Chapitre 3
Tests et logique booléenne

1. Les tests et conditions

1.1 Principe

Dans le précédent chapitre, vous avez pu vous familiariser avec les expressions mettant en place des opérateurs, qu'ils soient de calcul, de comparaison ou booléens. Ces opérateurs et expressions trouvent tout leur sens une fois utilisés dans des conditions (qu'on appelle aussi des branchements conditionnels). Une expression évaluée est ou vraie (le résultat est différent de zéro) ou fausse. Suivant ce résultat, l'algorithme va effectuer une action, ou une autre. C'est le principe de la condition.

Grâce aux opérateurs booléens, l'expression peut être composée : plusieurs expressions sont liées entre elles à l'aide d'un opérateur booléen, éventuellement regroupées avec des parenthèses pour en modifier la priorité.

```
(a=1 OU (b*3=6)) ET c>10
```

est une expression tout à fait valable. Celle-ci sera vraie si chacun de ses composants respecte les conditions imposées. Cette expression est vraie si a vaut 1 et c est supérieur à 10 ou si b vaut 2 (2*3=6) et c est supérieur à 10.

Reprenez l'algorithme du précédent chapitre qui calcule les deux résultats possibles d'une équation du second degré. L'énoncé simplifié disait que pour des raisons pratiques seul le cas où l'équation a deux solutions fonctionne. Autrement dit, l'algorithme n'est pas faux dans ce cas de figure, mais il est incomplet. Il manque des conditions pour tester la valeur du déterminant : celui-ci est-il positif, négatif ou nul ? Et dans ces cas, que faire et comment le faire ?

Imaginez un second algorithme permettant de se rendre d'un point A à un point B. Vous n'allez pas le faire ici réellement, car c'est quelque chose de très complexe sur un réseau routier important. De nombreux sites internet vous proposent d'établir un trajet avec des indications. C'est le résultat qui est intéressant. Les indications sont simples : allez tout droit, tournez à droite au prochain carrefour, faites trois kilomètres et au rond-point prenez la troisième sortie direction B. Dans la plupart des cas, si vous suivez ce trajet vous arrivez à bon port. Mais quid des impondérables ? Par où allez-vous passer si la route à droite au prochain carrefour est devenue un sens interdit (cela arrive parfois, y compris avec un GPS, prudence) ou que des travaux empêchent de prendre la troisième sortie du rond-point ?

Reprenez le trajet : allez tout droit. Si au prochain carrefour la route à droite est en sens interdit : continuez tout droit puis prenez à droite au carrefour suivant puis à gauche sur deux kilomètres jusqu'au rond-point. Sinon : tournez à droite et faites trois kilomètres jusqu'au rond-point. Au rond-point, si la sortie vers B est libre, prenez cette sortie. Sinon, prenez vers C puis trois cents mètres plus loin tournez à droite vers B.

Ce petit parcours ne met pas uniquement en lumière la complexité d'un trajet en cas de détour, mais aussi les nombreuses conditions qui permettent d'établir un trajet en cas de problème. Si vous en possédez, certains logiciels de navigation par GPS disposent de possibilités d'itinéraire bis, de trajectoire d'évitement sur telle section, ou encore pour éviter les sections à péage. Pouvez-vous maintenant imaginer le nombre d'expressions à évaluer dans tous ces cas de figure, en plus de la vitesse autorisée sur chaque route pour optimiser l'heure d'arrivée ?

1.2 Que tester ?

Les opérateurs s'appliquent sur quasiment tous les types de données, y compris les chaînes de caractères, tout au moins en pseudo-code algorithmique. Vous pouvez donc quasiment tout tester. Par tester, comprenez ici évaluer une expression qui est une condition. Une condition est donc le fait d'effectuer des tests pour, en fonction du résultat de ceux-ci, effectuer certaines actions ou d'autres.

Une condition est donc une affirmation : l'algorithme et le programme ensuite détermineront si celle-ci est vraie, ou fausse.

Une condition retournant VRAI ou FAUX a comme résultat un **booléen**.

En règle générale, une condition est une comparaison même si en programmation une condition peut être décrite par une simple variable (ou même une affectation) par exemple. Pour rappel, une comparaison est une expression composée de trois éléments :

– une première valeur : variable ou scalaire ;

– un opérateur de comparaison ;

– une seconde valeur : variable ou scalaire.

Les opérateurs de comparaison sont :

– l'égalité : **=** ;

– la différence : **!=** ou **<>** ;

– inférieur : **<** ;

– inférieur ou égal : **<=** ;

– supérieur : **>** ;

– supérieur ou égal : **>=**.

Le pseudo-code algorithmique n'interdit pas de comparer des chaînes de caractères. Évidemment, vous prendrez soin de ne comparer que les variables de types compatibles. Dans une condition, une expression sera toujours évaluée comme étant soit vraie, soit fausse.

