Introduction au langage `Pascal`

Yannick Chevalier

Contents

Yannick Chevalier

1 Commençons par un petit exemple...

```
PROGRAM circonference;
VAR rayon, circonference :  REAL;

BEGIN
  readln(rayon);
  circonference:=rayon*2*3.1416 ;
  writeln(rayon,circonference);
END.
```

Ce programme demande a un utilisateur de taper un nombre, et il rend ce nombre et la circonférence d'un cercle dont ce nombre est le rayon. Ce n'est pas trop compliqué. On peut tout de suite identifier les parties les plus importantes.

Il y a d'abord le squelette du programme :

```
PROGRAM ...  ;
(* Calcul de la circonference d'un cercle *)
VAR ....  ;

BEGIN
...
END.
```

1. La partie `PROGRAM...;` sert à donner un nom au programme ;
2. la partie `(*...*)` sert à donner des commentaires. Ils sont essentiels pour comprendre ce que fait un programme. Ils ne sont pas facultatifs ;
3. la partie `VAR...;` sert à déclarer les variables dont on aura besoin plus tard ;
4. la partie `BEGIN ... END.` contient ce que le programme fait. C'est la partie qui contient l'algorithme. Elle se termine par un point ".". C'est la partie *algorithme* du programme.

On va maintenant voir plus en détail ces différentes parties. Je ne reviens pas sur le nom du programme.

2 La partie des déclarations

2.1 Les différents types de base

```
VAR rayon, circonference :  REAL;
```

On déclare que dans l'algorithme, on utilise deux variables, qu'on appelle `rayon` et `diametre`. En plus, on déclare que ces deux variables sont de types `REAL`. Cela signifie qu'on pourra les utiliser un peu comme des nombres réels. Mais il y a d'autres types qui sont définis :

1. le type `integer`. Il s'agit du type des nombres qui sont des entiers relatifs ;
2. le type `boolean`. Il s'agit du type des booléens. Ces variables ne peuvent prendre que 2 valeurs, `TRUE` (vrai) ou `FALSE` (faux) ;
3. le type `char`. Il s'agit du type qui contient les caractères. Une variable de ce type peut avoir la valeur `'A'`, `'b'`, `'7'`,... Le `7` d'une variable de type `char` est un caractère. Il n'est pas question de faire des opérations dessus !

2.2 Les opérations qui préservent le type

On utilise le *type* d'une expression pour savoir qu'elles opérations on peut appliquer.

Sur les types de nombres, on a toujours `+`, `-`, et `*`. Question :
que signifie 3/5 ? (division euclidienne ou réelle ?)
On voit que la division ne voudra pas dire la même chose pour les `integer` et pour les `real`.

Pour les `integer`, on utilise la division euclidienne. Autrement dit, on sépare en reste et en partie entiere. Par exemple, $17 = 3 * 5 + 2$. Sur les entiers, on définit les 2 opérations `div` et `mod`, qui auront pour valeur :

1. `17 mod 3` a pour valeur `2` ;
2. `17 div 3` a pour valeur `5`.

Pour les `real`, on utilise le symbole standard de division, `/`. Il n'y a pas de surprises. Il est possible de faire des opérations mélant des `real` et des `integer`. Dans ce cas, les deux opérandes sont considérés de type `real`.

Sur les caractères, il n'y a pas d'opérations possibles. Mais ils sont codés dans le code `ASCII`. Il faut faire attention, car tous les caractères n'y sont pas, il n'y en a que 256. L'intérêt est qu'il est possible de passer d'un entier entre `0` et `255` à un caractère, en utilisant la fonction `CHR`. Par exemple, on a :

- `CHR(67)` prend la valeur `'C'` ;
- `CHR(51)` prend la valeur `'3'`.

Sur les booléens, on a les opérations classiques de logique, `AND`, `OR`, `NOT`, ainsi que l'opération `XOR`.

Exercice :
Donner une expression utilisant x, y, `not` , `and` et `or` qui rend le même résultat que l'opération x `xor` y.

$x \backslash y$	false	true
false	false	true
true	true	true

x OR y

$x \backslash y$	false	true
false	false	true
true	false	true

x AND y

x	not x
false	true
true	false

not x

$x \backslash y$	false	true
false	false	true
true	true	false

x XOR y

Figure 1: les opérations booléennes

2.3 Les opérations qui rendent un type `boolean`

Il y a içi toutes les opérations de comparaison <, >, <=, >= et <> *(différent)*. On compare entre elles deux expressions d'un même type, si ce type peut être ordonné. Il est aussi possible de comparer des expressions de type `real` avec des expressions de type `integer`. Pour les caractères, on utilise l'ordre du code ASCII. Par exemple :

- $CHR(60) < CHR(49)$ rend `false` ;
- 3.5 >= 3 rend `true` ;
- 3.0 <> 3 rend `false`.

3 La partie algorithme

Il s'agit de la partie où est défini l'algorithme. On va l'étudier un peu plus en détail. Cette partie est composée d'*instructions* qui sont séparées par des point-virgules.

```
readln(rayon);
circonference:=rayon*2*3.1416 ;
writeln(rayon,circonference);
```

Il est obligatoire de mettre le point-virgule entre chaque instruction. Le dernier point-virgule, lui, n'est pas obligatoire. Le mettre revient à ajouter une instruction vide. Il faut noter qu'on peut mettre autant d'intructions vides à la suite qu'on le désire.

3.1 Interface avec l'utilisateur

Les deux commandes :

```
readln(rayon);
...
writeln(rayon,circonference);
```

servent à interagir avec l'utilisateur du programme.

`readln` est utilisée pour ligne une donnée entrée par l'utilisateur, et qu'il valide en appuyant sur la touche `return`. L'utilisateur *doit* donner une valeur qui correspond au type attendu. Si ce n'est pas le cas, le programme s'arrête. Quelques exemples de valeurs possibles :

- pour un `integer` : $3, 5, 17, \ldots$ mais pas 3.5 ou 'A' ;
- pour un `real` : $4, 3.5, 45.74e-1, .07, \ldots$ mais toujours pas 'A' ;
- pour un `boolean` : il n'est pas possible de les demander directement à l'utilisateur. Par contre, il est toujours possible de les écrire ;
- pour les `char` : `3`, `A`, `'`, `è`,..., mais pas 'AA'.

3.2 Les affectations

Il reste une dernière instruction à voir, l'*affectation* :

```
circonference:=rayon*2*3.1416 ;
```

Cette instruction est définie par `:=`. À droite de ce signe, il y a une expression :

```
...:=rayon*2*3.1416
```

qui rend une valeur de type `real`. À gauche de ce signe

```
circonference:=...
```

il y a le nom d'une variable de même type, `real`. Cette instruction stocke la valeur à droite dans la variable à gauche. Lorsque, à droite, on utilise la variable `rayon`, on va en fait utiliser la dernière valeur qui a été stockée dans cette variable.

