



(Séquence 2.1

Spécification et commentaire



Écriture de programmes

- ▶ Différence entre **grammaire** et **convention d'écriture**
- ▶ Voir la [charte pour l'écriture de programmes en Scheme](#)



Commentaires

Les commentaires sont faits pour les humains et sont ignorés des machines.

Règles d'usage pour l'emploi des commentaires :

```
;;; la spécification d'une fonction
```

```
;; la spécification d'une fonction interne  
;; ou d'un passage compliqué
```

```
; commentaire local sur ce qui suit  
; ou commentaire local en bout de ligne
```

Principe : on ne commente pas ce que l'on fait mais on commente pourquoi on l'a fait de cette manière !



Spécification

Rappel : Pour pouvoir utiliser une fonction prédéfinie, il faut connaître sa *spécification* :

- ▶ son nom,
- ▶ son type,
- ▶ ce qu'elle calcule,
- ▶ la signification de ses variables.

Quand on définit une nouvelle fonction il faut donner sa spécification pour pouvoir l'utiliser (l'appliquer) correctement.



Fonction : spécification et définition

1. **Spécification** (pour l'humain donc en commentaire)

1.1 **Signature** :

- ▶ nom de la fonction
- ▶ type des arguments
- ▶ type du résultat

1.2 **Sémantique**

- ▶ le **QUOI**
- ▶ ce que la fonction fait (texte en français)

2. **Définition** (pour le calcul)

- ▶ le **COMMENT**
- ▶ implantation, calcul effectué (code en Scheme)



Multiples définitions

On peut écrire plusieurs définitions pour une même spécification.

```
;;; moyenne3: Nombre * Nombre * Nombre -> Nombre
;;; (moyenne3 x y z) rend la moyenne arithmétique
;;; de x, y et z
```

```
;;; Une première définition:
(define (moyenne3 x y z)
  (/ (+ x y z)
     3))
```

```
;;; Une autre définition:
(define (moyenne3 x y z)
  (+ (/ x 3)
     (/ y 3)
     (/ z 3)))
```



Multiples spécifications

On peut écrire plusieurs spécifications pour une même définition.

```
;;; Une première spécification:  
;;; additionTableur: Nombre * Nombre -> Nombre  
;;; (additionTableur x y) additionne x et y
```

```
;;; Une seconde spécification:  
;;; additionTableur: Valeur * Valeur -> Nombre  
;;; (additionTableur x y) additionne x et y si  
;;; ce sont des nombres, rend zero sinon.
```

```
(define (additionTableur x y)  
  (if (and (number? x) (number? y))  
    (+ x y)  
    0 ) )
```



Application de fonction

1. Appliquer la fonction avec des arguments particuliers

```
(moyenne3 13 18 14)
(moyenne3 (+ 4 5) 5 (- 8 1))
```

2. Évaluer une application

- ▶ évaluer chacun des arguments,
- ▶ puis évaluer l'application de la fonction aux valeurs obtenues.

```
(moyenne3 (+ 4 5)
           (moyenne3 5 4 6)
           (- 8 1) )
```

```
(moyenne3 (+ 4 5) (/ 5 0) (- 8 1))
```

ERREUR



Fonctions et formes spéciales

Une expression Scheme est une composition d'applications

- ▶ **Évaluation d'une application**
 - ▶ évaluer chacun des arguments,
 - ▶ évaluer l'application de la fonction aux valeurs obtenues.
- ▶ **sauf pour les formes spéciales : évaluation spéciale**
 - ▶ la forme spéciale `define` : *Définition*
 - ▶ la forme spéciale `if` : *Alternative*
 - ▶ les formes spéciales `and` et `or` : *Connecteurs*
 - ▶ la forme spéciale `let` : *Bloc lexical*



La syntaxe d'une alternative

Règle de grammaire :

```
<alternative> →  
  (if <condition> <conséquence> <alternant> )
```

La *condition* est une expression ayant pour valeur

- ▶ soit vrai #t
- ▶ soit faux #f

La *conséquence* et l'*alternant* sont des expressions.

Convention d'écriture (différente de la syntaxe)

```
(if (positive? x)           ; condition  
   x                       ; conséquence  
   (- x) )                 ; alternant
```



Évaluation d'une alternative

Tout comme la définition, l'alternative est une **forme spéciale**, elle a une **règle d'évaluation** particulière.

- ▶ la condition est évaluée (retourne Vrai ou Faux)
- ▶ selon le résultat,
 - ▶ la conséquence est évaluée
 - ▶ ou l'alternant est évalué
- ▶ et devient la valeur de l'alternative.

Remarque : `define`, `if` ne sont donc pas des fonctions!



Exemple d'alternative

Fonction `valeur-absolue` :

```
;;; Spécification:  
;;; valeur-absolue: Nombre -> Nombre  
;;; (valeur-absolue x) rend la valeur absolue de x
```

```
;;; Une première définition:  
(define (valeur-absolue x)  
  (if ( $\geq$  x 0)  
    x  
    (- x) ) )
```

```
;;; Une autre définition:  
(define (valeur-absolue x)  
  (if (negative? x )  
    (- x)  
    x ) )
```



Exemple 2 (suite)



```
;;; Encore une définition:  
(define (valeur-absolue x)  
  ((if ( $\geq$  x 0) + -) 0  
    x ) )
```

```
;;; Applications:  
(valeur-absolue (+ 3 2))  
(valeur-absolue -5)  
(valeur-absolue (- 5))
```





Fin séquence)