Remarque

L'opérateur d'affectation peut aussi être utilisé dans une condition. Dans ce cas, si vous affectez 0 à une variable, l'expression sera fausse et si vous affectez n'importe quelle autre valeur, elle sera vraie.

En langage courant, il vous arrive de dire "choisissez un nombre entre 1 et 10". En mathématique, vous écrivez cela comme ceci :

```
1 ≤ nombre ≤ 10
```

Si vous écrivez ceci dans votre algorithme, attendez-vous à des résultats surprenants le jour où vous allez le convertir en véritable programme. En effet les opérateurs de comparaison ont une priorité, ce que vous savez déjà, mais l'expression qu'ils composent est aussi souvent évaluée de gauche à droite. Si la variable nombre contient la valeur 15, voici ce qui se passe :

– L'expression 1 <= 15 est évaluée : elle est vraie.
– Et ensuite ? Tout va dépendre du langage, l'expression suivante vrai <= 10 peut être vraie aussi.
– La condition est vérifiée et le code associé va être exécuté !

Vous devez donc proscrire cette forme d'expression. Voici celles qui conviennent dans ce cas :

```
nombre>=1 ET nombre<=10
```

Ou encore :

```
1<=nombre ET nombre<=10
```

1.3 Tests SI

1.3.1 Forme simple

Il n'y a, en algorithmique, qu'une seule instruction de test, "**Si**", qui prend cependant deux formes : une simple et une complexe. Le test SI permet d'exécuter du code si la condition (la ou les expressions qui la composent) est vraie.

La forme simple est la suivante :

```
Si booléen Alors
  Bloc d'instructions
FinSi
```

Remarque

Notez ici que le booléen est la condition. Comme indiqué précédemment, la condition peut aussi être représentée par une seule variable. Si elle contient 0, elle représente le booléen FAUX, sinon le booléen VRAI.

Que se passe-t-il si la condition est vraie ? Le bloc d'instructions situé après le "**Alors**" est exécuté. Sa taille (le nombre d'instructions) n'a aucune importance : de une ligne à n lignes, sans limite. Dans le cas contraire, le programme continue à l'instruction suivant le "**FinSi**". L'exemple suivant montre comment obtenir la valeur absolue d'un nombre avec cette méthode.

```
PROGRAMME ABS
VAR
  Nombre :entier
DEBUT
  nombre←-15
  Si nombre<0 Alors
    nombre←-nombre
  FinSi
  Afficher nombre
FIN
```

En PHP, c'est le "if" qui doit être utilisé avec l'expression booléenne entre parenthèses. La syntaxe est celle-ci :

```
if(boolean) { /*code */ }
```

Si le code PHP ne tient que sur une ligne, les accolades peuvent être supprimées, comme dans l'exemple de la valeur absolue. Cet exemple montre également une seconde possibilité offerte par les bibliothèques de fonction de PHP.

```
<html>
  <head><meta/>
    <title>ABS</title>
  </head>
  <body>
  <?php
```

```
    $number=-15;
    if($number<0) $number=-$number;
    echo $number;

    echo "<br />";
  // seconde possibilitité
    $number2=-32;
    $number2=abs($number2);
    echo $number2;
  ?>
  </body>
</html>
```

1.3.2 Forme complexe

La forme complexe n'a de complexe que le nom. Il y a des cas où il faut exécuter quelques instructions si la condition est fausse sans vouloir passer tout de suite à l'instruction située après le FinSi. Dans ce cas, utilisez la forme suivante :

```
Si booléen Alors
  Bloc d'instructions
Sinon
  Bloc d'instructions
FinSi
```

Si la condition est vraie, le bloc d'instructions situé après le Alors est exécuté. Ceci ne diffère pas du tout de la première forme. Cependant, si la condition est fausse, cette fois c'est le bloc d'instructions situé après le Sinon qui est exécuté. Ensuite, le programme reprend le cours normal de son exécution après le FinSi.

Notez que vous auriez pu très bien faire un équivalent de la forme complexe en utilisant deux formes simples : la première avec la condition vraie, la seconde avec la négation de cette condition. Mais ce n'est pas très joli, même si c'est correct. Retenez que :

– Si dans une forme complexe, l'un des deux blocs d'instructions est vide, alors transformez-la en forme simple : modifiez la condition en conséquence.

