Assimiler

Programmation et Algorithmie

Rédigé par

David ROUMANET
Professeur BTS SIO

Changement

Date	Révision

Sommaire

A Les symboles et équivalences en JavaScript....1
A.1 Affectations de variables....1
A.2 Conditions....1
A.2.1 Conditions simples....1
A.2.2 Conditions complexes....2
A.3 Itérations....3
A.3.1 Répétition simple par comptage....3
A.3.2 Répétitions imbriquées....4
A.3.3 Exercices : les répétitions....5
B Les algorithmes....6
B.1 Les fonctions....8
B.1.1 Fonctions internes....8
B.1.2 Les fonctions personnelles....8
B.1.2.a Structure d'une fonction....8
B.1.2.b exemple d'une fonction "carré"....10
B.1.3 Utilité des fonctions....11
B.1.4 Fonctions et paramètres....12
B.1.4.a Paramètre en entrée....12
B.1.4.b Paramètre en sortie....14
B.1.5 Fonction : en résumé....15
B.1.6 Exercices sur les fonctions....17

Nomenclature :

- **Assimiler** : cours pur. Explication théorique et détaillée (globalement supérieur à 4 pages).
- **Décoder** : fiche de cours, généralement inférieure à 5 pages.
- **Découvrir** : Travaux dirigés. Faisable sans matériel.
- **Explorer** : Travaux pratiques. Nécessite du matériel ou des logiciels.
- **Mission** : Projet encadré ou partie d'un projet.
- **Voyager** : Projet en autonomie totale. Environnement ouvert : Vous êtes le capitaine !

A Les symboles et équivalences en JavaScript

L'algorithmie est une manière de décrire un programme, indépendante du langage qui sera réellement utilisé. Apprendre l'écriture algorithmique, c'est faciliter l'apprentissage de la programmation.

A.1 Affectations de variables

La première représentation est d'utiliser des variables pour mémoriser des nombres, des caractères, des tableaux ou même des objets.

<table>
<tr><th>Algorithmie</th><th>Javascript</th></tr>
<tr><td><pre>DECLARE
age ← 25
an_actuelle ← 2023
nom ← "DUPOND"
liste_temp ← tableau[1 à n]
an_naissance ← an_actuelle - age
pi ← 3.14159</pre></td><td><pre>// Déclaration de variables
let age = 25
let an_actuelle = 2023
let nom = "DUPOND"
let list_temp = []
let an_naissance = an_actuelle - age
const pi = 3.14159</pre></td></tr>
</table>

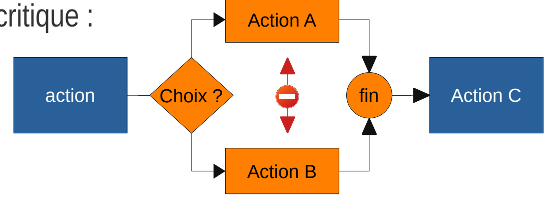
A.2 Conditions

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.

A.2.1 Conditions simples

Par exemple, dans un jeu de rôle, un lancer = 20 est un coup critique :

- Cas 1 : le Dé 20 = 20 ⇒ coup critique réussi
- Cas 2 : le Dé 20 différent ⇒ *rien n'est précisé*



<table>
<tr><th>Algorithmie</th><th>Javascript</th></tr>
<tr><td><pre>si d20 = 20
alors coup_critique = vrai
fin si</pre></td><td><pre>if (d20 == 20) {
    coup_critique = true
}</pre></td></tr>
</table>

ou

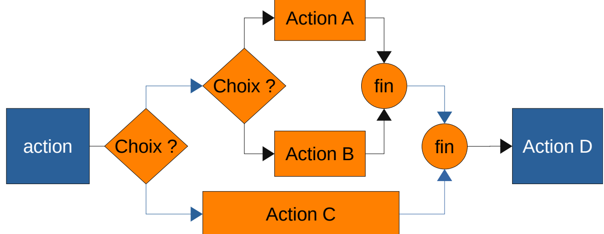
<table>
<tr><th>Algorithmie</th><th>Javascript</th></tr>
<tr><td><pre>si d20 = 20
    alors coup_critique = vrai
sinon
    coup_critique = faux
fin si</pre></td><td><pre>if (d20 == 20) {
    coup_critique = true
} else {
    coup_critique = false
}</pre></td></tr>
</table>

⚠️ notez le double égal en JavaScript : un simple égal signifiant l'affectation d'une valeur.

