Chapitre 3
Conditions, tests et booléens

1. Les tests et conditions

1.1 Les conditions sont primordiales

Dans notre vie, notre comportement est dirigé par une multitude de décisions à prendre. Toujours avec l'exemple du passage piéton, nous allons traverser si le voyant piéton est vert, sinon nous allons attendre **s'il** est rouge. Il en va de même pour un algorithme et un programme, nous devons guider l'ordinateur pour qu'il puisse prendre les bonnes décisions et donc exécuter correctement nos instructions.

Souvenez-vous que nous devons tout expliquer à la machine, elle ne prendra jamais une décision par elle-même, sauf peut-être le fait de s'éteindre en cas de court-circuit. Vous devez indiquer à l'ordinateur dans quels cas il peut exécuter vos instructions. Par exemple comme quand un adulte apprend à un jeune enfant à dessiner : l'adulte lui montre comment tenir le crayon, quel est le bout qui permet de dessiner, qu'on ne peut dessiner que sur une feuille et non sur une table ou un mur, etc. L'avantage de la programmation pour nous est que nos explications sont beaucoup plus simples à formuler et que la machine nous écoute forcément sans jamais en faire à sa tête et surtout comprend parfaitement du premier coup.

Les tests et conditions représentent une idée de base très simple pour bien guider notre programme : choisir d'exécuter telle ou telle instruction selon la validité d'une condition. Nous testons donc la validité d'une condition pour donner l'autorisation ou non de continuer le déroulement du programme comme : **si** le feu piéton est vert (condition), **alors** je traverse (instruction).

Le test peut également contenir une ou plusieurs alternatives : **si** le feu piéton est vert (condition), **alors** je traverse (instruction) **sinon** j'attends que le feu piéton passe au vert.

Qui dit condition, dit booléen. Les booléens sont les types les plus simples en informatique. Ils ne peuvent prendre que deux valeurs : `VRAI` ou `FAUX`. La relation entre une condition et un booléen est donc totalement explicite car une condition est toujours une expression de type booléenne.

Une condition est systématiquement le résultat d'une comparaison ou de plusieurs comparaisons reliées entre elles par les opérateurs logiques (`ET`, `OU`, `NON`).

Commençons par étudier les structures conditionnelles avec une seule comparaison pour introduire par la suite la logique booléenne (ou algèbre de Boole) pour gérer des tests avec plusieurs conditions.

1.2 Structures conditionnelles

En algorithmie, il existe deux structures conditionnelles :

- Le **SI ALORS SINON** permet de tester n'importe quelle condition avec, ou non, une ou plusieurs alternatives.
- Le **CAS PARMI,** lui, ne permet que de tester plusieurs conditions d'égalité avec une même variable en une seule instruction.

1.2.1 SI ALORS SINON

Le SI algorithmique est un bloc d'instructions soumises à une condition pour être exécutées. De ce fait, toutes les lignes comprises dans le SI doivent être **indentées** avec une nouvelle tabulation.

Remarque

Nous rappelons l'importance de l'indentation d'un algorithme ou d'un programme. L'indentation permet d'avoir une lecture simplifiée car nous pouvons voir sans réfléchir où commence un bloc et où il finit. Pour le moment, vous ne pouvez pas vraiment vous rendre compte de son importance, mais dans la suite de cet ouvrage, cela deviendra plus que pertinent avec nos premiers programmes complexes et complets.

Pour indiquer les instructions à exécuter si la condition se vérifie, nous devons les faire précéder du mot-clé ALORS. Le déroulement de l'algorithme reprend normalement, sans test donc, après le mot-clé FINSI.

Voici la syntaxe du SI ALORS :

```
SI (conditionnelle)
ALOR
   ...      // instructions à exécuter si la conditionnelle
   ...      // est vraie
FINSI
```

Exemple :

```
PROGRAMME Entier_positif
VAR
   x : ENTIER
DEBUT
   ECRIRE("Entrez un entier de votre choix")
   x <- LIRE()
   SI x > 0
   ALORS
      ECRIRE("Votre entier est positif")
   FINSI
FIN
```

Dans l'algorithme précédent, nous testons si un entier entré par l'utilisateur est positif. Que se passe-t-il si nous voulons également tester si l'entier est négatif ? Votre première pensée serait d'ajouter un nouveau SI.

Avec deux SI à la suite, l'algorithme teste quoiqu'il advienne les deux conditions, si l'entier est positif puis si l'entier est négatif, ou inversement selon l'ordre des deux SI.

Cependant, nous sommes d'accord, un entier ne peut pas être positif et négatif à la fois. Ajoutons donc simplement un SINON à notre algorithme pour les négatifs :

```
PROGRAMME Entier_positif_negatif
VAR
   x : ENTIER
DEBUT
   ECRIRE("Entrez un entier de votre choix")
   x <- LIRE()
   SI x > 0
   ALORS
      ECRIRE("Votre entier est positif")
   SINON
      ECRIRE("Votre entier est négatif")
   FINSI
FIN
```

Notre algorithme devient de plus en plus juste mais il nous reste un point à définir : l'entier valant zéro. Zéro n'est ni positif ni négatif, c'est donc un cas particulier.

Nous pouvons ajouter un SI au début de l'algorithme pour tester la valeur zéro mais cette solution n'est pas optimale. Effectivement, quelle que soit la valeur de l'entier, il sera testé deux fois : une pour l'égalité et une pour la supériorité à cause des SI qui se suivent.