Il faut noter deux choses :

- il y a un risque d'erreur si on utilise une variable dans laquelle rien n'a été stocké. Il faut toujours vérifier qu'il y a une valeur de stockée dans la variable avant de l'utiliser ;
- les types à gauche et à droite de l'affectation doivent toujours être les mêmes, sauf si à droite c'est une expression de type `integer`, et à gauche une variable de type `real`.

4 Contrôle de flux

4.1 Exécution conditionnelle

On veut faire un programme qui donne, en sortie, la valeur absolue d'un nombre que donne l'utilisateur. Ce nombre est a priori de type `real`. Ce qui donne le squelette suivant pour ce programme :

```
Program valeur_absolue ;
(*
  Ce programme calcule la valeur absolue d'un nombre donne par
  un utilisateur. Les variables sont assez explicites
  *)
nombre,valeurabsolue : real;

begin
  writeln('De quel nombre voulez-vous la valeur absolue ?');
  readln(nombre);
  (* calcul de la valeur absolue *)
  writeln('La valeur absolue de ',nombre,' est ',valeurabsolue);
end.
```

Il reste à savoir comment calculer la valeur absolue. On sait que :

- si `nombre` $>= 0$, la valeur absolue de `nombre` est `nombre` ;
- si `nombre` < 0, la valeur absolue de `nombre` est l'opposé de `nombre`.

Il y a une instruction, en `Pascal`, qui agit comme le "si". C'est l'instruction :

`if` *expr_bool* `then` *inst*

expr_bool est une expression booléenne qu'on commence par calculer. Si elle vaut `true`, alors on exécute l'instruction *inst*. Sinon, on ne fait rien (c'est comme si on avait utilisé une instruction vide). Le programme devient :

```
Program valeur_absolue ;
(*
  Ce programme calcule la valeur absolue d'un nombre donne par
  un utilisateur. Les variables sont assez explicites.
  On utilise deux conditionnelles.
  *)
Var nombre,valeurabsolue : real;

begin
  writeln('De quel nombre voulez-vous la valeur absolue ?');
  readln(nombre);
  if (nombre >= 0)  then valeurabsolue:=nombre ;
  if (nombre < 0)  then valeurabsolue:=-nombre ;
  writeln('La valeur absolue de ',nombre,' est ',valeurabsolue);
end.
```

Pour calculer la valeur absolue, on peut aussi utiliser la construction `IF...THEN...ELSE...`. Cette construction prend une expression de type `boolean`, et deux instructions. le programme exécute la première si l'expression de type `boolean` vaut `true`, et la deuxième si l'expression vaut `false` :

`if` *expr_bool* `then` *inst1* `else` *inst2*

Le programme devient alors :

```
Program valeur_absolue ;
(*
  Ce programme calcule la valeur absolue d'un nombre donne par
  un utilisateur. Les variables sont assez explicites
  On n'utilise plus qu'une conditionnelle
  *)
Var nombre,valeurabsolue : real;

begin
  writeln('De quel nombre voulez-vous la valeur absolue ?');
  readln(nombre);
  if (nombre >= 0) then
    valeurabsolue:=nombre
  else
    valeurabsolue:=-nombre ;
  writeln('La valeur absolue de ',nombre,' est ',valeurabsolue);
end.
```

On ne peut pas ajouter de point-virgules après l'instruction :

```
valeurabsolue:=nombre
```

car on n'a le droit de mettre qu'une seule instruction après le `then` et après le `else`. Le point-virgule qui suit l'instruction

```
valeurabsolue:=-nombre
```

signifie la fin de l'instruction `if...then...else...`.

4.2 Les blocs d'instructions

Cette limitation à une seule instruction est très vite gênante. Heureusement, on peut regrouper des instructions qui vont ensemble en une seule instruction, qu'on appelle un *bloc d'instructions*. On regroupe les instructions avec la construction :

`begin` *inst1* ; ... ; *instn* `end`

Bien sûr, la suite d'instruction peut être vide, ou ne contenir que des instructions vides, ou encore être une suite normale d'instructions. En reprenant le programme de calcul de valeur absolue, on a maintenant :

```
Program valeur_absolue ;
(*
  Ce programme calcule la valeur absolue d'un nombre donne par
  un utilisateur. Les variables sont assez explicites
  On illustre ici l'utilisation de blocs d'instructions.
  *)
Var nombre,valeurabsolue : real;

begin
  writeln('De quel nombre voulez-vous la valeur absolue ?');
  readln(nombre);
  if (nombre ¿= 0) then
    begin
      valeurabsolue:=nombre;
      writeln('Le nombre entré est positif,');
      writeln('donc il est égal à sa valeur absolue')
    end
  else
    begin
      valeurabsolue:=-nombre ;
      writeln('Le nombre entré est négatif,');
      writeln('donc il est égal à l'opposé de sa valeur absolue')
    end;
  writeln('La valeur absolue de ',nombre,' est ',valeurabsolue);
end.
```

4.3 Les boucles sur des types énumérés

On parle de *type énuméré* pour désigner un type dans lequel on peut donner tous les éléments les uns à la suite des autres. Pour l'instant, on en a vu deux, le type `integer` et le type `char`, mais on en verra d'autres dans la suite du cours.

Exercice :
Afficher toutes les lettres de A à E (une lettre par ligne)

Pour résoudre cet exercice, une première méthode est d'écrire le programme suivant :

```
Program lettresAE;
(*
  On affiche les lettres de A à E, une par ligne...
  On n'a pas besoin de déclarer de variables.
  *)

begin
  writeln('A');
  writeln('B');
  writeln('C');
  writeln('D');
  writeln('E')
end.
```

On peut faire la même chose en utilisant l'instruction :

`for` < *variable* > `:=` < *minimum* > `to` < *maximum* > `do` < *instruction* >

l'instruction sera exécutée pour toutes les valeurs de la variable entre le minimum et le maximum dans l'ordre croissant. On appelle la variable *indice* de la boucle. Bien sûr, cette instruction peut être un bloc... Utilisons cette nouvelle construction :

```
Program lettresAEboucle;
(*
  On affiche les lettres de A à E, en utilisant une
  boucle.
  *)
Var lettre : char;

begin
  for lettre := 'A' to 'E' do
    writeln(lettre)
end.
```

Exercice :

Écrire tous les mots de exactement 3 lettres écris avec les lettres de 'A' à 'E'.

Cette fois-çi, ce serait un peu long d'écrire les lignes à la main (il y en a 125).

```
Program motstroislettres;
(*
  On affiche les mots de trois lettres en utilisant trois
  boucles les unes dans les autres. Il y a une boucle pour la première
  lettre, une pour la deuxième, une pour la troisième.
  *)
Var lettre1, lettre2, lettre3 : char;

begin
  for lettre1 := 'A' to 'E' do
    for lettre2 := 'A' to 'E' do
      for lettre3 := 'A' to 'E' do
        writeln(lettre1,lettre2,lettre3)
end.
```

Les boucles "`for` " sont utiles mais il faut prendre quelques précautions quand on les utilise :

1. il ne faut jamais changer la valeur de l'indice d'une boucle dans l'instruction. Si il le faut, on utilisera une autre variable qui prend la valeur de l'indice ;
2. La valeur de l'indice est limitée à l'instruction de la boucle. C'est une variable muette.