A.2.2 Conditions complexes

Les programmes peuvent gérer de nombreuses conditions, il existe le "si… sinon si… sinon si… sinon…"

Le principe reste simple :



Tester plusieurs possibilités, comme un intervalle et indiquer une action selon la valeur.

Par exemple, un programme qui donne un commentaire en fonction du pourcentage de réussite : on va tester les valeurs entre 0 % et 25 % puis 26 % et 50 % puis 51 % à 75 % et enfin, supérieures à 75 %

Algorithmie	Javascript

reussite ← resultat_du_test()
 	
si reussite < 26
    alors afficher("Très insuffisant")
sinon si reussite > 25 et reussite <= 50
    afficher("Résultat insuffisant")
sinon si reussite > 50 et reussite <= 75
    afficher("Résultat correct")
sinon
    afficher("Résultat excellent")
fin si
	let reussite = resultat_du_test()
	
	if (reussite < 26) {
	    console.log("Très insuffisant")
	} else if (reussite > 25 && reussite <= 50) {
	    console.log("Résultat insuffisant")
	} else if (reussite > 50 && reussite <= 75) {
	    console.log("Résultat correct")
	} else {
	    console.log("Résultat excellent")
	}

⚠️ il faut noter que le ET devient && (le OU devient ||)

L'instruction `afficher()` peut aussi faire référence à un affichage sur la page HTML du script, dans ce cas, ce sera une instruction comme document.write(), document.getQuerySelector(x).innerHTML(), etc.

💀 Le risque dans un algorithme, est d'oublier une condition : il est recommandé d'utiliser un tableau pour vérifier que toutes les conditions sont prises en compte. C'est notamment le cas, lorsqu'on doit vérifier deux variables dans plusieurs conditions

		A	
		FAUX	VRAI
B	FAUX	❌	✅
	VRAI	✅	❌

A.3 Itérations

Les itérations (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 en JavaScript :

- POUR (*condition*) FAIRE { actions }
- FAIRE { actions } TANTQUE (*condition*)
- TANTQUE (*condition*) FAIRE { actions }

A.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)*

comptage

Algorithmie	Javascript

`pour compteur allant de 1 à 10 faire`
 	
`    afficher(compteur, "je dois écrire...")`
`fin pour`
	`for (let compteur = 1 ; compteur < 11 ; compteur++)`
	`{`
	`    console.log(compteur, "je dois écrire...")`
	`}`

Faire… tant que…

Algorithmie	Javascript

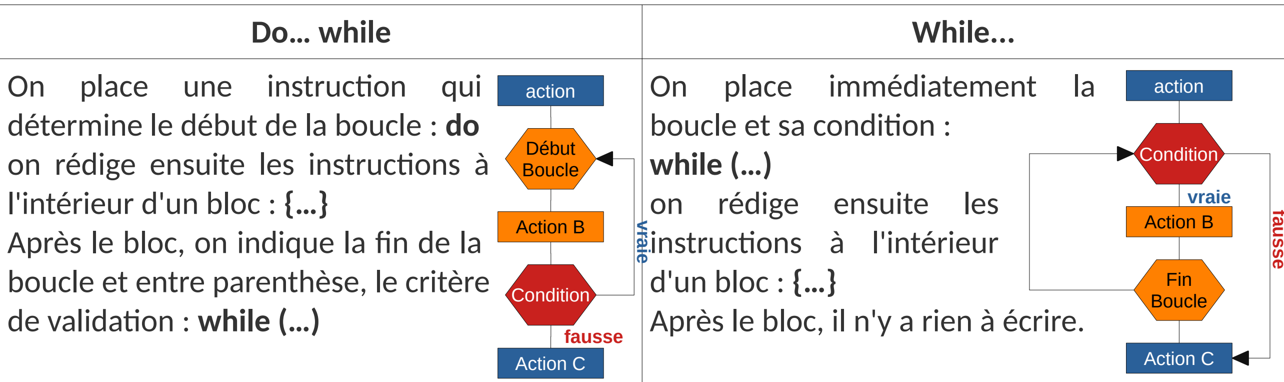
`compteur ← 1`
`faire :`
`    afficher(compteur, "je dois écrire…")`
`    compteur ← compteur + 1`
`tant que (compteur < 11)`
	`let compteur = 1`
	`do {`
	`    console.log(compteur, "je dois écrire…")`
	`    compteur = compteur + 1   // ou compteur++`
	`} while (compteur < 11)`

Tant que… faire...

