

Introduction à l'informatique

Pseudo-code

Benjamin Monmege

2019/2020


1 Comment (d)écrire des algorithmes ?

Maintenant que l'on sait ce que sont les algorithmes, il nous reste à savoir comment les décrire, tant à une personne qu'à un ordinateur. La méthode que nous avons utilisée plus haut pour la méthode de dénombrement distribuée consiste à une suite de phrases en langue naturelle. C'est proche de la façon dont nous écrirons des algorithmes tout au long de ce cours : on appelle cette description un *pseudo-code*. C'est tout à fait suffisant lorsqu'il s'agit de communiquer un algorithme à une autre personne. Cependant, cela ne suffit pas s'il faut le faire comprendre à une machine. Pour cela, on utilise des langages de programmation, tels que Python (que vous utiliserez dans l'UE *Mise en œuvre informatique*) ou Java (que vous utiliserez dans l'UE *Programmation 1*). Il existe aussi des langages de description graphique par blocs, tels que Scratch, permettant de manipuler le code à l'aide de blocs aimantés.

Le pseudo-code que nous allons utiliser dans la suite a une syntaxe proche de celle de Python, mais nous nous autoriserons des raccourcis et une syntaxe un peu moins rigide. Cependant, comme en Python, nous utiliserons des variables, chaque instruction élémentaire sera écrite sur une ligne séparée, le code sera indenté et nous découperons notre code en fonctions. Contrairement à Python, le pseudo-code de ce cours sera rédigé en français (et non en anglais). Les ingrédients principaux seront cependant les mêmes. Commençons par décrire dans ce chapitre les *structures de contrôle* permettant de structurer le pseudo-code.

1.1 Structures de contrôle : une introduction en Scratch

Nous allons décrire cinq types de structures de contrôle : itérations, fonctions, conditionnelles, écriture/lecture et variables. Pour motiver ce choix d'ingrédients de base, illustrons leur utilisation sur un petit exemple en Scratch : vous pouvez écrire et exécuter le code au fur et à mesure en utilisant l'éditeur Scratch en ligne, disponible dans l'onglet *Créer* du site <https://scratch.mit.edu>, et en chargeant le fichier donné en ligne sur le cours Ametice). L'objectif est de faire sortir un petit chat d'un labyrinthe très simple représenté en FIGURE 1, c'est-à-dire le faire atteindre la cible jaune en haut à droite du chemin blanc.

Après quelques essais, on trouve aisément une solution au problème, où la première ligne permet de dire à Scratch qu'on exécute le programme dès le début de l'exécution (c'est-à-dire quand l'utilisateur appuie sur le bouton  dans l'interface) :

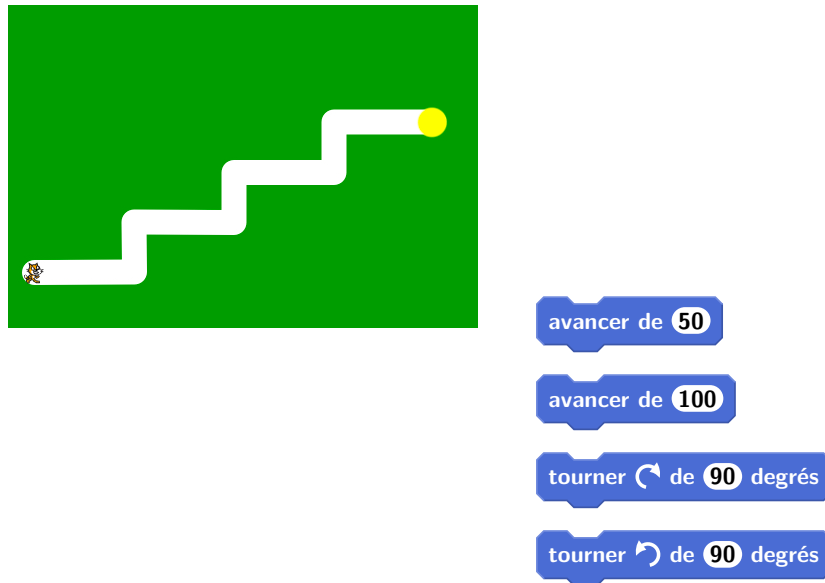


FIGURE 1 – Le labyrinthe d'où nous devons faire sortir le chat et les opérations élémentaires qu'on s'autorise



