SUPPORT DE COURS B1-Dev

Date	Révision
Août 2020	Adaptation

TABLE DES MATIÈRES

1 Les éléments d'un programme....3
1.1 Syntaxe....3
1.2 Conditions....4
1.2.1 Conditions simples....4
1.2.2 Conditions complexes....5
1.2.3 Essai 1 : Fusée Apollo....7
1.3 Répétitions....8
1.3.1 Répétition simple par comptage....8
1.3.2 Répétitions simples par condition....9
1.3.3 Répétitions imbriquées....9
1.3.4 Exercices : les répétitions....10
2 Les algorithmes....11
2.1 Les fonctions....13
2.1.1 Fonctions internes....13
2.1.2 Les fonctions personnelles....13
2.1.2.1 Structure d'une fonction....13
2.1.2.2 exemple d'une fonction "carré"....15
2.1.3 Utilité des fonctions....16
2.1.4 Fonctions et paramètres....17
2.1.4.1 Paramètre en entrée....17
2.1.4.2 Paramètre en sortie....19
2.1.5 Fonction : en résumé....20
2.1.6 Exercices sur les fonctions....21

1 LES ÉLÉMENTS D'UN PROGRAMME

La programmation reprend des éléments de fonctionnement de la pensée. La difficulté est moins d'apprendre un langage que de pouvoir exprimer ses réflexions.

Nous verrons que rendre la monnaie avec des pièces et des billets est un programme simple mais qui nécessite de décortiquer le fonctionnement de notre pensée.

Dans un premier temps nous allons décrire la "syntaxe" utilisée dans la plupart des langages de programmation ; par la suite, nous pourrons étudier deux principes importants en programmation : les conditions et les répétitions.

1.1 SYNTAXE

La plupart des langages de programmation dit "de haut niveau" se ressemble par leurs syntaxes : en effet, quel programmeur aimerait repartir de zéro pour chaque nouveau langage à apprendre ?

De plus, certains symboles n'existent pas sur les claviers informatiques, ce qui impose donc quelques règles aux codeurs de tout pays.

Blocs	Un bloc est délimité par les caractères **{** et **}**. Cela signifie que les instructions entre les **accolades** font partie du même ensemble et ne sont pas dissociables. Les textes sont encadrés par des doubles quotes " qui est un symbole anglais (touche [3] Les conditions sont souvent mises entre **parenthèses**. Ex. **(**nom == "David"**)**
Commentaires	Un commentaire sur une ligne débutera par **//** Un bloc de commentaire commence par une ligne **/*** et termine par une ligne ***/** Cela permet de séparer les lignes d'instructions des lignes réservées aux humains.
Mots réservés	Les langages ont des mots réservés. Généralement ont trouve : - for, while, do, until, repeat - if, elseif, else - var, let, const
Signes et symboles	Le signe **=** signifie une affectation, tandis que **==** signifie une comparaison. Le signe ≠ n'existe pas, il est souvent remplacé par**!=** ou parfois par **<>**. Le signe ≥ n'existe pas, il est remplacé par **>=**. De même ≤ devient **<=**.

1.2 CONDITIONS

Les conditions permettent à un programme de faire un choix, de la même manière que notre esprit juge et décide à chaque instant de prendre des décisions pour notre vie.

1.2.1 CONDITIONS SIMPLES

Par exemple, s'il fait beau, j'irai à la piscine :

- Cas 1 : il fait beau ⇒ je vais à la piscine
- Cas 2 : il ne fait pas beau ⇒ *rien n'est précisé*

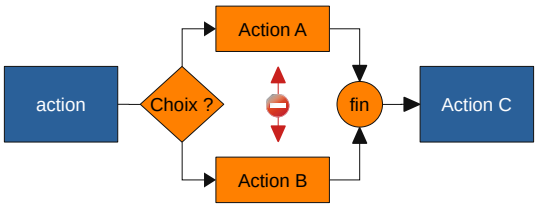
En programmation l'écriture sera similaire :

```
beau = ConsulterMeteo("Seyssins")
SI (beau == Vrai) ALORS
    Piscine = Vrai
SINON
FINSI
```

Comme vous pouvez le voir, l'ordinateur ne peut pas déduire seul que piscine est Faux, il faut donc l'écrire explicitement. C'est là où votre esprit a déduit seul que s'il ne fait pas beau, alors vous n'irez pas à la piscine.