Ne pas suivre la première règle peut donner un programme valide (qui marche bien), mais il y a des risques d'erreurs et il est possible de rendre le programme incompréhensible. Par exemple, que fait la boucle `for` suivante :

```
Program mauvaiseboucle;
(*
  Exemple de programme où on change le valeur de
  l'indice à l'intérieur d'une boucle for
  *)
Var lettre1, lettre2, lettre3 : char;

begin
  for lettre1 := 'A' to 'E' do
    begin
      writeln(lettre1);          lettre1:= 'A'      end
end.
```

Ces boucles sont à utiliser dès qu'il faut faire un même traitement plusieurs fois sur un domaine fini, et connu à l'avance. Mais il peut arriver qu'on ne sache pas, au départ, combien de fois il faudra faire un traitement. Dans ce cas, il faut utiliser une autre construction.

4.4 Les boucles `while` et `repeat`

Un cas classique est le calcul d'un point où une fonction s'annule par la méthode de Newton. On s'arrête dès qu'on trouve une valeur dont l'image est suffisament proche de 0. Mais on ne sait pas, au départ, combien de d'itération il faudra faire.

Calcul du zéro d'une fonction par la méthode de Newton

a) Choisir une valeur x_0 dans l'intervalle $[A, B]$
b) Tant que $|f(x_n)| > \epsilon$, calculer :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

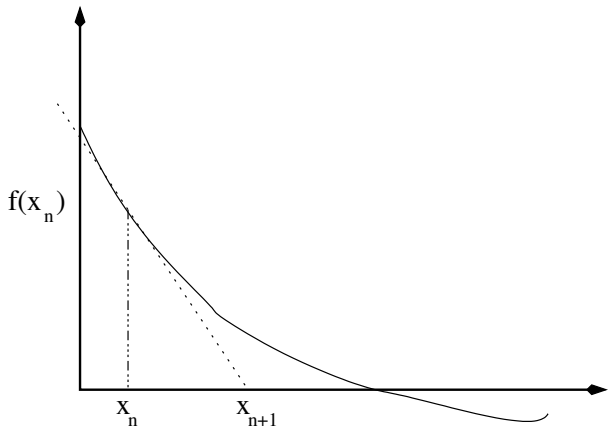

Figure 2: Calcul de x_{n+1} en fonction de x_n : on calcule x_{n+1} comme le point d'intersection de la tangente en $(x_n, f(x_n))$ à la courbe $y = f(x)$ avec la droite d'équation $y = 0$

Chaque fois qu'on doit faire un calcul tant qu'une condition est vérifiée, on utilise une boucle `while` :

`while` < *condition* > `do` < *instruction* >

Lorsqu'on utilise cette boucle, le programme fera, dans cet ordre :

1. (a) tester si la condition est vraie ;
   (b) si elle est vraie, faire l'instruction, sinon, passer à l'instruction suivante.
2. (a) tester si la condition est vraie ;
   (b) si elle est vraie, faire l'instruction, sinon, passer à l'instruction suivante.
3. ...

Bien sûr, il y a des cas où on ne sortira jamais de la boucle (penser à une fonction qui ne s'annule pas). Il n'existe pas de moyen général permettant d'assurer qu'on sortira de la boucle. Il faut donc être sûr qu'on s'arrêtera un jour avant de commencer une telle boucle.

Il existe une autre instruction qui fait presque la même chose :

$$\texttt{repeat} < instruction1; \ldots; instructionN > \texttt{until} < condition >$$

Il y a quelques différence entre cette construction et la construction `while` :

- on remarque qu'il peut y avoir, dans ce cas, plusieurs instructions. Il n'est donc pas utile d'utiliser un bloc `begin`...`end`;
- On commence par faire les instructions avant de tester la condition. Et on recommence jusqu'à ce que cette condition soit vraie (dans la boucle `while` , on recommence tant que la condition est vraie).

5 D'autres types

Un *type* défini un ensemble d'objets, les éléments de ce type. Pour l'instant, on a vu ce qu'on pourrait appeler les types mathématiques. Ce sont ceux qui sont définis en `Pascal`. Mais on a aussi vu le type `char`, qui s'applique à des objets n'ayant pas de fortes propriétés.

On va voir qu'on peut ajouter des types à ceux définis par défaut dans `Pascal`. Cela permet plus de précision dans le nommage des variables.

5.1 Types énumérés non standards

L'exemple classique est celui des jours de la semaine. Pour les utiliser dans un programme `Pascal`, on peut faire une table :

- lundi = 0
- mardi = 1
- ...

et de déclarer les variables qui contiendront des jours de la semaine comme des entiers. Mais `Pascal` nous offre beaucoup mieux, puisqu'il permet de définir le type JOUR, un peu comme il existe déjà le type entier :

`TYPE` jour = (lundi,mardi, mercredi,jeudi,vendredi,samedi,dimanche)

C'est ce qu'on appelle un type énuméré : on peut donner la liste des valeurs possibles les unes après les autres. Les autres types énumérés déjà définis sont les `boolean`, les `char`, mais *aussi les* `integer`.

Tous ces types se comportent comme des entiers, et la déclaration qu'on a faite plus haut est traduite sous la forme $lundi = 0$, $mardi = 1$ et ainsi de suite. L'avantage est qu'il existe des fonctions définies sur tous les types énumérés :

- la fonction `ORD`, qui donne le numéro dans la liste de déclaration (`ord(mardi)=1`, on commence à 0) ;
- les fonctions `SUCC` et `PRED`, qui donnent le successeur et le prédécesseur dans la liste : `succ(mardi)` = `mercredi`, `pred(samedi)`=`vendredi`. Attention, `pred(lundi)` et `succ(dimanche)` ne sont pas définis. On ne sait pas ce qu'ils valent ;
- les opérateurs de comparaison ;
- les boucles `for` ;

Il faut faire attention, les fonctions `readln` et `writeln` ne marchent pas avec ces types.

5.2 Les types intervalles

À partir de l'exemple du type `jour`, on maintenant besoin d'avoir tous les jours de travail. On peut définir *(après avoir défini le type* `jour`*)* le type `jourtravail`. Ce sont les jours entre lundi et vendredi (dans l'intervalle $[lundi, vendredi]$). On note les intervalles avec "`..`". On peut alors déclarer les jours de travail :

`type` jourtravail=lundi`..`vendredi

On peut aussi définir les minuscules :

`type` minuscule ='a'..'z'

ou toute autre sorte de type...