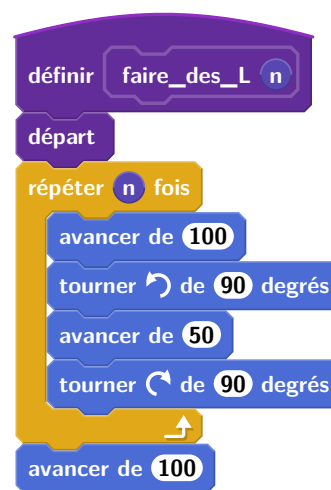
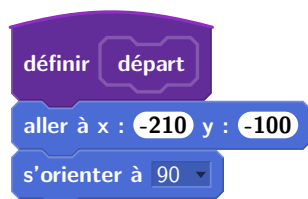
Ce code n'est que moyennement lisible et réutilise trois fois la même séquence d'opérations. On peut le simplifier grandement en utilisant une boucle permettant de répéter un certain nombre de fois la même séquence d'opérations :



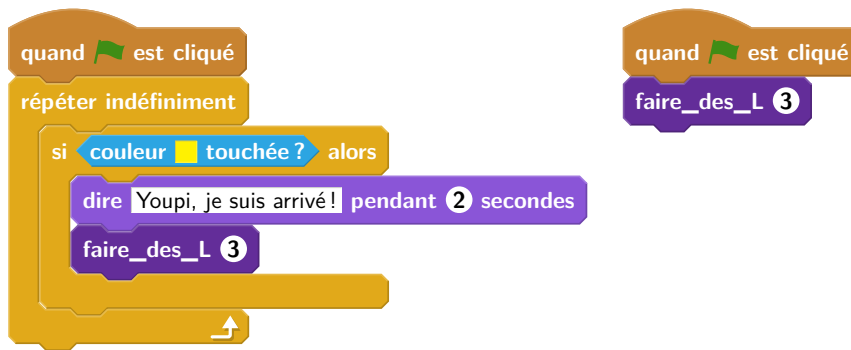
Comment faire s'arrêter l'algorithme lorsqu'on atteint effectivement la cible de couleur jaune ? En Scratch, on peut décrire un autre bloc, indépendant du premier, qui exécutera en boucle un test, qu'on décrit à l'aide d'un bloc conditionnel :



Si, au contraire, on souhaite faire recommencer l'animation, il nous faut ré-exécuter le code précédent, plutôt que d'écrire le bloc d'arrêt. Pour éviter la recopie de code, on peut utiliser une fonction permettant de sauver un morceau de code qu'on peut ensuite réutiliser autant de fois que nécessaire en employant son nom. Ici, on a besoin de deux fonctions : une fonction qui remet le petit chat à sa position de départ (là encore, il faut quelques essais avant d'y parvenir...) et une fonction qui exécute la boucle précédente, en généralisant le nombre de « L » qu'il exécute en cas de modifications dans le futur.



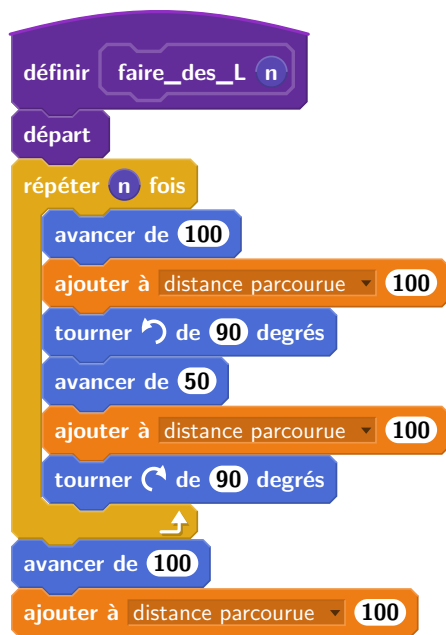
Le code principal se raccourcit alors beaucoup. Si de plus on fait *dire* au chat une petite phrase une fois qu'il a atteint la cible avant de repartir, le code devient