Il faudra alors écrire à l'ordinateur, tous les comportements à avoir :

```
beau = ConsulterMeteo("Seyssins")
SI (beau == Vrai) ALORS
    Piscine = Vrai
SINON
    Piscine = Faux
FINSI
```



L'expression à l'intérieur de la parenthèse est vérifiée et son résultat est évalué à Vrai (tous les résultats sauf 0) ou Faux (zéro). C'est comme un drapeau que l'ordinateur lève si la condition se réalise. Les termes True et False remplacent Vrai et Faux, car l'informatique est généralement en anglais.

Notez la présence de l'affectation par l'instruction "ConsulterMeteo()" qui est un morceau de code qui sera toujours placé ailleurs. Pour en savoir plus, vous pouvez regarder la section sur Les fonctions personnelles.

1.2.2 CONDITIONS COMPLEXES

Dans la réalité, nous sommes capables de gérer plusieurs cas de figures, notamment, s'il y a plusieurs conditions.

Prenons l'exemple suivant :

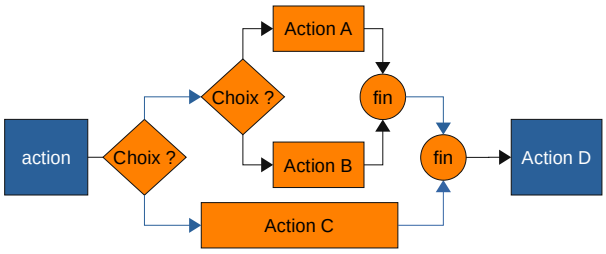
S'il fait beau et que j'ai assez d'argent, j'irai à la piscine.

Pour l'ordinateur, notre programme indique que Piscine sera toujours Faux, sauf lorsque Beau sera Vrai et que mon argent > prixTicket : le processeur évalue d'abord beau == vrai et lève ou non le drapeau ; il évalue ensuite argent > prixTicket et lève ou non le deuxième drapeau ; Enfin, le ET booléen permet de vérifier si les deux drapeaux sont levés.

```
SI ((beau == Vrai) ET (argent > prixTicket)) ALORS
    Piscine = Vrai
SINON
    Piscine = Faux
FINSI
```

Il est possible d'écrire les mêmes conditions, sous la forme d'une imbrication :

```
SI (beau == Vrai) ALORS
    SI (Argent > prixTicket)
          Piscine = Vrai
    SINON
          Piscine = Faux
    FINSI
SINON
    Piscine = Faux
FINSI
```



Le principe reste simple : si mon premier test échoue, pas la peine de vérifier le suivant. La condition en rouge ne peut être testé que s'il fait beau. C'est la notion d'imbrication. Dans la plupart des langages, on utilise un décalage pour montrer que le bloc d'instruction (en rouge) est à l'intérieur d'un autre bloc (ici la condition "SI (beau == Vrai) alors).

Remarquez aussi que nous utilisons un double signe = pour les tests, pour ne pas confondre avec une affectation :

```
toto = 12
SI (toto == 3)
    ecrire("toto est égal à trois")
FINSI
```

Enfin, dans l'exemple au-dessus, un texte encadré par " ou ' ou ` est considéré comme une chaîne de caractère. Le guillemet " est sur la touche 3, ' est sur la touche 4 et ` nécessite d'appuyer sur [Alt Gr] et [7].

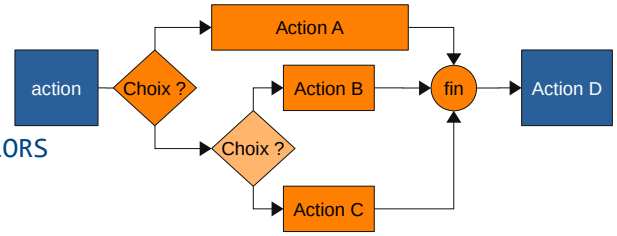
Mais nous sommes habitués à des sélections plus complexes :

S'il fait beau et que j'ai assez d'argent, j'irai à la piscine, sinon, si j'ai assez d'argent j'irai au cinéma, sinon je reste à la maison.