5.3 L'instruction CASE

Souvent, avec les types énumérés, on veut faire un traitement qui dépend de la valeur. Pour cela, on peut bien sûr utiliser des instruction `if` , mais cela à tendance à rendre le programme plus lourd que nécessaire. Lorsqu'on veut fair eun traitement par cas, il y a une instruction beaucoup plus pratique, l'instruction `case` :

```
CASE <expression> OF
  listedecas1 :  instruction1 ;
  listedecas2 :  instruction2 ;
  ...   listedecasN : instructionN
```

On a une liste de `un cas : une instruction` séparée par des `;`. Les *listedecas* peuvent être de plusieurs types :

1. une valeur, par exemple `mardi` ;
2. une liste de valeurs, séparées par des virgules. Pour reprendre l'exemple précédent, on peut choisir `samedi,lundi` ;
3. un intervalle : `jeudi..samedi`.

Exercice :
Faire un programme qui prend un caractère un entrée, et qui dit si ce caractère est une lettre minuscule, majuscule, un chiffre ou autre chose.

---

6 Les tableaux

Il s'agit de la construction la plus importante dans un langage de programmation. Sans les tableaux, on ne peut pas faire grand chose. Un *tableau* sert à rassembler des valeurs du même type, et à les organiser pour pouvoir accèder à chaque valeur. En mathématiques, c'est la même chose que des vecteurs ou des matrices. Un tableau, c'est une suite de cases contenant des valeurs.

Pour définir un tableau, il faut deux choses :

1. son domaine, qui indique comment on peut accèder aux cases ;
2. le type des valeurs qu'il contient.

Par exemple, on peut définir un tableau qui contient le nombre d'heures de cours pour chaque jour de la semaine :

```
Var horaires = Array [lundi..vendredi] of integer;
```

Mais on aurait aussi pu écrire (toujours avec les définitions précédentes) :

```
Var horaires = Array [jourstravail] of integer;
```

On utilise ensuite chaque case du tableau comme une variable :

- `horaires[mercredi]:=5`
- `cours:=horaires[lundi]`
- etc.

6.1 Les tableaux de tableaux

Le type de la valeur contenue dans le tableau peut aussi être un tableau. Par exemple, si on veut définir une matrice 3x3 :

```
Var matrice : Array [1..3] of Array [1..3] of real
```

Ce qui peut aussi s'écrire :

```
Var matrice : Array [1..3,1..3] of real
```

Dans les deux cas, on accède à l'élément (1,2) avec `matrice[1,2]`. On peut aussi accèder à la deuxième ligne, avec `matrice[2]`, mais on ne peut pas accèder à la deuxième colonne...

6.2 Les types de tableaux

On peut aussi définir des types de tableaux. Par exemple, avec les matrices, c'est mieux de définir des types matrices et vecteurs, et ensuite des variables ayant ces types :

```
Type vecteur : Array [1..3] of real;
  matrice : Array [1..3,1..3] of real;
Var u,v : vecteur ;
  M : matrice ;
...
```

6.3 Les chaînes de caractères

Une chaîne de caractères est un tableau de caractères. On peut lire et écrire des chaînes de caractères avec les fonctions `readln` et `writeln`. Il faut penser à prendre un tableau suffisament grand pour qu'il puisse contenir tout ce que va taper l'utilisateur.

6.4 Les constantes

Dans l'exemple précédent, on prenait des matrices de type 3x3, avec des vecteurs de dimension 3. Si on veut changer la dimension de 3 en 4, cela peut être pénible de rechercher partout dans le programme les 3 qui correspondent à la dimension de l'espace, et de remplacer tous ces 3 par des 4.

Une méthode rapide est de définir une constante. Il s'agit d'un nom qui sera remplacé dans tout le programme par la valeur qu'on lui donne. ce n'est pas une variable, il n'est donc pas possible de changer la valeur de cette constante. Si on rassemble toutes les déclarations vues jusqu'içi (d'autres arrivent), la partie déclaration d'un programme `Pascal`ressemble à :

```
Const dimension = 3 ;
  pi = 3.1415;
  message = 'Bienvenue.';
Type vecteur = Array [1..dimension]  of real;
  matrice : Array [1..dimension,1..dimension]  of real;
Var u,v : vecteur ;
  M : matrice ;
```

7 Fonctions et procédures

7.1 Introduction

Les constructions qu'on va maintenant voir servent à regrouper plusieurs instructions. Cela est particulièrement intéressant dans les cas suivants :

- pour des morceaux de programmes qu'on va devoir faire plusieurs fois, en différents endroits du programme ;
- pour des morceaux de programmes qu'on veut isoler du reste, soit pour des raisons de clarté, soit pour pouvoir les changer plus facilement.

On a déjà vu et utilisé, au cours des exercices ou dans le cours, les fonctions `SQRT`, `SIN`, les procédures `writeln`, `readln`,... Dans tous ces cas, on est juste intéressé par le résultat du bloc d'instruction, pas par la manière de trouver ce résultat. Un autre exemple est la fonction `div`. On peut faire plusieurs blocs d'instructions différents pour obtenir le résultat, mais ce résultat sera toujours le même (tant qu'il n'y a pas d'erreurs).

7.2 Déclaration des fonctions, raffinement

Un dernier intérêt des procédures et des fonctions est l'utilisation pour la construction de programmes par raffinement. Par exemple, si on veut écrire un programme qui calcule le déterminant d'une matrice 3x3 donnée par l'utilisateur :

```
Program Calcul_de_determinant; Const dimension = 3 ;
  pi = 3.1415;
  message = 'Bienvenue.';
Type vecteur = Array [1..dimension] of real;
  matrice : Array [1..dimension,1..dimension] of real;
Var u,v : vecteur ;
  M : matrice ;
  det : float ;

  (* déclarations des fonctions et procédures *)

begin
  Entrer_matrice(M);
  det:=Calcul_determinant(M);
  writeln('le déterminant de cette matrice est ',det)
end.
```

On va voir comment déclarer les fonctions et les procédures, mais on voit qu'on peut tout de suite avoir un programme qui marche en faisant une procédure qui écrit `'Bonjour'` et une fonction qui renvoie toujours 1 (ou 0...). Dans le cadre d'un long travail, il est toujours utile de commencer à écrire ses fonctions, même si elles rendent une valeur qui n'a aucun sens. Cela permet de vérifier, étape par étape, que tout marche bien.

7.3 Les procédures

On utilise des procédures pour regrouper des instructions dont le but n'est pas de rendre un résultat de type `char`, `boolean`, `integer`,`real`. On déclare une procédure avec :

```
Procedure <nom de la procedure> (<liste d'arguments>);
  Var liste ;
begin
  liste d'instruction
end;
```

Les arguments de la procédure donnent des noms et des types aux valeurs avec lesquelles la procédure est appelée. Par exemple, dans le cas précédent, la procédure prend un argument de type matrice. Le nom qu'on donne à l'argument est celui qu'on utilisera pour cette valeur lors de l'utilisation de la procédure.

Les variables qui sont déclarées dans l'instruction `Var` ne peuvent être utilisées que pendant l'instruction. Elles n'ont pas de valeurs en dehors de la procédure.