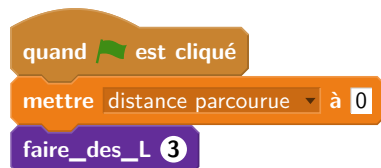
Il se peut que l'utilisateur ne veuille pas que ce code tourne indéfiniment, et donc avoir une possibilité de l'arrêter. Par exemple, on pourrait demander à l'utilisateur son avis, puis attendre sa réponse pour prendre une décision. En utilisant une conditionnelle supplémentaire pour tester sa réponse (qui est automatiquement stocké dans un bloc **réponse**) et réagir différemment si l'utilisateur souhaite continuer ou non, le code devient



Finalement, équipons le chat d'un podomètre, comptant le nombre de pas qu'il a effectué au total depuis le début de l'exécution du programme. Pour ce faire, il nous faut stocker ce nombre de pas dans ce qu'on appelle une *variable* : nommons-là « distance parcourue » pour clarifier sa signification. Une fois la variable créée, on peut la modifier et ajouter à son contenu une valeur entière, par exemple. Cela permet donc de modifier la fonction « faire_des_L » pour qu'elle enregistre dans la variable les modifications :



On n'oublie pas d'ajouter la remise à zéro de la variable dans le code principal :



Maintenant que nous avons vu l'utilité des différentes structures de contrôle en Scratch, entrons dans le détail pour indiquer comment les décrire en pseudo-code.

1.2 Variables

L'utilisation de variables permet l'écriture de programmes stockant des données. Attention, le mot *variable* a deux sens, selon qu'on l'utilise dans son acception mathématique ou informatique :

- En mathématiques, une variable est une grandeur dont la valeur est (provisoirement) indéterminée, sur laquelle on effectue une combinaison d'opérations avec des constantes et d'autres variables. Par exemple, on peut considérer la variable x dans l'équation $x^2 - 3x + 2 = 0$. Elle n'a pas de valeurs, mais pourra en avoir une ou plusieurs une fois l'équation résolue : en l'occurrence, l'équation à deux solutions $x = 1$ ou $x = 2$.
- En informatique, une variable est un identifiant désignant un emplacement de la mémoire et son contenu peut donc évoluer au cours du temps. Si une variable n'est pas initialisée, sa valeur est temporairement non définie. Par exemple, on dénote par

$$x := 1$$

l'affectation de la valeur 1 dans la variable x .

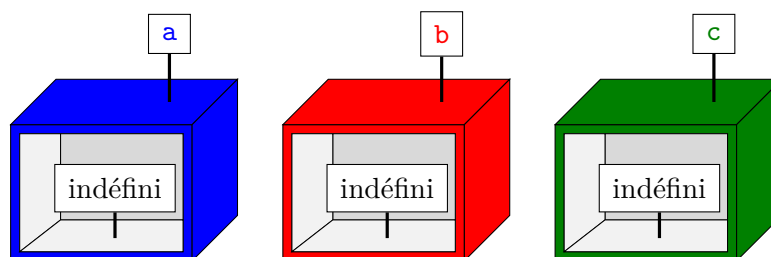
Considérons ainsi un programme qui enchaîne plusieurs affectations sur trois variables a , b et c :

```

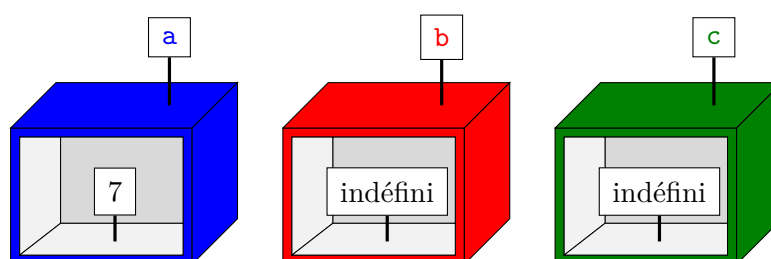
a := 7
b := 3
c := b - a
a := a - c

