $i := i + 1 = 2$ et on calcule $S := S + i * n = 2 + 2 * 2 = 6$. Puis on passe au pas suivant : $i = 3$ et on calcule $S := S + i * n = 6 + 3 * 2 = 12$. Puisqu'on est arrivé à $i = 3 = k$, on arrête l'exécution de la boucle `for` et on reprend l'exécution du programme après le "end do ;", soit, dans notre exemple, on exécute "return S ;".

Détails de l'exécution

Quand on "entre" dans la boucle, S vaut 0 et i prend la valeur 1. Puis on calcule $S + i * n = 0 + 1 * 2 = 2$ que l'on affecte à S (qui vaut donc maintenant 2). Puis on passe au pas suivant : $i := i + 1 = 2$ et on calcule $S := S + i * n = 2 + 2 * 2 = 6$. Puis on passe au pas suivant : $i = 3$ et on calcule $S := S + i * n = 6 + 3 * 2 = 12$. Puisqu'on est arrivé à $i = 3 = k$, on arrête l'exécution de la boucle `for` et on reprend l'exécution du programme après le "end `do ;`", soit, dans notre exemple, on exécute "return $S ;$ ".

Détails de l'exécution

Quand on "entre" dans la boucle, S vaut 0 et i prend la valeur 1. Puis on calcule $S + i * n = 0 + 1 * 2 = 2$ que l'on affecte à S (qui vaut donc maintenant 2). Puis on passe au pas suivant : $i := i + 1 = 2$ et on calcule $S := S + i * n = 2 + 2 * 2 = 6$. Puis on passe au pas suivant : $i = 3$ et on calcule $S := S + i * n = 6 + 3 * 2 = 12$. Puisqu'on est arrivé à $i = 3 = k$, on arrête l'exécution de la boucle `for` et on reprend l'exécution du programme après le "end `do ;`", soit, dans notre exemple, on exécute "return $S ;$ ".

Boucle `while`

La syntaxe de la boucle `while` est la suivante :

```
> while <condition> do  
<instruction1>;  
<instruction2>;  
...  
end do;
```

Boucle `while`

La syntaxe de la boucle `while` est la suivante :

```
> while <condition> do  
  <instruction1>;  
  <instruction2>;  
  ...  
end do;
```

Boucle `while`

Chap. III :
Programmation

Laurent
Poinsot

La syntaxe de la boucle `while` est la suivante :

```
> while <condition> do  
<instruction1>;  
<instruction2>;  
...  
end do;
```

Boucle `while`

Chap. III :
Programmation

Laurent
Poinsot

La syntaxe de la boucle `while` est la suivante :

```
> while <condition> do  
<instruction1>;  
<instruction2>;  
...  
end do;
```

Boucle `while`

Chap. III :
Programmation

Laurent
Poinsot

La syntaxe de la boucle `while` est la suivante :

```
> while <condition> do  
<instruction1>;  
<instruction2>;  
...  
end do;
```


Boucle `while`

La syntaxe de la boucle `while` est la suivante :

```
> while <condition> do  
<instruction1>;  
<instruction2>;  
...  
end do;
```

Explications

- 1 Les instructions `<instruction1>`, `<instruction2>`, ..., sont exécutées tant que la `<condition>` est vraie ;
- 2 Dès que la `<condition>` est fausse, l'exécution du programme se poursuit après l'instruction finale `"end do ;"`.

Explications

- 1** Les instructions `<instruction1>`, `<instruction2>`, ..., **sont exécutées tant que la `<condition>` est vraie ;**
- 2** Dès que la `<condition>` est fausse, l'exécution du programme se poursuit après l'instruction finale `"end do ;"`.

Explications

- 1** Les instructions `<instruction1>`, `<instruction2>`, ..., sont exécutées tant que la `<condition>` est vraie ;
- 2** Dès que la `<condition>` est fausse, l'exécution du programme se poursuit après l'instruction finale `"end do ;"`.