Par exemple, si on veut faire une procédure qui affiche n astérisques, où n est donné par ailleurs, on a :

```
Procedure ligne (n:integer);
  Var i:integer ;
begin
  for i:= 1  to n  do
    write('*');(* affiche sans aller à la ligne*)
  writeln(* va à la ligne *)
end;
```

Le passage des arguments, lors de l'appel de la procédure, se fait normalement par valeur. Cela signifie que la valeur des arguments est donnée à la procédure pour son exécution, mais pas leur "nom". Pour voir la différence, il faut faire des petites boîtes :

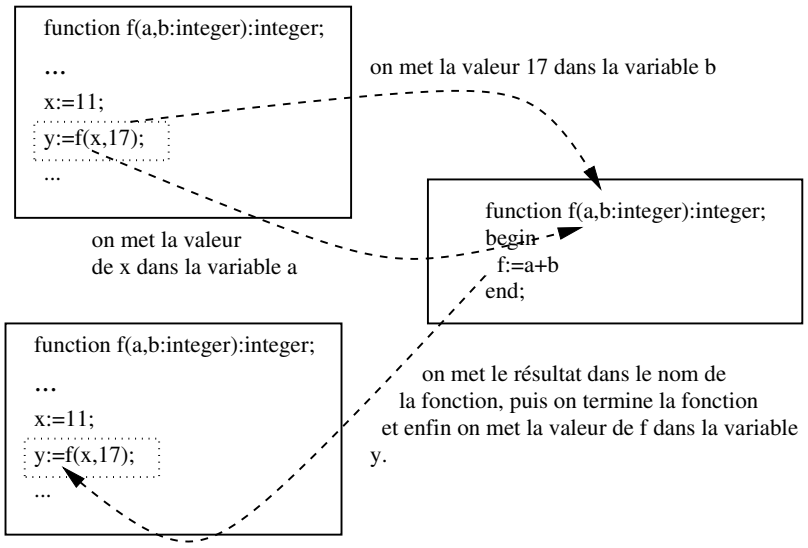

Figure 3: Schéma de la recopie des arguments lors d'un appel par valeur

Heureusement, il y a la possibilité d'appeler la procédure avec le *nom* de la variable. Dans ce cas, les changements qui seront faits à l'intérieur de la procédure changeront aussi la variable *à l'extérieur* de la procédure. C'est ce qui permet la procédure *Entrer_Matrice* vue plus haut. Pour cela, il faut préciser qu'on veut la variable, et pas uniquement sa valeur :

```
Procedure Entrer_matrice (Var M:matrice);
  Var i,j:integer ;
begin
(* instructions *)
end;;
```

Dans ce cas, l'appel de la procédure se fait suivant le schéma ci-dessous :

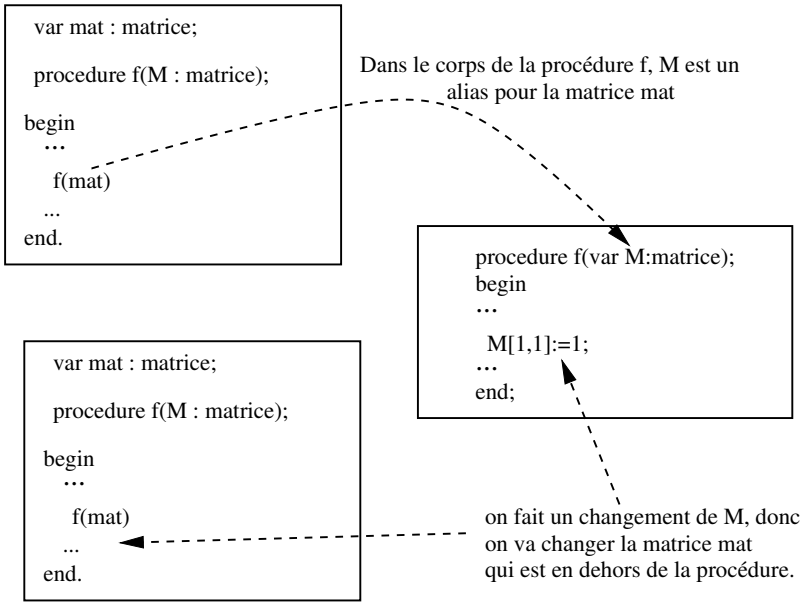

Figure 4: Schéma du renommage des variables lors d'un appel par nom

Il y a donc un renommage des variables entre l'intérieur et l'extérieur de la procédure. Mais c'est juste la possibilité d'appeler la même variable avec un alias, qui sera toujours le même, même lorsque les variables entre différents appels changent.

7.4 Les fonctions

C'est essentiellement la même chose, mis à part qu'une fonction rend une valeur d'un type simple (donc pas de tableau). Il faut :

1. préciser le type de la valeur retournée par la fonction ;
2. il doit y avoir une instruction qui donne cette valeur.

On utilise la déclaration suivante :

```
Function <nom_de_la_fonction> (arguments):type;
  Var liste ;
begin;
...
<nom_de_la_fonction>:= ... ; end;
```

Exercice :
Écrire un programme qui demande en entrée 2 matrices $3x3$, et qui calcule leur somme, et qui affiche le résultat.

1. de quelles procédures a-t-on besoin ?
2. lesquelles modifient leurs arguments ?
3. écrire ces procédures.

8 Algorithmes récursifs

8.1 Des exemples bien connus

En mathématiques, on prouve facilement que le pgcd de deux entiers positifs (et non-nuls) a et b, c'est :

- aussi le pgcd de a et $b-a$ si $a > b$ (on peut échanger les rôles de a et b) ;
- c'est a si $a = b$.

Partant de là, on a vu qu'on pouvait faire une fonction qui calcule le pgcd de 2 nombres :

```
Function pgcd(a,b :integer):integer;
(* fonction qui calcule le pgcd de a et b*)
begin
  if (a < 0) then a:=-a;
  if (b < 0) then b:=-b;
  if (a = 0) then
    pgcd:=b
  else
    if (b = 0) then
      pgcd:=b
    else
      while (a <> b) do
        if (a < b) then
          b:=b-a
        else
          a:=a-b;
  pgcd:=a;
end;
```

On a déjà vu cette fonction, mais elle s'éloigne un peu de la propriété mathématique, puiqu'elle utilise une boucle, alors qu'il n'y a pas d'itération dans la propriété.

Ce n'est pas le seul cas où on utilise une boucle là où il n'y en a pas à priori. On peut aussi prendre le cas de la fonction factorielle, définie pour un entier $n \geq 0$:

- $0! = 1$;
- $(n+1)! = (n+1).n!$.

La fonction qui calcule la factorielle d'un entier n positif peut s'écrire :

```
Function factorielle(a :integer):integer;
  Var temp,indice : integer;
(* fonction qui calcule la factorielle de a *)
begin
  temp :=1;
  for indice:=1 to a do
    temp:=temp*indice;
  factorielle:=temp ;
end;
```

On a donc là encore utilisé une boucle (une boucle `for` cette fois). Ces deux exemples ont un point commun :

La valeur de la fonction est définie par une autre valeur de la fonction

Par exemple, la valeur de $pgcd(6, 51)$ est définie comme étant égale à la valeur de $pgcd(6, 45)$. De même, la valeur de 7! est définie comme étant égale à la valeur de 6! multipliée par 7.