```

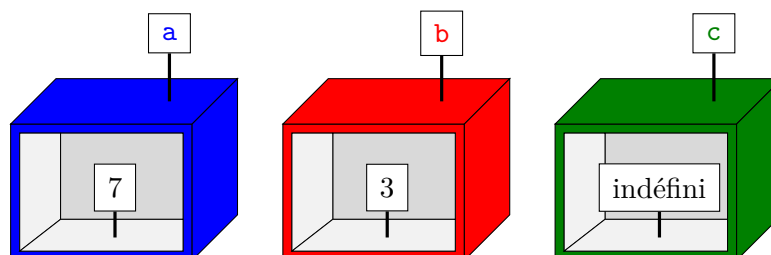
Au début de l'exécution du programme, le contenu de chaque variable est indéfini :



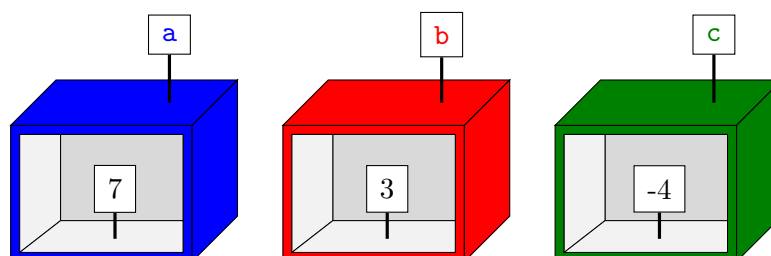
La première instruction $a := 7$ s'exécute, modifiant le contenu de la variable a :



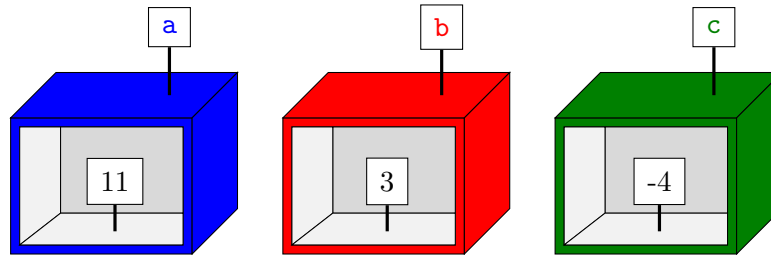
La seconde instruction $b := 3$ s'exécute de même :



La troisième instruction $c := b - a$ s'exécute alors en deux temps : d'abord les valeurs des variables b et a sont extraites, puis on effectue l'opération de soustraction avant de modifier le contenu de la variable c :



Finalement, la dernière instruction $a := a - c$ commence par extraire les valeurs de a et c , calcule leur différence puis modifie le contenu de la variable a :



Notez que cela n'a donc rien à voir avec la résolution de l'équation mathématique

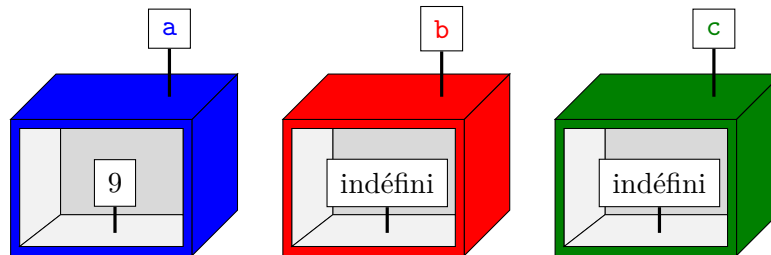
$$a = a - c$$

qui se résoudrait en $c = 0$...