Explications

- 1** Les instructions `<instruction1>`, `<instruction2>`, ..., sont exécutées tant que la `<condition>` est vraie ;
- 2** Dès que la `<condition>` est fausse, l'exécution du programme se poursuit après l'instruction finale `"end do ;"`.

Exemple

Calcul de la somme des entiers strictement inférieurs à un

entier n : $\sum_{i=1}^{n-1} i$:

```
> somme := proc (n)
```

```
local S, i;
```

```
i := 1;
```

```
S := 0;
```

```
while i < n do
```

```
S := S+i;
```

```
i := i+1;
```

```
end do;
```

```
return S;
```

```
end proc;
```

Exemple

Calcul de la somme des entiers strictement inférieurs à un

entier n : $\sum_{i=1}^{n-1} i$:

```
> somme := proc (n)
local S, i;
i := 1;
S := 0;
while i < n do
S := S+i;
i := i+1;
end do;
return S;
end proc;
```

Exemple

Calcul de la somme des entiers strictement inférieurs à un

entier n : $\sum_{i=1}^{n-1} i$:

> somme := proc (n)

```
local S, i;
```

```
i := 1;
```

```
S := 0;
```

```
while i < n do
```

```
S := S+i;
```

```
i := i+1;
```

```
end do;
```

```
return S;
```

```
end proc;
```


Exemple

Calcul de la somme des entiers strictement inférieurs à un

entier n : $\sum_{i=1}^{n-1} i$:

```
> somme := proc (n)
```

```
local S, i;
```

```
i := 1;
```

```
S := 0;
```

```
while i < n do
```

```
S := S+i;
```

```
i := i+1;
```

```
end do;
```

```
return S;
```

```
end proc;
```

Exemple

Calcul de la somme des entiers strictement inférieurs à un

entier n : $\sum_{i=1}^{n-1} i$:

```
> somme := proc (n)
```

```
local S, i;
```

```
i := 1;
```

```
S := 0;
```

```
while i < n do
```

```
S := S+i;
```

```
i := i+1;
```

```
end do;
```

```
return S;
```

```
end proc;
```

Exemple

Calcul de la somme des entiers strictement inférieurs à un

entier n : $\sum_{i=1}^{n-1} i$:

```
> somme := proc (n)
```

```
local S, i;
```

```
i := 1;
```

```
S := 0;
```

```
while i < n do
```

```
S := S+i;
```

```
i := i+1;
```

```
end do;
```

```
return S;
```

```
end proc;
```

Exemple

Calcul de la somme des entiers strictement inférieurs à un

entier n : $\sum_{i=1}^{n-1} i$:

```
> somme := proc (n)
```

```
local S, i;
```

```
i := 1;
```

```
S := 0;
```

```
while i < n do
```

```
S := S+i;
```

```
i := i+1;
```

```
end do;
```

```
return S;
```

```
end proc;
```

Exemple

Calcul de la somme des entiers strictement inférieurs à un

entier n : $\sum_{i=1}^{n-1} i$:

```
> somme := proc (n)
```

```
local S, i;
```

```
i := 1;
```

```
S := 0;
```

```
while i < n do
```

```
S := S+i;
```

```
i := i+1;
```

```
end do;
```

```
return S;
```

```
end proc;
```

Exemple

Calcul de la somme des entiers strictement inférieurs à un

entier n : $\sum_{i=1}^{n-1} i$:

```
> somme := proc (n)
```

```
local S, i;
```

```
i := 1;
```

```
S := 0;
```

```
while i < n do
```

```
S := S+i;
```

```
i := i+1;
```

```
end do;
```

```
return S;
```

```
end proc;
```

Exemple

Calcul de la somme des entiers strictement inférieurs à un

entier n : $\sum_{i=1}^{n-1} i$:

```
> somme := proc (n)
```

```
local S, i;
```

```
i := 1;
```

```
S := 0;
```

```
while i < n do
```

```
S := S+i;
```

```
i := i+1;
```

```
end do;
```

```
return S;
```

```
end proc;
```