En mathématique, si on a une suite dont les valeurs sont définies en fonction d'autres valeurs de la suite, on parle de *suites récurrentes*. Et les suites sont des fonctions de N dans R !

En informatique, c'est pareil. On parle de *fonction récurrente* pour désigner une fonction dont la valeur est définie par rapport à une autre valeur de la fonction.

En pascal, c'est très facile d'écrire des fonctions récursives. Par exemple, pour le calcul du pgcd, on peut définir la fonction suivante, qui prend uniquement des entiers strictement positifs en entrée :

```
Function pgcd(a,b :integer):integer;
(* fonction qui calcule le pgcd de a et de b
  * de manière récursive *)
begin
  if (a = b ) then
    pgcd := a
  else
    if (a > b) then
      pgcd:= pgcd(a-b,b)
    else
      pgcd:= pgcd(a,b-a)
end;
```

De la même manière, on peut calculer la factorielle d'un entier a de manière récursive :

```
Function factorielle(a :integer):integer;
(* fonction qui calcule la factorielle de a
  * de manière récursive *)
begin
  if (a=0) then
    factorielle:=1
  else
    factorielle:=a*factorielle(a-1);
end;
```

8.2 Arrêt d'une fonction récursive

8.2.1 Le problème

La question qu'on peut se poser est la suivante. Si La valeur de la fonction en un point est définie par rapport à la valeur de la fonction en un autre point, qu'est-ce qui empêche le calcul de ne pas s'arrêter ?

La réponse est assez simple : rien, si on ne fait pas d'hypothèses supplémentaires. Par exemple, on peut définir la fonction tcaf, qu'on note ? :

- $0? = 1$;

- $n? = n * (n+1)?$.

Regardons un peu ce qui se passe. Si on veut calculer 0?, la valeur est donnée tout de suite, et c'est 1. Essayons maintenant de calculer 1?. On a :

$$1? = 1 * 2? = 1 * 2 * 3? = 1 * 2 * 3 * 4? = \dots$$

Ca ne terminera jamais ! On voit même que pour un entier n strictement positif, cette fonction ne donnera jamais de valeur. Par contre, on peut remarquer que la fonction $tcaf$ est bien définie pour tous les entiers ≤ 0. Et qu'il se passe la même chose que pour la fonction factorielle, qui elle ne sera pas définie pour les entiers négatifs.

C'est assez compliqué, aussi on va essayer d'y voir un peu plus clair.

8.2.2 Graphe de dépendance d'une fonction récursive

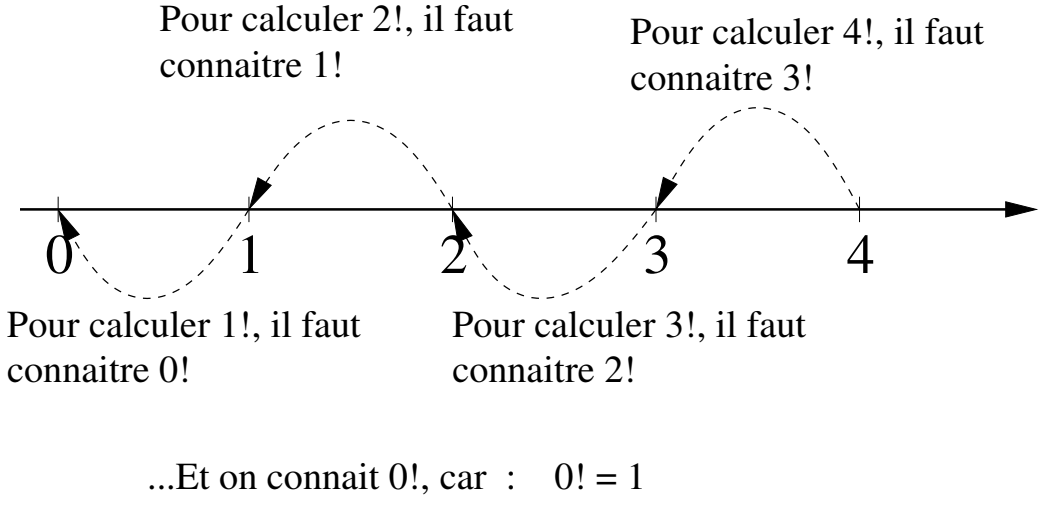

Figure 5: Graphe de dépendance des arguments de la fonction factorielle pour $n = 4$

On voit qu'on a une suite qui décroît, et qui est toujours positive. Théorème de mathématiques : toute suite décroissante et minorée converge. Et si une suite qui ne prend que des valeurs entières converge, alors, au bout d'un moment, elle est stationnaire (on ne bouge plus). Faire le raisonnement. Et si elle est strictement décroissante, alors on est obligé de s'arrêter.

Autrement dit, si, dans les appels d'une fonction récursive, on trouve une quantité entière qui est toujours positive et strictement décroissante, alors cette fonction donnera toujours une valeur (on s'arrêtera au bout d'un moment). Le problème est que cette quantité n'est pas toujours évidente, comme dans le cas de la fonction factorielle.

On a utilisé, sans le savoir, le *graphe de dépendance* des appels d'une fonction récursive.

Construction du graphe de dépendance

Pour donner ce graphe, il faut tracer, pour la valeur d'un argument, des flêches qui vont de cette valeur vers les valeurs des arguments dont on a besoin. Si, en partant d'un point, les chemins, en suivant ces flêches, est fini, le calcul de la valeur de ce point terminera.

On peut reprendre l'exemple de la fonction pgcd, en regardant ce qui se passe pour pgcd(7,4). Cette fois-ci, comme il y a deux arguments, on se place dans le plan :

On a bien l'impression que ça va toujours s'arrêter, mais ni a, ni b ne sont strictement décroissants. Içi, on a toujours $a + b$ qui est strictement décroissant, jusqu'à ce qu'on s'arrête.