Pour notre ordinateur, nous pouvons fixer des valeurs initiales à Faux pour éviter trop de choix. Cela donne :

```
Piscine = Faux
Cinema = Faux
Maison = Faux

SI (beau == Vrai ET argent > prixTicket) ALORS
    Piscine = Vrai
SINON SI (argent > prixTicket)
    Cinema = Vrai
SINON
    Maison = Vrai
FINSI
```



En écrivant que Piscine, Cinema et Maison sont tous Faux, nos conditions n'ont qu'à inverser l'une de ces valeurs à Vrai (une seule bien sûr) .

Voici le tableau des possibilités :

	€	☼	Alors...
si...	FAUX	FAUX	
si...	FAUX	VRAI	
si...	VRAI	FAUX	
si...	VRAI	VRAI	→ ■

Les conditions mesurent des égalités :

- nombreA > nombre B, nombreA >= nombreB (signifie plus grand ou égal)
- nombreA == nombreB (on n'écrit pas = tout seul car cela signifie une affectation)

Lorsque l'égalité est vraie, on exécute la première partie du code.

Lorsque l'égalité est fausse, on exécute la partie de code après le premier SINON.

1.2.3 Essai 1 : Fusée Apollo

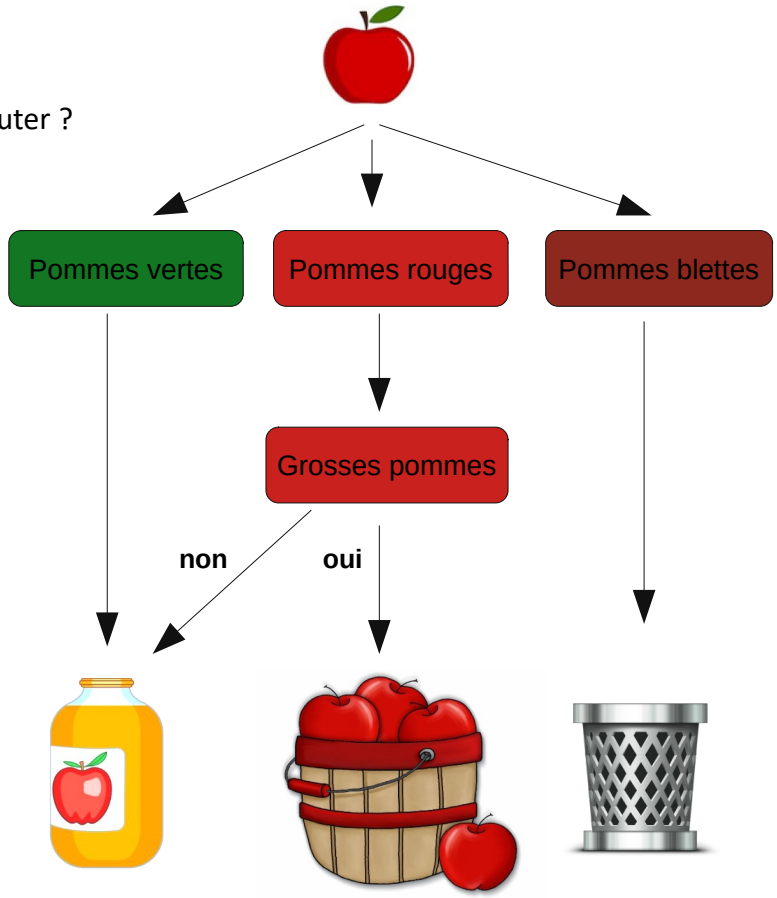
Sauriez-vous écrire le programme pour l'ordinateur de la condition suivante :

Si l'altitude est inférieure à 300 pieds et que le module Apollo est en descente, sortir le train d'atterrissage et allumer les moteurs.

Essai 2 : Tri de pommes

Une IA équipée d'une caméra voit passer des pommes et doit décider où envoyer la pomme :

Pouvez-vous écrire les conditions à exécuter ?



1.3 RÉPÉTITIONS

Les répétitions permettent de ne décrire qu'une seule fois les instructions et de les exécuter plusieurs fois. Pour éviter tout problème de boucle infinie, il faut simplement indiquer une ou plusieurs conditions de sortie.

Nous verrons qu'il y a 3 sortes de boucles :

- FOR (*condition*) FAIRE { actions }
- DO { actions } TANTQUE (*condition*)
- TANTQUE (*condition*) FAIRE { actions }

1.3.1 RÉPÉTITION SIMPLE PAR COMPTAGE

Les répétitions consistent à faire plusieurs fois les mêmes actions. À chaque boucle, nous parlerons d'itération : une boucle qui compte de 1 à 10 aura donc 10 itérations.