– Laisser un bloc d'instructions vide dans une forme complexe n'est pas conseillé : c'est une grosse maladresse de programmation qui peut être facilement évitée. Cependant, certains langages l'autorisent, ce n'est pas une erreur ni même une faute lourde, mais ce n'est pas une raison…

L'algorithme suivant est une illustration de la forme complexe. Elle vérifie si trois valeurs entrées au clavier sont triées par ordre croissant :

```
PROGRAMME TRI
VAR
  x,y,z:entiers
DEBUT
  Afficher "Entrez trois valeurs entières distinctes"
  Saisir x,y,z
  Si z>y ET y>x Alors
    Afficher "Triés en ordre croissant"
 Sinon
    Afficher "Ces nombres ne sont pas triés"
 FinSi
FIN
```

Le code suivant en PHP n'est pas bien difficile à comprendre.

```
<html>
  <head><meta/>
    <title>trie ?</title>
  </head>
  <body>
  <?php
  if(!isset($_GET['x'])) {

  //si il existe une variable x dans l'URL
  ?>
    <form method="GET">
      x : <input type="text" size="4" name="x" /><br />
      y : <input type="text" size="4" name="y" /><br />
      z : <input type="text" size="4" name="z" /><br />
      <input type="submit" name="OK" /><br />
    </form>
  <?php
  } else { //sinon
    $x=$_GET['x'];
    $y=$_GET['y'];
    $z=$_GET['z'];

       //si x > y et y > x
    if($z>$y && $y>$x)
      echo "Tries en ordre croissant";
    else
      echo "Ces nombres ne sont pas tries";
 }
 ?>
 </body>
</html>
```

1.4 Tests imbriqués

Vous connaissez le dicton populaire « avec des si, on mettrait Paris en bouteille », qui répond généralement à une accumulation de conditions « s'il fait beau et qu'il fait chaud et que la voiture n'est pas en panne, alors nous irons à la mer, sinon si la voiture est en panne nous prendrons le train, sinon s'il y a grève alors nous ferons du stop, sinon si tout va mal alors nous resterons à la maison ». C'est un peu lourd dit comme cela, mais qui n'a jamais échafaudé des plans douteux devant vérifier plusieurs conditions avec des scénarios de secours ?

Vous retrouverez parfois le même problème en programmation. Les deux formes de Si ci-dessus permettent de s'en sortir assurément, mais la syntaxe peut devenir très lourde. Vous pouvez en effet parfaitement imbriquer vos tests en plaçant des Si en cascade dans les blocs d'instructions situés après les Alors et les Sinon. Prenez l'exemple suivant : il faut deviner si un nombre saisi au clavier est proche ou non d'une valeur prédéfinie. Pour le savoir, le programme affiche froid, tiède, chaud ou bouillant suivant l'écart entre la valeur saisie et celle prédéfinie. Cet écart est calculé avec une simple soustraction, puis en déterminant sa valeur absolue.

```
PROGRAMME IMBRIQUE
VAR
  nombre,saisie,ecart :entiers
DEBUT
  Nombre←63
  Afficher "Saisissez une valeur entre 0 et 100 :"
  Saisir saisie
  ecart←nombre-saisie
  Si ecart < 0 Alors
    Ecart←-ecart
  FinSi
  Si ecart=0 Alors
    Afficher "Bravo !"
  Sinon
    Si ecart<5 Alors
      Afficher "Bouillant"
    Sinon
      Si ecart<10 Alors
        Afficher "Chaud"
      Sinon
```

```
        Si ecart<15 Alors
          Afficher "Tiède"
        Sinon
          Afficher "Froid"
        FinSi
      FinSi
    FinSi
  FinSi
FIN
```

Peut-être trouvez-vous cette syntaxe trop longue et surtout peu lisible. Une astuce consiste à tracer des traits verticaux allant des Si aux FinSi pour ne pas s'y perdre. C'est d'ailleurs recommandé par certains professeurs d'algorithmique, y compris pour les boucles (abordées dans un prochain chapitre). L'algorithme ci-dessus est parfaitement valable et d'une syntaxe correcte. Cependant, il est possible de faire plus concis avec la forme suivante.

```
Si booléen Alors
  Bloc d'instructions 1
SinonSi booléen Alors
  Bloc d'instructions 2
SinonSi booléen Alors
  Bloc d'instructions n
Sinon
  Bloc d'instruction final
FinSi
```

Cette forme est plus propre, plus lisible et évite de se mélanger les pinceaux. De multiples conditions sont testées. Si la première n'est pas vraie, on passe à la deuxième, puis à la troisième et ainsi de suite, jusqu'à ce qu'une condition soit vérifiée. Dans le cas contraire, c'est le bloc d'instructions final qui est exécuté. Les tests du précédent exemple doivent donc être réécrits comme ceci :

```
Si ecart=0 Alors
  Afficher "Bravo !"
SinonSi ecart<5 Alors
  Afficher "Bouillant"
SinonSi ecart<10 Alors
  Afficher "Chaud"
SinonSi ecart<15 Alors
  Afficher "Tiède"
Sinon
  Afficher "Froid"
FinSi
```