Considérons un deuxième exemple de code :

```
a := 9
c := a + b
b := c
c := 2
```

Une fois la première ligne exécutée, on est dans la situation suivante :



Lorsqu'on essaie d'exécuter la seconde instruction `c := a + b`, on extrait les valeurs des variables `a` et `b` : on échoue alors puisque la variable `b` est indéfinie pour l'instant. Ce code n'est donc pas valide.

1.3 Fonctions

Considérons un autre exemple nécessitant l'usage de variables. On se donne ainsi les coordonnées GPS d'un restaurant et d'un client (en train de faire sa recherche Google Maps pour trouver un restaurant à Marseille...). Pour simplifier, supposons ici que ces coordonnées GPS sont données par une abscisse et une ordonnée dans un repère bidimensionnel orthonormé. On note ainsi `x_restaurant` et `y_restaurant` les coordonnées du restaurant, `x_client` et `y_client` celles du client. En se rappelant que la distance entre un point de coordonnées (x_1, y_1) et un point de coordonnées (x_2, y_2) est donné par la formule

$$d((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

on peut écrire le code suivant pour obtenir la distance entre le restaurant et le client :

```
dx := x_restaurant - x_client
dy := y_restaurant - y_client
distance_carrée := dx×dx + dy×dy
distance := √distance_carrée
```

S'il y a 100 restaurants dont on veut connaître la distance au client, il faut donc répéter 100 fois ces mêmes quatre lignes, en modifiant les coordonnées du restaurant. Ce serait bien répétitif, source d'erreurs et difficilement maintenable si on décide désormais de changer de représentation pour les coordonnées GPS. À la place, il vaut mieux utiliser une fonction.

Attention, comme pour le mot variable, le mot *fonction* a deux sens selon qu'on l'utilise dans son acception mathématique ou informatique :

- En mathématiques, une fonction est une relation entre un ensemble d'entrées et un ensemble de sorties avec la propriété que chaque entrée est reliée à *au plus une sortie*. La fonction peut souvent être décrite par une expression utilisant des variables représentant les entrées. Par exemple, on peut considérer la fonction f qui à un entier x associe $f(x) = x^2 + 2x$.
- En informatique, une fonction est une portion de code représentant un sous-programme, qui effectue une tâche ou un calcul relativement indépendant du reste du programme. Une fonction peut avoir des arguments représentés par des variables que le code de la fonction peut utiliser, et peut renvoyer un résultat.

Ainsi, on peut écrire une fonction qui calcule la distance entre un restaurant et un client :

- il prend en entrée les coordonnées du restaurant et du client
- et produit en sortie la distance attendue.

Dans le pseudo-code utilisé dans ce cours, on déclarera une fonction de la façon suivante :

```
fonction calcule_distance(x_restaurant, y_restaurant, x_client, y_client):  
    dx := x_restaurant - x_client  
    dy := y_restaurant - y_client  
    distance_carrée := dx×dx + dy×dy  
    distance := √distance_carrée  
    retourner(distance)
```

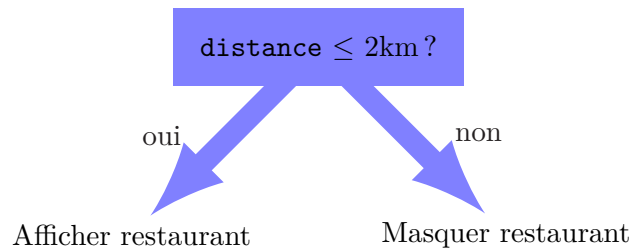
Le mot clé **fonction** est suivi du nom de la fonction qu'on définit, ainsi que ses arguments en parenthèses : ce sont des variables qu'on peut ensuite utiliser dans le corps de la fonction. On utilise le mot clé **retourner** pour renvoyer le résultat attendu. Cela arrête l'exécution de la fonction : on ne peut donc retourner qu'au plus une fois par fonction.

