la boucle itérative

La syntaxe des boucles itératives est la suivante :

```
for nom de variable from valeur initiale by pas to valeur finale do  
    bloc d'instructions  
od;
```

Pour une variable *nom de variable* variant de la *valeur initiale* à la *valeur finale* avec le pas *pas* on exécute le bloc d'instructions.

exemple :

```
> for i from 3 to 10 do print(i) od;
```

va afficher les entiers de 3 à 10.

Si on veut calculer la somme des entiers compris entre 3 et 10 on modifie la ligne de maple précédente de la façon suivante :

```
> s := 0;  
for i from 3 to 10 do  
    s := s + i ;  
od ;  
print(s);
```

Nous en savons suffisamment pour réécrire à notre goût la fonction **sum** de Maple. Cette fonction doit prendre comme argument une fonction f , la valeur initiale a , la valeur finale b et doit calculer $\sum_{k=a}^b f(i)$.

Ce qui nous donne par exemple :

```
> somme := proc(f, a, b)  
    local s, i;  
    s := 0;  
    for i from a to b do  
        s := s + f(i) ;  
    od ;  
    RETURN(s);  
end;
```

le branchement conditionnel

La syntaxe des branchements conditionnels est la suivante :

```
if expression conditionnelle 1 then bloc d'instructions 1
elif expression conditionnelle 2 then bloc d'instructions 2
...
elif expression conditionnelle n then bloc d'instructions n
else bloc d'instructions n+1
fi
```

Si la première expression conditionnelle est vraie, on exécute le premier bloc d'instructions et on sort après le **fi** sinon on teste l'expression conditionnelle 2. Si elle est vraie, on exécute le bloc d'instruction 2 et on sort après le **fi**. Si toutes les expressions conditionnelles sont fausses alors on exécute le bloc d'instruction (n+1) après le **else** et on sort.

exemple : on peut ainsi facilement définir la fonction $f : \begin{pmatrix} x < 0 & \mapsto & x \\ x \in [0, 2] & \mapsto & 2x \\ x > 2 & \mapsto & 4x \end{pmatrix}$ de la façon suivante :

```
> f := proc(x)
  if (x < 0) then RETURN(x)
  elif (x <= 2) then RETURN(2 * x)
  else RETURN(4 * x)
fi;
end;
```

On peut améliorer la fonction **somme** précédemment écrite en considérant que : $\sum_{k=a}^b f(i) = \sum_{k=b}^a f(i)$ si $b < a$ (remarque : la première version de la fonction **somme** retourne la valeur 0 lorsque $b < a$). Ce qui nous donne :

```
> somme := proc(f, a, b)
  local s, i, m, n;
  if a <= b then m := a; n := b;
  else m := b; n := a;
  s := 0;
  for i from m to n do
    s := s + f(i);
  od :
  RETURN(s);
end;
```

la boucle conditionnelle

La syntaxe des boucles conditionnelles est la suivante :

```
for nom de variable from valeur initial while expression logique do  
    bloc d'instructions  
od;
```

Pour la variable *nom de variable* à partir de la *valeur initiale* tant que l'*expression logique* est vraie, on exécute le *bloc d'instructions*.

Ainsi, toute boucle itérative peut s'écrire sous la forme d'une boucle conditionnelle.

```
for i from 3 to 10 do  
    s := s + i ;  
od ;
```

peut s'écrire :

```
for i from 3 while i <= 10 do  
    s := s + i ;  
od ;
```

Imaginons un exemple où l'on ne peut pas se passer d'une boucle conditionnelle. On veut écrire une fonction qui étant donné un entier naturel positif p retourne le plus grand des entiers n tel que $\sum_{k=0}^n k \leq p$.

La suite $\left(\sum_{k=0}^n k\right)_{n \in \mathbb{N}}$ étant croissante, si n_0 est la valeur cherchée, on aura $n_0 + 1$ qui sera la première de n pour laquelle $\sum_{k=0}^n k > p$ ce qui nous suggère le programme suivant :

```
> somme := proc(p)  
    local s, n;  
    s := 0;  
    for n from 0 while s <= p do  
        s := s + n : # s contient la somme  $\sum_{k=0}^n k$   
    od ;  
    RETURN(n - 1);  
end;
```