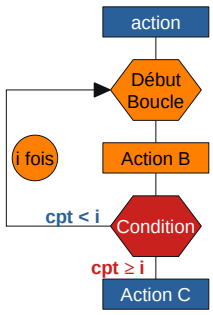
Comme il est très rare de faire des boucles à l'infinie, il y a généralement une condition de sortie.

Exemple : *je dois écrire dix fois "je ne bavarde pas en classe"* *(merci de m'envoyer un email si cela ne vous est jamais arrivé)*

Cette répétition se traduit en langage informatique :

```
FOR (i=1 ; i<11; i=i+1) {
    écrire "je ne bavarde pas en classe"
}


// écriture en pseudo-code
COMPTER à partir de 1 et jusqu'à 10
    écrire "je ne bavarde pas en classe"
FIN COMPTER
```



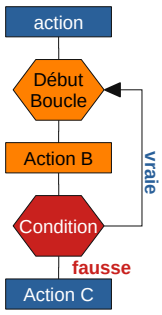
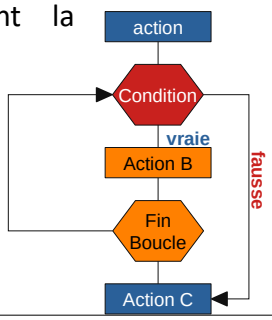
Dans ce cas, la condition de sortie est l'usage du compteur "jusqu'à 10" (la valeur du compteur doit être inférieure à 11).

1.3.2 Répétitions simples par condition

Les répétitions avec une condition de sortie sont fréquentes. Il y a deux boucles différentes :

- do {…} while (condition) que l'on peut traduire par "faire ceci tant que…"
- while (condition) {…} qui signifie "tant que … faire"

Il faut noter que la première solution effectuera les instructions entre les accolades au moins une fois, tandis que la seconde solution peut éviter les instructions si la condition est déjà remplie.

Do… while	While…
On place une instruction qui détermine le début de la boucle : **do** on rédige ensuite les instructions à l'intérieur d'un bloc : **{…}** Après le bloc, on indique la fin de la boucle et entre parenthèse, le critère de validation : **while (…)** 	On place immédiatement la boucle et sa condition : **while (…)** on rédige ensuite les instructions à l'intérieur d'un bloc : **{…}** Après le bloc, il n'y a rien à écrire. 

Dans les deux structures, tant que la condition est vraie, on recommence la boucle.

Seule la dernière boucle (*while () {…}*)permet de ne pas faire d'action si la condition est fausse dès le départ.

Les deux autres (*for () {…}* et *do {…} while ()*)exécuteront **au moins une fois** les actions avant de tester la condition ;

1.3.3 Répétitions imbriquées

C'est un cas particulier, car il est possible dans une première boucle, d'exécuter une autre boucle (et encore une autre, etc.). En revanche, les boucles ne doivent pas se croiser.

Bon	Mauvais

`COMPTER à partir de 1 et jusqu'à 10`
`  TANTQUE FinPage = 0`
`  écrire "je ne bavarde ..."`
`  FIN TANTQUE`
`FIN COMPTER`
	`COMPTER à partir de 1 et jusqu'à 10`
	`  TANTQUE FinPage = 0`
	`  écrire "je ne bavarde ..."`
	`  FIN COMPTER`
	`FIN TANTQUE`

1.3.4 EXERCICES : LES RÉPÉTITIONS

Essai N°1 :

Vous devez piloter un robot qui remplit une bouteille d'eau, sauf si elle est déjà pleine. Quelle boucle choisissez-vous ?

Essai N°2

Vous travaillez pour un imprimeur de cahier de texte. Celui-ci veut une table de multiplication de 1 à 12 sur chaque page (il y a 10 pages). Vous utiliserez la fonction changePage() après chaque table de multiplication. Sur chaque page, il faut écrire 'X fois Y égal Z' avec X qui est le numéro de la page (de 1 à 10), Y qui va de 1 à 12 et Z qui est le résultat de l'opération.

Créez l'algorithme.