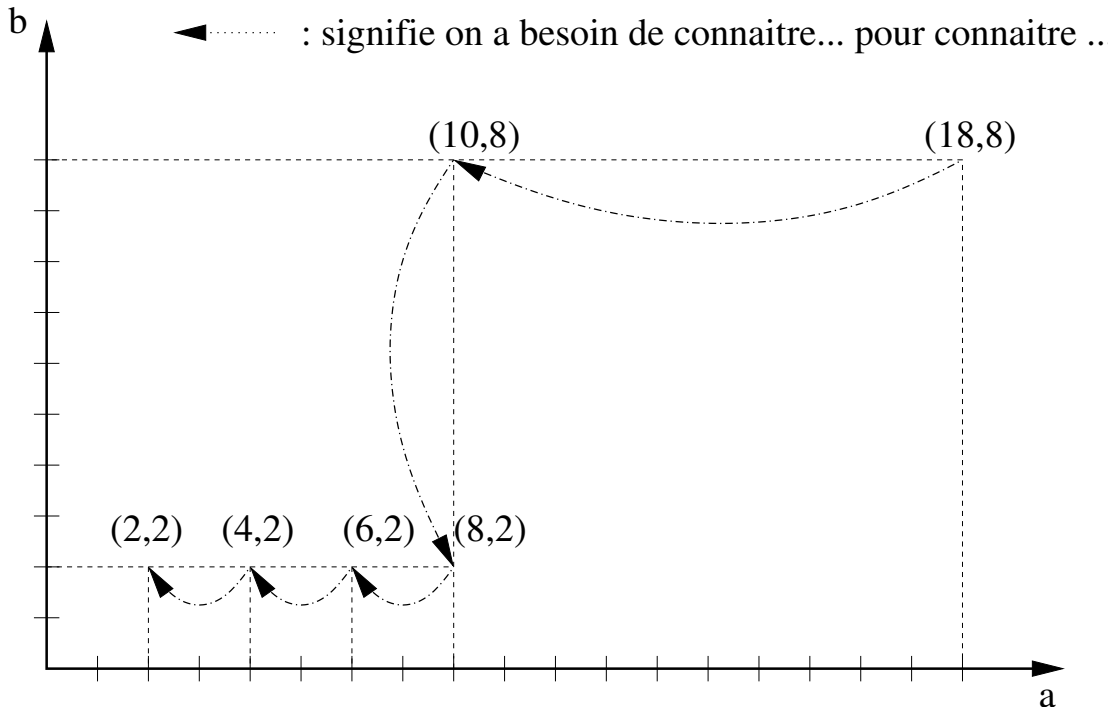

Figure 6: Graphe de dépendance des arguments de la fonction pgcd pour (7,4)

Règle de programmation : Lorsqu'on utilise une fonction récursive, il faut toujours mettre en commentaire la quantité qui est strictement décroissante.

Exercice :

On définit C_n^p pour un entier $n \geq 0$ et $0 \leq p \leq n$ par :

- si $p = 0$ ou $p = n$, $C_n^n = C_n^0 = 1$;
- sinon, $C_n^p = C_{n-1}^{p-1} + C_{n-1}^p$.

1. Le graphe de dépendance doit-il être tracé sur une droite ou sur un plan ?
2. tracer le graphe de dépendance pour C_4^2.

8.2.3 Fonctions mutuellement définies

On considère le cas de deux suites numériques $(u_n)_{n\in N}$ et $(v_n)_{n\in N}$, et on suppose qu'elles sont définies l'une par rapport à l'autre :

$$\begin{cases} u_0 = 1 & u_{n+1} = f(u_n, v_n) \\ v_0 = 1 & v_{n+1} = g(u_n, v_n) \end{cases}$$

On peut calculer les valeurs de ces suites de manière naïve. Il suffit de remarquer qu'une suite est une fonction des entiers vers les nombres réels :

```
(* déclaration des fonctions f et g *)

Function u(n:integer):real;
(* fonction qui calcule u_n *)
(* n décroit à chaque appel de u et v *)
begin
  if (n = 0) then
    u:= 1
  else
    u:=f(u(n-1),v(n-1))
end;

Function v(n:integer):real;
(* fonction qui calcule v_n *)
(* n décroit à chaque appel de u et v *)
begin
  if (n = 0) then
    v:= 1
  else
    v:=g(u(n-1),v(n-1))
end;
```

On commence par regarder si ce calcul est possible en faisant un graphe de dépendance. Par exemple, pour $n = 3$, on a besoin :

1. pour calculer u_3, il faut d'abord calculer u_2 et v_2 ;
2. pour calculer u_2, il faut d'abord calculer u_1 et v_1 ;
3. pour calculer v_2, il faut d'abord calculer u_1 et v_1 ;
4. pour calculer u_1, il faut d'abord calculer u_0 et v_0 ;
5. pour calculer v_1, il faut d'abord calculer u_0 et v_0 ;
6. on connaît les valeurs de u_0 et v_0.

On peut maintenant tracer le graphe de dépendance 7. On utilise 2 couleurs pour les appels à la fonction u et à la fonction v :

On voit sur ce graphe que toutes les flêches vont vers la gauche (n décroît strictement). On peut vérifier, sur la formule, que n est strictement décroissant entre chaque appel à la fonction u ou à la fonction v.

Mais il y a un problème : dans la déclaration de la fonction u, on utilise la fonction v qui n'est pas encore définie à ce moment. On peut alors décider de changer l'ordre de déclaration de u et v, mais dans ce cas, ce sera dans la fonction v qu'il y aura un problème : la fonction u ne sera pas déclarée.

Il faut regarder d'un peu plus près, car on n'avait pas ce problème pour une fonction simplement récursive, qui est simplement définie à partir d'elle-même. On commence par un peu de vocabulaire :

En-tête de la fonction	`Function u(n:integer):real;` (* fonction qui calcule u_n *) (* n décroît à chaque appel des fonctions u et v *)