On peut alors calculer la distance entre plusieurs restaurants et plusieurs clients en appelant la fonction préalablement définie :

```
distance1 := calcule_distance (x_restaurant1, y_restaurant1,  
                               x_client, y_client)  
distance2 := calcule_distance (x_restaurant2, y_restaurant2,  
                               x_client2, y_client)  
distance3 := calcule_distance (5.12, 145.1, 5.45, 148.3)
```

1.4 Conditionnelles

Comment faire exécuter deux choses différentes à notre code selon qu'un restaurant est à moins de deux kilomètres d'un client ou pas ?



Il nous faut écrire une condition dans le pseudo-code, suivie de deux possibilités suivant que la condition est satisfaite ou non :

```

Si (distance ≤ 2) alors
    afficher restaurant...
Sinon
    masquer restaurant...
FinSi
  
```

Quelles sont les conditions que l'on peut tester ?

— Comparaisons : on peut comparer une variable avec une valeur ou une autre variables

$\text{distance} \leq 2$ $\text{distance} > 3$ $x \neq y$

— Divisibilité : on peut tester si le reste dans la division euclidienne d'un entier n par un entier m vaut k à l'aide de l'expression

$$n \equiv k \pmod{m}$$

En particulier, on peut tester la parité de l'entier n grâce au code

$$n \equiv 0 \pmod{2}$$

— Combinaison de tests : on peut combiner les tests avec les opérateurs **et**, **ou** et **non**. Si on cherche à n'afficher que les restaurants proches et ayant au moins 3 étoiles, on exécute

$(\text{distance} \leq 2) \text{ et } (\text{nombre_étoiles} \geq 3)$

Si on préfère ne voir que les restaurants qui sont très proches ou alors qui peuvent être plus éloignés mais ont au moins 4 étoiles, on utilisera plutôt

$(\text{distance} < 1) \text{ ou } (\text{nombre_étoiles} \geq 4)$

Au passage, notons l'utilisation du mot-clé **mod** qui permet de calculer le reste dans la division euclidienne. On peut l'utiliser en dehors d'une conditionnelle : par exemple, pour affecter dans une variable a le reste dans la division euclidienne du contenu de la variable x par 3 à l'aide de

```
a := x mod 3
```

1.5 Itérations

On l'a vu en Scratch, il est souvent utile de répéter une séquence d'opérations plusieurs fois. Plutôt que de copier-coller le morceau de code, on utilise des boucles permettant d'itérer ce morceau de code. Contrairement à l'exemple simpliste en Scratch, on a souvent besoin de connaître le nombre i d'itérations qui ont été déjà exécutées avant pour exécuter un code différent, dépendant de ce nombre i . Par exemple, essayons d'écrire le code calculant la somme des entiers de 1 à 1000. La façon naïve consiste à utiliser le code suivant (qu'on n'a pas écrit en entier...) :

```
somme := 0
somme := somme + 1
somme := somme + 2
somme := somme + 3
...
somme := somme + 1000
```

Le nombre d'*opérations élémentaires* (si on compte l'addition comme une opération élémentaire et l'affectation comme une autre opération élémentaire) effectuées par ce code est 2001, puisqu'il y a 1000 sommes et 1001 affectations. C'est long et pénible à écrire. À la place, on peut utiliser une boucle **Pour** qui exécute la même instruction pour toutes les valeurs de la variable décrite dans la boucle :

```
somme := 0
Pour n de 1 à 1000 faire
    somme := somme + n
FinPour
```

Notez qu'on exécute exactement le même nombre d'opérations élémentaires, 2001 dans ce cas, mais ce code est bien plus court et lisible que le code précédent.