Essai N°3

Dans les équipes du TGV, le service technique a déjà programmé trois fonctions : getSpeed(), setSpeed(vitesse) et setBrake(freinage). Les valeurs sont en kilomètre/heure.

- GetSpeed() donne la vitesse actuelle
- setSpeed() augmente la vitesse par pas de 10 km/h
- setBrake() diminue la vitesse par pas de 10 km/h

Les signaux automatiques sur le parcours donnent les vitesses autorisées.

Départ	Voie rapide	Zone travaux	Voie rapide	Arrivée
90 km/h	290 km/h	110 km/h	290 km/h	0 km/h

Créez le programme pour les 5 phases du trajet.

2 LES ALGORITHMES

Pour le moment, nous n'avons utilisé aucun langage de programmation : ni Python, ni JavaScript, ni Java, ni C#…

Pourtant, savoir programmer n'est pas une histoire de choix de langage mais d'expression d'idée.

La difficulté est de pouvoir exprimer en mots simples ce que doit faire le système à programmer pour obtenir le résultat voulu.

Exemple : peindre la salle informatique en bleu nuit et gris avec un liseré orange"

Vous avez tous en tête un résultat et l'idée de comment s'y prendre… ceci étant, auriez-vous coché toutes les actions ci-dessous ?

- Acheter les pots de peinture
  - Déterminer la superficie à couvrir
    - Prendre les mesures de la pièce
  - Trouver un magasin ayant les couleurs voulues
  - Commander la peinture
  - Attendre la livraison
- Acheter les pinceaux
  - Déterminer le nombre de peintre
  - Acheter 2 types de pinceaux pour chacun
    - Le rouleau
    - La queue de vache
- Acheter les protections
  - Protections du peintre
    - Prévoir lunettes
    - Prévoir blouse
    - Prévoir gants
  - Protections des sols et du mobilier
    - Bâches plastique
    - Prévoir superficie (prendre les mesures...)
- Préparer la salle
  - Si on a les protections du sol et du mobilier
  - ...

Eh oui, on n'a pas encore commencé à peindre !

L'esprit humain est tellement puissant qu'il intègre de nombreuses étapes comme participant à la fonction peindre la salle.

Cela signifie que la fonction peindre(salle) contient de nombreuses instructions qui sont masqués par le terme "peindre la salle".

Notez les options de peinture : bleu nuit, gris et liseré orange

Vous constatez déjà que le problème pour l'ordinateur, sera identique : il faut lui apprendre toutes les instructions pour effectuer une fonction.

2.1 LES FONCTIONS

Heureusement, la plupart des langages de programmation incluent des fonctions déjà préprogrammées pour afficher des informations, recevoir des données et pour effectuer des calculs plus complexes.

Les fonctions utilisent plusieurs instructions groupées ensembles. Cela évite au développeur de devoir écrire de longues suites d'instructions pour effectuer des actions fréquemment utilisées.

2.1.1 FONCTIONS INTERNES

Ce sont les fonctions intégrées au langage de programmation. L'une de ces fonctions les plus classiques est l'affichage d'un message :

```
écrire("Salut le monde !")
```

Le résultat sur un écran, serait

```
Salut le monde !
```

Cependant, la plupart des langages spécifient où le message sera écrit, car ce peut être un écran, une fenêtre, une imprimante, etc.

```
Ecran.écrire("Salut le monde !")
```

Cette syntaxe se comprend comme ceci : appliquer sur l'écran, la fonction écrire() dont le message transmis est la chaîne de caractères "Salut le monde !".

2.1.2 LES FONCTIONS PERSONNELLES

C'est ici que le programmeur exprime complètement sa créativité : **il peut inventer des fonctions.**

Utiliser le langage de programmation pour créer vos propres outils, décrire ce que vous voulez que l'ordinateur fasse pour vous ! C'est un peu comme si vous pouviez créer vos propres briques de Lego!