Exemple

Calcul de la somme des entiers strictement inférieurs à un

entier n : $\sum_{i=1}^{n-1} i$:

```
> somme := proc (n)
```

```
local S, i;
```

```
i := 1;
```

```
S := 0;
```

```
while i < n do
```

```
S := S+i;
```

```
i := i+1;
```

```
end do;
```

```
return S;
```

```
end proc;
```


Exemple

Calcul de la somme des entiers strictement inférieurs à un

entier n : $\sum_{i=1}^{n-1} i$:

```
> somme := proc (n)
```

```
local S, i;
```

```
i := 1;
```

```
S := 0;
```

```
while i < n do
```

```
S := S+i;
```

```
i := i+1;
```

```
end do;
```

```
return S;
```

```
end proc;
```

Cette même procédure écrite avec une boucle `for` en lieu et place de la boucle `while` prend la forme suivante :

```
> somme := proc (n)
local S,i;
S := 0;
for i from 1 to n-1 do
S := S+i;
end do;
return S;
end proc;
```

Cette même procédure écrite avec une boucle `for` en lieu et place de la boucle `while` prend la forme suivante :

```
> somme := proc (n)
local S,i;
S := 0;
for i from 1 to n-1 do
S := S+i;
end do;
return S;
end proc;
```

Cette même procédure écrite avec une boucle `for` en lieu et place de la boucle `while` prend la forme suivante :

```
> somme := proc (n)
  local S,i;
  S := 0;
  for i from 1 to n-1 do
  S := S+i;
  end do;
  return S;
end proc;
```

Cette même procédure écrite avec une boucle `for` en lieu et place de la boucle `while` prend la forme suivante :

```
> somme := proc (n)
local S,i;
S := 0;
for i from 1 to n-1 do
S := S+i;
end do;
return S;
end proc;
```

Cette même procédure écrite avec une boucle `for` en lieu et place de la boucle `while` prend la forme suivante :

```
> somme := proc (n)
local S,i;
S := 0;
for i from 1 to n-1 do
S := S+i;
end do;
return S;
end proc;
```

Cette même procédure écrite avec une boucle `for` en lieu et place de la boucle `while` prend la forme suivante :

```
> somme := proc (n)
local S,i;
S := 0;
for i from 1 to n-1 do
S := S+i;
end do;
return S;
end proc;
```

Cette même procédure écrite avec une boucle `for` en lieu et place de la boucle `while` prend la forme suivante :

```
> somme := proc (n)
local S,i;
S := 0;
for i from 1 to n-1 do
S := S+i;
end do;
return S;
end proc;
```


Cette même procédure écrite avec une boucle `for` en lieu et place de la boucle `while` prend la forme suivante :

```
> somme := proc (n)
local S,i;
S := 0;
for i from 1 to n-1 do
S := S+i;
end do;
return S;
end proc;
```

Cette même procédure écrite avec une boucle `for` en lieu et place de la boucle `while` prend la forme suivante :

```
> somme := proc (n)
local S,i;
S := 0;
for i from 1 to n-1 do
S := S+i;
end do;
return S;
end proc;
```

Cette même procédure écrite avec une boucle `for` en lieu et place de la boucle `while` prend la forme suivante :

```
> somme := proc (n)
local S,i;
S := 0;
for i from 1 to n-1 do
S := S+i;
end do;
return S;
end proc;
```

Comparaison `while` et `for`

Chap. III :
Programmation

Laurent
Poinsot

On remarque donc que dans un `while` il est nécessaire d'initialiser le compteur de la boucle (dans l'exemple : $i := 1$) et de gérer soi-même l'incrémement de ce compteur (dans l'exemple : $i := i + 1$) ce qui est fait automatiquement dans une boucle `for`.

Notons que si la `<condition>` de la boucle `while` est fausse **avant** d'entrer dans la boucle `while`, celle-ci n'est jamais exécutée.

Comparaison `while` et `for`

Chap. III :
Programmation

Laurent
Poinsot

On remarque donc que dans un `while` il est nécessaire d'initialiser le compteur de la boucle (dans l'exemple : $i := 1$) et de gérer soi-même l'incréméntation de ce compteur (dans l'exemple : $i := i + 1$) ce qui est fait automatiquement dans une boucle `for`.