Évidemment il existe une solution bien plus simple pour réaliser ce calcul, puisqu'on connaît une formule mathématique pour calculer la somme des premiers termes d'une suite arithmétique de raison 1 et de premier terme 1 :

$$1 + 2 + \dots + 1000 = \sum_{n=1}^{1000} n = \frac{1000 \times 1001}{2} = 500 \times 1001 = 500500$$

Cependant, la boucle s'avère indispensable lorsqu'on ne connaît pas de telles formules. Par exemple, si on souhaite calculer la somme des entiers de 1 à 1000 qui sont divisibles par 3 ou par 5, on pourra utiliser le code

```
somme := 0
Pour n de 1 à 1000 faire
    Si (n  $\equiv$  0 mod 3) ou (n  $\equiv$  0 mod 5) alors
        somme := somme + n
    FinSi
FinPour
```

Le nombre d'opérations élémentaires est un peu plus important dans ce cas. Pour chaque itération de la boucle **Pour**, on exécute 2 tests sur n suivi d'une disjonction (**ou**), suivi, dans le pire des cas, d'une somme et d'une affectation : au total, chaque itération exécute donc au plus 5 opérations élémentaires. Puisqu'il y a 1000 itérations dans la boucle, le nombre total d'itérations est de 5000, auquel on ajoute la toute première affectation.

Notez au passage qu'on n'a pas fait figurer de **Sinon** dans la condition ci-dessus, puisqu'il n'y a rien à faire dans ce cas.

On a parfois besoin d'écrire des boucles **Pour** qui égrène les éléments en sens inverse, ou qui saute d'un pas de plus de 1. Par exemple, si on veut réaliser des opérations pour tous les entiers entre 1 et 100 en commençant par le plus grand, on utilisera :

```
Pour n de 100 à 1 faire
...
FinPour
```

Si on souhaite ne visiter que les entiers pairs entre 2 et 200 (c'est-à-dire 2, 4, 6, ..., 198, 200), on utilisera :

```
Pour n de 2 à 200, avec un pas de 2, faire
...
FinPour
```

On ne peut cependant pas utiliser directement ce genre de boucles lorsqu'on ne connaît pas à l'avance le nombre de tours de boucles à exécuter¹. Un exemple typique est illustré par le calcul du nombre d'étapes avant de tomber sur la face 5, lors de tirages répétés d'un dé. En supposant qu'on dispose d'une fonction `lancer_dé()` ne prenant aucun argument (d'où les parenthèses vides) et renvoyant une face entre 1 et 6 tiré de manière aléatoire, on peut trouver le nombre d'étapes attendues avec le code suivant :

```
nombre_étapes := 0
face := lancer_dé()
Tant que (face ≠ 5) faire
    face := lancer_dé()
    nombre_étapes := nombre_étapes + 1
FinTantQue
```

On utilise donc une boucle **Tant que** qui continue à exécuter le contenu de la boucle *tant que* la condition entre parenthèse reste vérifiée : ici, on continue tant qu'on n'est pas tombé sur la face 5. De manière générale, on sort donc de la boucle dès que la condition est violée, c'est-à-dire dès que la négation de la condition est satisfaite.

1.6 Lecture et écriture

Pour finir, comme en Scratch, il nous sera parfois utile d'interagir avec l'utilisateur en imprimant un message ou le contenu d'une variable, ou bien en posant une question à l'utilisateur et attendre sa réponse. On utilisera deux fonctions **écrire** et **lire** pour ces deux opérations, comme illustré par l'exemple suivant :

```
écrire("Êtes-vous sûr de vouloir quitter ?")
réponse := lire()
Si (réponse = "oui") alors
    quitter page...
FinSi
```

En résumé, les structures de contrôle que nous utiliserons dans ce cours sont rappelées en FIGURE 2.

1. sauf dans les langages de programmation permettant l'utilisation de mots-clés pour l'interruption prématurée de boucles, tels que **break** en Python

itérations

```
Pour .. de .. à .. faire
  ..
FinPour
```

```
Tant que .. faire
  ..
FinTantQue
```

fonctions

```
fonction abc(arguments) :
  ..
retourner(..)
```

conditionnelles

```
Si .. alors
  ..
FinSi

Si .. alors
  ..
Sinon
  ..
FinSi
```

écriture/lecture

```
écrire(« .. »)
réponse := lire()
```

variables

```
x := 3
y := 2
x := x+y
```

FIGURE 2 – Syntaxe pour les structures de contrôle, dans les pseudo-codes de ce cours