Algorithmie	Javascript

`compteur ← 1`
`tant que (compteur < 11) faire`
`    afficher(compteur, "je dois écrire…")`
`    compteur ← compteur + 1`
`fin tant que`
	`let compteur = 1`
	`while (compteur < 11) {`
	`    console.log(compteur, "je dois écrire…")`
	`    compteur = compteur + 1   // ou compteur++`
	`}`

La différence entre "faire … tant que" et "tant que… faire" est que dans le premier cas, le test est fait à la fin des actions, tandis que dans le second cas, il est fait avant d'exécuter les actions.



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 ;

A.3.2 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

`POUR livre de 1 et jusqu'à 10`
`   TANTQUE FinPage = Faux`
`        Afficher(page)`
`        page <- page + 1`
`   FIN TANTQUE`
`FIN POUR`
	`POUR livre de 1 et jusqu'à 10`
	`   TANTQUE FinPage = Faux`
	`        Afficher(page)`
	`        page <- page + 1`
	`   FIN POUR`
	`FIN TANTQUE`

A.3.3 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.

B 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 masque 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.

B.1 Les fonctions

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.

B.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 :

```
Afficher("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 !".

B.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!

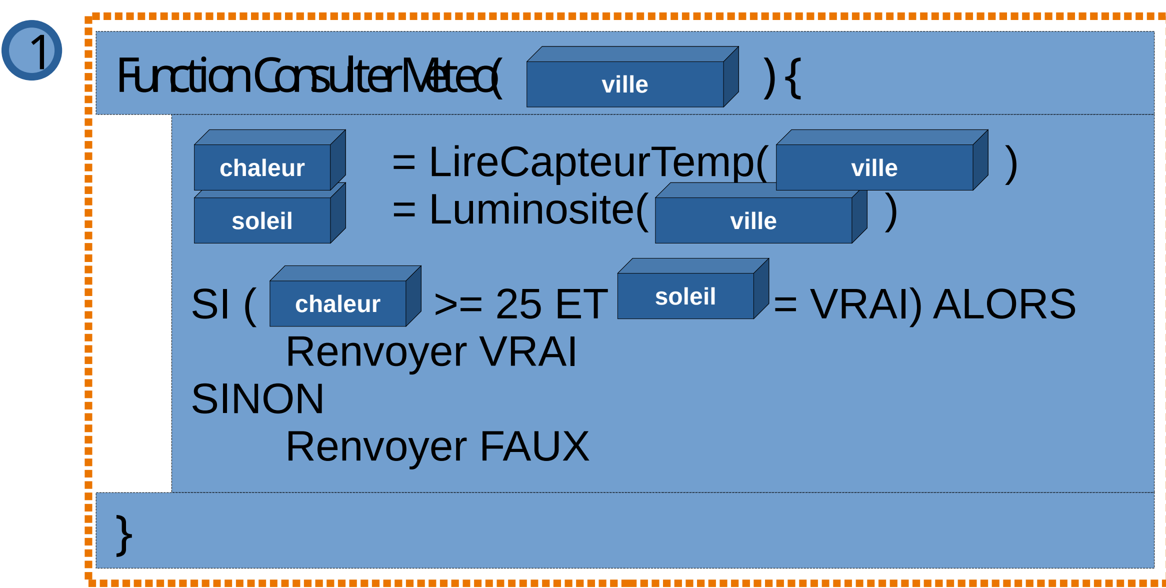
L'ordinateur fera des choses que les humains jugent pénibles, longues, répétitives, complexes, fastidieuses ou présentant un risque d'erreur.

B.1.2.a 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').

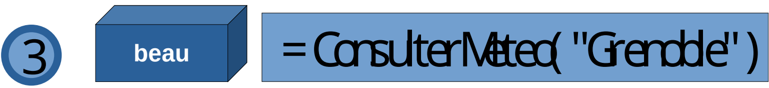
Il donne un nom à sa fonction ('ConsulterMeteo') et veut un paramètre : il s'agit d'une variable qui recevra une information donnée par l'utilisateur. Ici, la variable qui recevra un nom de ville, s'appelle 'ville'. Il ouvre le bloc de sa fonction.

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 :



B.1.2.b 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 :

Son nom

Ce que la fonction attend.

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

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.

B.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 **I**t **S**imple **S**tupid

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.

B.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.

B.1.4.a 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.

B.1.4.b 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.

B.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))

B.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.