Notons que si la `<condition>` de la boucle `while` est fausse **avant** d'entrer dans la boucle `while`, celle-ci n'est jamais exécutée.

Comparaison `while` et `for`

Chap. III :
Programmation

Laurent
Poinsot

On remarque donc que dans un `while` il est nécessaire d'initialiser le compteur de la boucle (dans l'exemple : $i := 1$) et de gérer soi-même l'incrémement de ce compteur (dans l'exemple : $i := i + 1$) ce qui est fait automatiquement dans une boucle `for`.

Notons que si la `<condition>` de la boucle `while` est fausse **avant** d'entrer dans la boucle `while`, celle-ci n'est jamais exécutée.

Comparaison `while` et `for`

Chap. III :
Programmation

Laurent
Poinsot

On remarque donc que dans un `while` il est nécessaire d'initialiser le compteur de la boucle (dans l'exemple : $i := 1$) et de gérer soi-même l'incréméntation de ce compteur (dans l'exemple : $i := i + 1$) ce qui est fait automatiquement dans une boucle `for`.

Notons que si la `<condition>` de la boucle `while` est fausse **avant** d'entrer dans la boucle `while`, celle-ci n'est jamais exécutée.

Opérateurs booléens (ou logiques)

Une expression booléenne (ou logique) est une expression qui a pour valeur un booléen, soit `true`, soit `false`. Par exemple la `<condition>` dans les instructions `if` et `while` est une expression booléenne. On peut obtenir de nouvelles expressions booléennes à partir d'expressions booléennes données en les "connectant" à l'aide d'opérateurs logiques que sont le "et" (`and`), le "ou" (`or`) et le "non" (`not`). C'est très utile justement pour écrire des conditions complexes dans les instructions `if` ou `while`.

Opérateurs booléens (ou logiques)

Chap. III :
Programmation

Laurent
Poinsot

Une expression booléenne (ou logique) est une expression qui a pour valeur un booléen, soit `true`, soit `false`. Par exemple la `<condition>` dans les instructions `if` et `while` est une expression booléenne. On peut obtenir de nouvelles expressions booléennes à partir d'expressions booléennes données en les "connectant" à l'aide d'opérateurs logiques que sont le "et" (`and`), le "ou" (`or`) et le "non" (`not`). C'est très utile justement pour écrire des conditions complexes dans les instructions `if` ou `while`.

Opérateurs booléens (ou logiques)

Chap. III :
Programmation

Laurent
Poinsot

Une expression booléenne (ou logique) est une expression qui a pour valeur un booléen, soit `true`, soit `false`. Par exemple la `<condition>` dans les instructions `if` et `while` est une expression booléenne. On peut obtenir de nouvelles expressions booléennes à partir d'expressions booléennes données en les "connectant" à l'aide d'opérateurs logiques que sont le "et" (`and`), le "ou" (`or`) et le "non" (`not`). C'est très utile justement pour écrire des conditions complexes dans les instructions `if` ou `while`.

Opérateurs booléens (ou logiques)

Une expression booléenne (ou logique) est une expression qui a pour valeur un booléen, soit `true`, soit `false`. Par exemple la `<condition>` dans les instructions `if` et `while` est une expression booléenne. On peut obtenir de nouvelles expressions booléennes à partir d'expressions booléennes données en les "connectant" à l'aide d'opérateurs logiques que sont le "et" (`and`), le "ou" (`or`) et le "non" (`not`). C'est très utile justement pour écrire des conditions complexes dans les instructions `if` ou `while`.

Opérateurs booléens (ou logiques)

Une expression booléenne (ou logique) est une expression qui a pour valeur un booléen, soit `true`, soit `false`. Par exemple la `<condition>` dans les instructions `if` et `while` est une expression booléenne. On peut obtenir de nouvelles expressions booléennes à partir d'expressions booléennes données en les "connectant" à l'aide d'opérateurs logiques que sont le "et" (`and`), le "ou" (`or`) et le "non" (`not`). C'est très utile justement pour écrire des conditions complexes dans les instructions `if` ou `while`.

