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 (l'égalité) 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 pas mélanger les torchons et les serviettes, en ne comparant que les variables de types compatibles. Dans une condition une expression, quel que soit le résultat de celle-ci, 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 ce 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.
– Le résultat est épouvantable : la condition est vérifiée et le code correspondant 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
```

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 une seconde possibilité par les mécanismes offerts par les bibliothèques de fonction de PHP.

```
<html>
  <head><meta/>
    <title>ABS</title>
  </head>
  <body>
  <?php
    $nombre=-15;
```

```
    if($nombre<0) $nombre=-$nombre;
    echo $nombre;

    echo "<br />";
  // seconde possibilitité
    $nombre2=-32;
    $nombre2=abs($nombre2);
    echo $nombre2;
  ?>
  </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, c'est un programme sale. Cependant, certains langages l'autorisent, ce n'est pas une erreur ni même une faute lourde, mais ce n'est pas une raison…