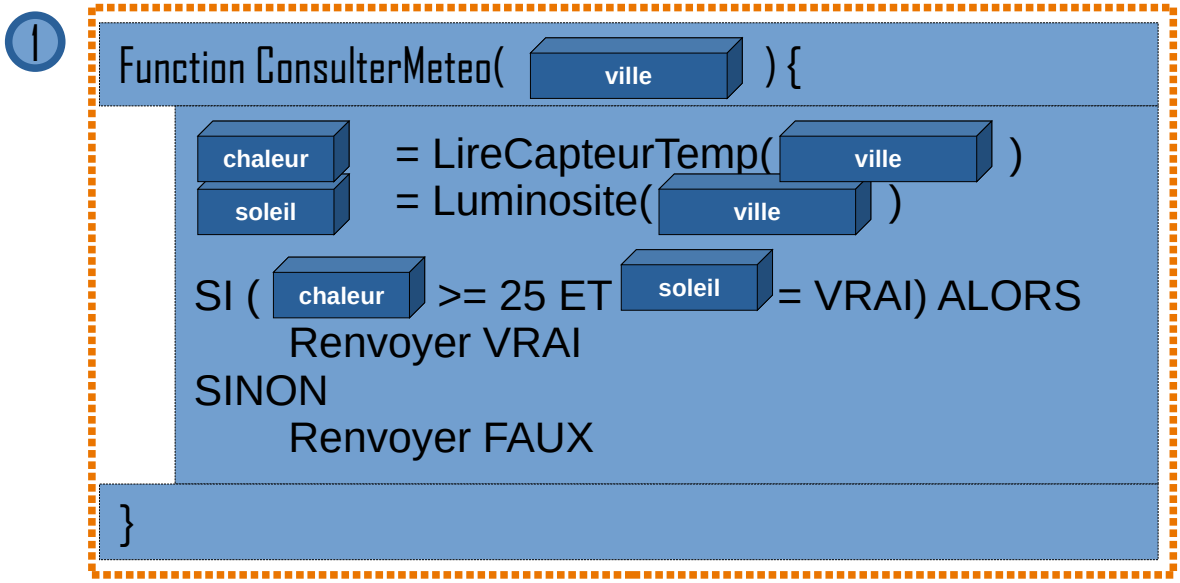
L'ordinateur fera des choses que les humains jugent pénibles, longues, répétitives, complexes, fastidieuses ou présentant un risque d'erreur.

2.1.2.1 STRUCTURE D'UNE FONCTION

Le développement d'une fonction se fait en deux étapes :

- **Création** de la fonction : il faut lui donner un nom, prévoir les paramètres qu'elle accepte et renvoyer (éventuellement) un résultat
- **Utilisation** de la fonction : une fonction est inerte et son code ne s'exécutera que lorsqu'on l'appelle.

Imaginons une fonction qui valide s'il fait beau dans une ville, en fonction de deux capteurs (température et luminosité) :



Le développeur indique qu'il va créer une fonction avec un mot-clé (ici, c'est 'function').

Il donne un nom à sa fonction ('ConsulterMeteo') et veut un paramètre : il s'agit d'une variable qui recevra une information donnée par l'utilisateur. Ici, la variable qui recevra un nom de ville, s'appelle 'ville'. Il ouvre le bloc de sa fonction.

Il utilise des fonctions existantes (on suppose ici que LireCapteurTemp() et Luminosite() sont fournies par le fabriquant du système). Il compare les résultats des deux fonctions avec ses critères (par exemple, en dessous de 25°C il ne fait pas beau, ou bien si le ciel est gris…)

Il renvoie un résultat, ce qui permet à l'utilisateur de la fonction d'obtenir ce résultat et de l'affecter à une variable ou directement dans une écriture.

Désormais, il est possible d'appeler la fonction ConsulterMeteo() : l'utilisateur doit indiquer une ville pour que la fonction s'exécute correctement. Le programme comprend alors qu'il doit retrouver l'endroit où est écrite la fonction, l'exécuter avec le paramètre placé dans sa variable et revenir à l'endroit où il était, avec le résultat de la fonction.

On peut appeler la fonction plusieurs fois :

2.1.2.2 EXEMPLE D'UNE FONCTION "CARRÉ"

Commençons par créer une fonction facile. Imaginez que vous deviez programmer la fonction suivante avec un langage qui ne sait pas ce qu'est le symbole ² (ou la fonction puissance) :

```
y = x²
```

La réponse est très simple…

Il suffit de multiplier x par x, ce qui s'écrit

```
y = x * x
```

Mais ceci est une opération, pas une fonction. Une fonction accepte un ou plusieurs arguments et renvoie un résultat entier. Ici, **x** est un argument et **y** également. En informatique, un argument est généralement une variable.