Opérateurs booléens (ou logiques)

Chap. III :
Programmation

Laurent
Poinso

Une expression booléenne (ou logique) est une expression qui a pour valeur un booléen, soit `true`, soit `false`. Par exemple la `<condition>` dans les instructions `if` et `while` est une expression booléenne. On peut obtenir de nouvelles expressions booléennes à partir d'expressions booléennes données en les "connectant" à l'aide d'opérateurs logiques que sont le "et" (`and`), le "ou" (`or`) et le "non" (`not`). C'est très utile justement pour écrire des conditions complexes dans les instructions `if` ou `while`.

Opérateurs booléens (ou logiques)

Chap. III :
Programmation

Laurent
Poinsot

Une expression booléenne (ou logique) est une expression qui a pour valeur un booléen, soit `true`, soit `false`. Par exemple la `<condition>` dans les instructions `if` et `while` est une expression booléenne. On peut obtenir de nouvelles expressions booléennes à partir d'expressions booléennes données en les "connectant" à l'aide d'opérateurs logiques que sont le "et" (`and`), le "ou" (`or`) et le "non" (`not`). C'est très utile justement pour écrire des conditions complexes dans les instructions `if` ou `while`.

La commande `evalb` (signifiant en anglais "EVALuate to Boolean") détermine si une expression booléenne est vraie ou fausse :

```
> evalb ((1 < 4) and (3 <> 3)) ;
```

false

```
> evalb ((4 < 1) or (5 = 5)) ;
```

true

```
> evalb (not (2 <= 3)) ;
```

false

La commande `evalb` (signifiant en anglais "EVALuate to Boolean") détermine si une expression booléenne est vraie ou fausse :

```
> evalb ((1 < 4) and (3 <> 3)) ;
```

false

```
> evalb ((4 < 1) or (5 = 5)) ;
```

true

```
> evalb (not (2 <= 3)) ;
```

false

La commande `evalb` (signifiant en anglais "EVALuate to Boolean") détermine si une expression booléenne est vraie ou fausse :

```
> evalb ((1 < 4) and (3 <> 3)) ;
```

false

```
> evalb ((4 < 1) or (5 = 5)) ;
```

true

```
> evalb (not (2 <= 3)) ;
```

false

La commande `evalb` (signifiant en anglais "EVALuate to Boolean") détermine si une expression booléenne est vraie ou fausse :

```
> evalb ((1 < 4) and (3 <> 3)) ;
```

false

```
> evalb ((4 < 1) or (5 = 5)) ;
```

true

```
> evalb (not (2 <= 3)) ;
```

false

La commande `evalb` (signifiant en anglais "EVALuate to Boolean") détermine si une expression booléenne est vraie ou fausse :

```
> evalb ((1 < 4) and (3 <> 3)) ;
```

false

```
> evalb ((4 < 1) or (5 = 5)) ;
```

true

```
> evalb (not (2 <= 3)) ;
```

false

La commande `evalb` (signifiant en anglais "EVALuate to Boolean") détermine si une expression booléenne est vraie ou fausse :

```
> evalb ((1 < 4) and (3 <> 3)) ;
```

false

```
> evalb ((4 < 1) or (5 = 5)) ;
```

true

```
> evalb (not (2 <= 3)) ;
```

false

La commande `evalb` (signifiant en anglais "EVALuate to Boolean") détermine si une expression booléenne est vraie ou fausse :

```
> evalb ((1 < 4) and (3 <> 3)) ;
```

false

```
> evalb ((4 < 1) or (5 = 5)) ;
```

true

```
> evalb (not (2 <= 3)) ;
```

false

La commande `evalb` (signifiant en anglais "EVALuate to Boolean") détermine si une expression booléenne est vraie ou fausse :

```
> evalb ((1 < 4) and (3 <> 3)) ;
```

false

```
> evalb ((4 < 1) or (5 = 5)) ;
```

true

```
> evalb (not (2 <= 3)) ;
```

false