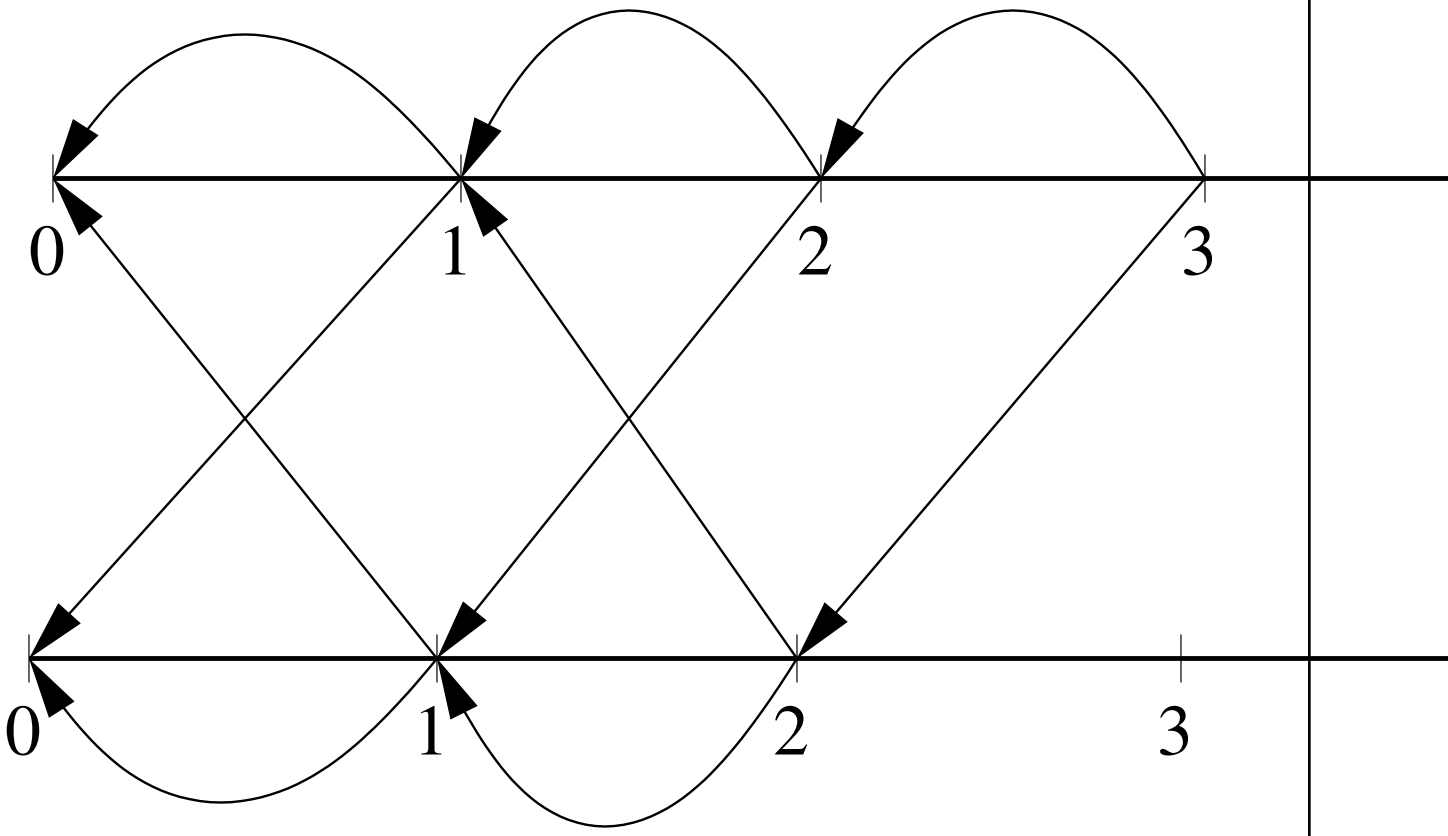

Figure 7: Graphe de dépendance entre les appels

Corps de la fonction

```
begin
  if (n = 0) then
    u:= 1
  else
    u:=f(u(n-1),v(n-1))
end;
```

Et les en-têtes sont utilisés de la manière suivante :

En `Pascal`, une fonction ou une procédure est déclarée à partir du moment où son en-tête est déclaré.

Donc, pour résoudre notre problème avec les fonctions u et v, il suffit de procéder de la manière suivante :

1. on déclare l'en-tête de la fonction v ;

2. on écrit la fonction u ;

3. on écrit le corps de la fonction v.

Il ne reste plus qu'un petit problème à résoudre : comment dire à `Pascal` que le corps de la fonction v doit être relié à l'en-tête qui a été déclaré bien avant ?

Pour cela, on précise, lors de la déclaration de l'en-tête, qu'on donnera le corps de la fonction plus loin :

```
Function u(n:integer):real; Forward
```

Et lors de la déclaration du corps de la fonction, on rappelle juste le *nom* de la fonction qu'on est en train de définir :

```
Function v;
(* fonction qui calcule v_n *)
(* n décroit à chaque appel de u et v *)
begin
  if (n = 0) then
    v:= 1
  else
    v:=g(u(n-1),v(n-1))
end;
```

On ne rappelle *que le nom*. Les arguments de la fonction sont déjà définis. Si des variables sont utilisées dans le corps de la fonction, c'est lors de la déclaration du corps de la fonction qu'on les marque, pas lors de la déclaration de l'en-tête. Le programme pour calculer les suites u et v devient :

```
Program CalculRecursif;
(* calcul de 2 suites mutuellements récursives *)
(* Il n'est pas nécessaire de calculer des variables *)

(* déclaration des fonctions f et g *)
Function f(x,y:real):real; (* par exemple *)
f:=x/y;
Function g(x,y:real):real; (* par exemple *)
g:=sqrt(x*y);

(* déclaration des fonctions "suite" *)
Function v(n:integer):real; Forward ;(* en-tête de la fonction v *)
Function u(n:integer):real;
(* fonction qui calcule u_n *)
(* n décroit à chaque appel de u et v *)
begin
  if (n = 0) then
    u:= 1
  else
    u:=f(u(n-1),v(n-1))
end;

Function v;
(* fonction qui calcule v_n *)
(* n décroit à chaque appel de u et v *)
begin
  if (n = 0) then
    v:= 1
  else
    v:=g(u(n-1),v(n-1))
end;

begin
  writeln('u(30) vaut',u(30));
end.
```

8.2.4 Taille en mémoire lors d'un calcul récursif

On reprend un calcul simple, celui des coefficients binomiaux C_n^p. On a déjà vu en cours une fonction récursive qui permet de les calculer, et on a vu en TD comment les calculer en utilisant une boucle `for` . On peut se demander Quelle méthode est la meilleure ?

Une fois qu'on est habitué à leur écriture, les algorithmes récursifs sont beaucoup plus simple que les algorithmes utilisant des boucles, car il n'y a plus besoin de s'occuper des initialisations de compteur, ni de chercher une manière de calculer qui n'est pas contenue dans la définition de la fonction.

> Utiliser des fonctions récursives, c'est donc minimiser le risque de taper une erreur.

C'est pour cette raison que les langages utilisant ces fonctions sont très utilisés pour écrire de gros programme, où il est surtout important de ne pas faire d'erreurs. L'inconvénient, c'est que pour calculer C_{12}^6, par exemple, on va réduire ce calcul à une grande somme dont tous les termes valent 1. Autrement dit, on va devoir garder en mémoire plus de 1000 variables qui attendent leur valeur pour faire ce calcul, alors qu'en reprenant le programme vu en TD, cela se fait en une cinquantaine de variables différentes. Et ce fossé grandit quand on prend des nombres plus grands.

Il est alors possible qu'un calcul prenne tellement de temps avec un algorithme récursif qu'on préfère parfois utiliser un calculer itératif. Dans l'exemple de calcul de suite qui vient d'être donné, il n'est pas certain que la manière récursive de calculer termine pour 30, alors qu'une suite de cette taille a été traîtée jusqu'au rang 1000 en TD, en faisant une boucle `for` .

9 Conclusion : Forme d'un programme `Pascal`

Les langages de programmation sont donnés par des grammaires, qui attendent des mots ou de signes spéciaux. Il n'est pas question içi d'aller très loin dans cette grammaire, mais juste de donner la forme que doit avoir un programme `Pascal` lorsqu'on suit la grammaire `Pascal`.

La forme générale d'un programme `Pascal` est :

```
Program nom_du_programme ;

Déclarations ;

begin
  <liste d'instruction>;
end.
```

Dans les déclarations, on met dans cet ordre :

1. les déclarations de constantes ;
2. les déclarations de type ;
3. les déclaration d'en-tête de fonctions et de procédures ;
4. les déclarations de fonctions et procédures.