Voici l'algorithme de la fonction :

```
fonction auCarré(accepte un entier x) : un entier
{
    renvoie x * x
}
```

Son nom

Ce que la fonction attend.

Ce que la fonction va renvoyer.

Dans un programme, cette fonction s'emploie comme ceci :

```
Ecran.écrire("Choisissez un nombre à élever au carré : ")
reponse_y = Clavier.saisir()
resultat = auCarré(reponse_y)
Ecran.écrire(reponse_y + "élevé au carré = " + resultat)
```

et le résultat serait le suivant :

```
Choisissez un nombre à élever au carré : 13
13 élevé au carré = 169
```

Votre fonction auCarré(x) est réutilisable plusieurs fois : elle attend un nombre entier et renvoie un nombre entier. Ce qui se passe à l'intérieur ne concerne que le programmeur (vous !), et le programme complet sera écrit comme suit :

```
// partie de la création de la fonction
fonction auCarré(accepte un entier x) : un entier
{
    renvoie x * x
}

// partie du programme utilisant la fonction
Ecran.écrire("Choisissez un nombre à élever au carré : ")
reponse_y = Clavier.saisir()
resultat = auCarré(reponse_y)
Ecran.écrire(reponse_y + "élevé au carré = " + resultat)
```

Voici donc un premier aperçu de ce que vous pouvez apprendre à l'ordinateur.

2.1.3 UTILITÉ DES FONCTIONS

Le principe de codage se résume à ce mot :

K I S S

Il ne s'agit pas du groupe pop/rock (dont le titre "I Was Made for Lovin' You" sortit en 1979 est le plus connu) ou du mot anglais qui signifie "embrasser, bise" mais d'un acronyme qui signifie :

Keep It Simple Stupid

Il faut conserver le code le plus simple et "stupide" possible.

Le moyen le plus simple d'atteindre cet objectif, est d'**éviter de recopier le même code à plusieurs endroits du programme**, car il y aurait alors deux risques :

1. Le premier est de copier un code faux (mal testé, ou qui contient encore des bogues). Dans ce cas, il faut procéder à plusieurs corrections dans plusieurs endroits du code et il serait facile de faire un oubli.
2. Le deuxième risque est de rendre le code difficilement partageable : une fonction peut être appelée pour effectuer une action particulière et un autre codeur peut l'utiliser sans connaître son contenu. Il suffit en effet, que le codeur original fournisse la description de ce qu'il faut donner en paramètre, et ce que la fonction fait ; peu importe de comment elle le fait.

La sécurité du code passe donc par les fonctions, c'est un point incontournable.

2.1.4 Fonctions et paramètres

Une fonction sans paramètre reste relativement peu utile : elle est rigide, ne peut pas renvoyer de résultats différents et finalement, complique un peu le code.

Il est donc très fréquent de passer des informations à une fonction, pour qu'elle puisse faire un traitement différent selon le ou les paramètres.

2.1.4.1 Paramètre en entrée

Imaginons que notre programme prenne la décision de laisser une personne conduire un véhicule si elle est majeure. Le constructeur automobile peut récupérer l'année de naissance et doit valider si oui ou non, la personne peut prendre le volant

Exemple : estMajeur(anneeNaissance)

```
fonction estMajeur(anneeNaissance: entier)
{
    si (anneeCourante-anneeNaissance > 17) {
        afficher("ok") ;
    } sinon {
        afficher("non") ;
    }
}
```

Cette fonction accepte un seul paramètre, un nombre entier. Il est possible d'appeler cette fonction plusieurs fois, avec des nombres différents :

```
estMajeur(0);
estMajeur(17);
estMajeur(18);
estMajeur(45);
estMajeur(99);
```

Le résultat affiché serait le suivant :

```
non
non
ok
ok
ok
```

Il y a une économie forte du nombre de lignes dans le code puisque chaque appel à la fonction remplace 5 lignes de codes (de l'instruction 'si' à la fin).

Maintenant imaginons que le constructeur vende ses voitures en France et aux États-Unis. La majorité est différente dans les deux pays, respectivement 18 et 21 ans. Comment gérer cette situation ?

Il suffit d'ajouter un paramètre à la fonction pour prendre en compte le nouveau problème.