Une solution propre et optimisée est de définir le zéro comme cas par défaut du SINON en y ajoutant un nouveau SI, testant si l'entier est négatif SINON c'est qu'il est à zéro (ni positif ni négatif). Mettre un SI dans SI est appelé imbriquer des instructions ou **tests imbriqués**.

```
PROGRAMME Entier_positif_negatif_nul
VAR
   x : ENTIER
DEBUT
   ECRIRE("Entrez un entier de votre choix")
   x <- LIRE()
   SI x > 0
```

```
   ALORS
         ECRIRE("Votre entier est positif")
   SINON
      SI x < 0
      ALORS
         ECRIRE("Votre entier est négatif")
      SINON
         ECRIRE("Votre entier est nul")
      FINSI
   FINSI
FIN
```

Remarque

Pensez toujours à limiter vos instructions et vos variables dans vos algorithmes et programmes afin d'optimiser la gestion de la mémoire pour les raisons évoquées au chapitre précédent et donc de rendre vos codes plus performants.

1.2.2 CAS PARMI

Lorsque nous imbriquons beaucoup de `SI`, notre algorithme peut devenir difficile à lire, donc à comprendre, donc à corriger en cas d'erreur. Une alternative possible est d'utiliser le `CAS PARMI`. Cette structure conditionnelle permet de tester la valeur d'une variable et de la comparer, **en termes d'égalité uniquement**, à plusieurs autres valeurs. Comme le `SI`, elle donne la possibilité d'avoir un cas par défaut si aucune des valeurs testées n'est correcte. En voici la syntaxe :

```
CAS variable PARMI :
   CAS1 : valeur1
      ...   // instructions à réaliser si variable vaut valeur1
   CAS2 : valeur2
      ...   // instructions à réaliser si variable vaut valeur2
   ...
   PARDEFAUT
      ...    // instructions à réaliser par défaut. Optionnel
FINCASPARMI
```

Vous pouvez tester autant de cas que nécessaire. Le cas par défaut est tout à fait facultatif.

Écrivons un algorithme permettant d'afficher le nom du mois en fonction de son numéro donné par l'utilisateur. Pour des raisons de lecture, nous nous limiterons aux six premiers mois de l'année. Notre première version sera écrite uniquement avec des `SI` et la deuxième avec le `CAS PARMI`.

```
PROGRAMME Mois_si_imbriques
VAR
   mois : ENTIER
DEBUT
   ECRIRE("Entrer un chiffre entre 1 et 6 compris")
   mois <- LIRE()
   SI mois = 1
   ALORS
       ECRIRE("Janvier")
   SINON
       SI mois = 2
       ALORS
          ECRIRE("Fevrier")
       SINON
          SI mois = 3
          ALORS
              ECRIRE("Mars")
          SINON
             SI mois = 5
             ALORS
               ECRIRE("Mai")
               SINON
               ECRIRE("Juin")
                 FINSI
            FINSI
       FINSI
   FINSI
FIN
```

```
PROGRAMME Mois_cas_parmi
VAR
   mois : ENTIER
DEBUT
   ECRIRE("Entrer un chiffre entre 1 et 6 compris")
   mois <- LIRE()
   CAS mois PARMI :
      CAS : 1
         ECRIRE("Janvier")
      CAS : 2
         ECRIRE("Février")
      CAS : 3
         ECRIRE("Mars")
      CAS : 4
         ECRIRE("Avril")
      CAS : 5
         ECRIRE("Mai")
      PARDEFAUT
         ECRIRE("Juin")
   FINCASPARMI
FIN
```

Nous remarquons facilement, que ce soit à l'écriture ou à la lecture, que l'algorithme utilisant des `SI` devient très rapidement complexe et peut nous inciter à une erreur, d'indentation ou de syntaxe par exemple. Avec l'algorithme utilisant le `CAS PARMI`, l'écriture et la lecture sont vraiment naturelles car la syntaxe est plus simple. Cependant, n'oubliez pas que le `CAS PARMI` ne peut que tester des égalités alors que `SI` peut tester tout type de comparaison.

Regardons maintenant comment combiner plusieurs tests dans une structure conditionnelle.

2. La logique booléenne

2.1 Conditions multiples

Écrire une condition qui teste la validité d'un seul fait est assez logique, voire simple. Les conditions augmentent en complexité dès lors que nous augmentons le nombre de fais à valider, que ce soit dans la vie de tous les jours ou en informatique.

Lorsque nous testons des conditions multiples en tant qu'être humain, notre cerveau raisonne tellement vite que nous n'avons pas l'impression de réfléchir ni même de résoudre une équation. Et pourtant…

Reprenons l'exemple du passage piéton. Il vous paraît normal, avant de traverser à un feu, de vérifier que le voyant piéton est vert et d'également vérifier qu'aucune voiture ne grille le feu. Vous analysez donc deux conditions et avez l'impression de les faire en même temps avec une seule et unique condition. Mais votre cerveau reçoit bien deux conditions à vérifier, donc il résout ces deux conditions dans une équation.

Comme nous l'avons indiqué dans le chapitre d'introduction de cet ouvrage, vous devez tout décrire à la machine, faire du vrai pas-à-pas, aucun raccourci n'est possible. Il nous faut donc une manière simple de représenter les conditions multiples.

Pour formuler correctement cette équation avec plusieurs conditions, George Boole, un mathématicien du XIX^{e} siècle, créa une algèbre binaire que Claude Shannon utilisa plus d'un siècle plus tard en informatique. L'algèbre de Boole, appelée aussi logique booléenne selon le contexte, permet d'analyser plusieurs conditions en même temps dans une même équation, donc dans une même structure conditionnelle. Vous comprenez maintenant d'où vient le nom du type booléen.

L'implémentation de cette algèbre est quasiment naturelle en informatique. Un ordinateur fonctionnant par impulsion électrique, le `FAUX` est donc représenté par le 0 ou l'absence de courant, le `VRAI` par le 1 ou la présence d'un courant.