Exemple : estMajeur(anneeNaissance, pays)

```
fonction estMajeur(anneeNaissance: entier, pays: texte)
{
    si (pays == "France") {
          majeur = 18;
    } sinon {
          majeur = 21 ans;
    }
    si (anneeCourante-anneeNaissance > majeur) {
        afficher("ok") ;
    } sinon {
        afficher("non") ;
    }
}
```

Il est ainsi possible d'avoir le nombre de paramètre nécessaire sans difficultés. En revanche, la table des possibilités s'étendra sur autant de dimensions.

```
EstMajeur(0, "France");
estMajeur(17, "France");
estMajeur(18, "France");
estMajeur(18, "US");
estMajeur(21, "États-Unis");
estMajeur(45, "Amérique");
estMajeur(99, "France");
```

Le résultat affiché serait le suivant :

```
non
non
ok
non
ok
ok
ok
```

Nous aurons en fait la matrice suivante :

	17	18	20	21	30
France					
Autres pays*					

Notez que l'algorithmique est la science qui consiste à passer de ce tableau, à l'algorithme plus haut : ici, le raisonnement est inversé pour montrer l'intérêt des fonctions.

2.1.4.2 PARAMÈTRE EN SORTIE

Il n'y a qu'une seule possibilité en sortie, un seul paramètre. Il faut imaginer la fonction comme donnant un seul résultat : cela peut-être une valeur booléenne (vraie ou fausse), une valeur numérique ou une chaîne de caractère.

Par contre, cela permet d'écrire :

variable = fonction(x, y, z) ;

Pour reprendre l'exemple de la majorité précédente, le constructeur automobile ne souhaite pas un affichage, il veut seulement savoir si le conducteur est majeur, oui ou non. Nous pouvons lui renvoyer une valeur booléenne, et voici comment :

Exemple : estMajeur(anneeNaissance)

```
fonction estMajeur(anneeNaissance: entier) : booléen
{
    si (anneeCourante-anneeNaissance > 17) {
        retourner VRAI ;
    } sinon {
        retourner FAUX ;
    }
}
```

Il reste possible d'appeler la fonction comme précédemment mais rien ne s'affichera :

```
estMajeur(0) ;
```

Pour obtenir un résultat afficher, on peut demander d'afficher le résultat que renvoit la fonction ;

```
affiche(estMajeur(0)) ;
```

```
non
```

Mais il est plus probable que le fabricant utilisera la fonction dans un test :

```
si (estMajeur(0) == VRAI) {
    accepterDemarrage();
} sinon {
    bloquerDemarrage();
}
```

Il est également possible de garder la valeur de retour dans une variable :

```
peutDemarrer = estMajeur(0);
```

Comme vous pouvez le voir, la fonction peut donner un résultat qui peut être utilisé par d'autres fonctions.

2.1.5 FONCTION : EN RÉSUMÉ

Les fonctions sont très importantes pour éviter les erreurs et les codes trop longs.

On peut voir une fonction comme un bloc de code dont on ne connaît pas la composition mais qui accepte entre zéro et n paramètres et va renvoyer un seul résultat.

Une fonction qui renvoie la valeur maximum de A et de B ne renvoie bien qu'une seule valeur (A ou B). Pour renvoyer une valeur, il ne faut pas oublier d'écrire l'instruction "retourner" suivi de la valeur à retourner.

La structure d'une fonction est la suivante :

```
Fonction MaFonction(param1, param2…) : typeRetour {
     bloc d'instructions…
     retourner resultat ;
}
```

Les fonctions peuvent appeler d'autres fonctions en paramètre.

Maximum = max(4, max(8, 1))

2.1.6 EXERCICES SUR LES FONCTIONS

Rédiger les fonctions suivantes...

1. fonction max(A, B)
2. fonction moyenne(A, B)
3. fonction calculTTC(PrixHT, tva)
4. fonction PrixRemise(Prix, pourcentageRemise)
5. calculer le prix TTC d'un article dont le prix est remisé de 5 %
6. fonction nomPermis(typePermis) → renvoie "Voiture" pour un type B, moto pour un type A, etc. (regardez vos permis de conduire)
7. fonction peutConduireVoiture(ageLegal, ageConducteur, nomPermis(type)) → renvoie VRAI si l'age du conducteur est supérieur à l'age légal et si le type B est voiture.
8. Fonction maxi(A, B, C) → en utilisant la fonction max(A, B)
9. Fonction convertirTemp(A) → renvoie la température en Kelvin de A qui est en degré.