Licence 3 d'Informatique — Programmation fonctionnelle — TP 1

Début : jeudi 14 septembre 2023

Cette première séance de travaux pratiques a pour but de vous familiariser avec Racket — parfois désigné sous le nom de PLT Scheme (plus ancien) — et son environnement de travail DrRacket — parfois désigné sous l'ancien nom de DrScheme. Tout le matériel de cette première séance se trouve dans le cours « Programmation fonctionnelle, scripts — XML » de moodle, dans le répertoire « Travaux pratiques Scheme 2023 > for-lc-1 », il est composé :

— du présent fichier lc-1.pdf,

— et du fichier d'amorce for-lc-1.scm, écrit dans le langage Scheme.

0 *Getting Started*

0.1 Premier contact

DrRacket — l'interprète du langage Scheme que nous allons utiliser dans le cadre des travaux pratiques de la partie « Programmation fonctionnelle » de l'unité d'enseignement PFS [1] de la 3e année de la Licence d'Informatique — a été installé sur les machines de travaux pratiques fonctionnant sur le réseau Linux : la commande à taper dans une fenêtre Terminal est — remarquez que nous lançons une tâche graphique en arrière-plan — :

```
/opt/racket-8.9/bin/drracket &
```

En fait, pour éviter de devoir retaper le *pathname* complet de cette commande à chaque fois que nous appelons DrRacket, le plus simple est d'ajouter la ligne suivante à la fin du fichier .bashrc de votre *home directory* (créer ce fichier s'il n'existe pas) :

```
alias drracket="/opt/racket-8.9/bin/drracket"
```

après quoi l'abréviation `drracket` sera connue à chaque lancement d'une fenêtre Terminal [2]. Au premier lancement de cet interprète, vous devez choisir le *langage* qu'il traite, par le menu « Language > Choose Language », puis sélectionnez « The Racket Language ». Ensuite, cliquez sur le bouton Exécuter à droite dans le menu de la barre horizontale du haut de la fenêtre. Ce dernier bouton Exécuter sert aussi à provoquer l'évaluation de vos programmes ; par contre, la sélection du langage, une fois effectuée, est en principe faite une fois pour toutes et donc n'aura pas à être recommencée lors des sessions ultérieures [3].

1. **P**rogrammation **F**onctionnelle et **S**cripts.

2. Le programme DrRacket a aussi été installé sur le réseau Windows des machines de travaux pratiques, mais il s'agit d'une vieille version, incomplète et difficilement utilisable pour notre propos ; de plus, les performances sont bien moindres.

3. Si vous avez déjà utilisé DrRacket par le passé et que vous possédez des répertoires .plt-scheme ou .racket dans votre *home directory*, il vaut mieux les supprimer au préalable.

0.2 Installation personnelle

Si vous désirez travailler sur votre propre matériel, vous pouvez télécharger la dernière version de Racket — c'est un programme libre d'emploi — à partir de la *home page* de ce projet :

`http://racket-lang.org`

Après l'installation proprement dite, vous devrez effectuer un ajout supplémentaire, par le menu « File > Install Package ». La solution la plus simple est d'utiliser une référence à la plate-forme GitHub, et l'adresse à renseigner pour le *package source* est :

`https://github.com/lexi-lambda/racket-r7rs.git?path=r7rs#master`

Utilisez également le bouton « Show Details » et voici comment compléter les rubriques :

Package Name ⟸ Use: r7rs
Package Source Type ⟸ Infer
Action to Take ⟸ Install
Package Scope ⟸ Installation
Auto+Update ⟸ update dependencies whenever possible

En cas de problème, n'hésitez bien sûr pas à demander de l'aide à votre gentil animateur. Là aussi, cette manipulation n'est à exécuter qu'une seule fois après l'installation, vous n'aurez pas à la recommencer lors des séances ultérieures de travaux pratiques [4].

1 *Preludio*

Durant cette séance d'initiation au langage de programmation Scheme, vous allez vous familiariser avec la syntaxe et le mécanisme d'évaluation. Il est également essentiel d'apprendre à sauvegarder des fichiers et à les relire, car certains exercices de travaux pratiques vont s'étaler sur *plusieurs* séances : il sera alors nécessaire de sauvegarder votre travail à la fin d'une séance puis de le reprendre à la séance suivante. Il est également important d'apprendre dès le début à bien *indenter* [5] les programmes en Scheme, comme cela vous sera démontré par votre gentil animateur. L'apprentissage de ce réflexe dès le début est *essentiel* : s'il peut être difficile de chercher une erreur dans un programme, cette tâche tourne rapidement à une punition si ledit programme est tellement mal présenté qu'il en devient illisible. En outre, si vous travaillez en binôme, n'oubliez pas que le module PFS comportera — entre autres — une épreuve pratique *individuelle* en Scheme, et que donc, ce sont les *deux* membres du groupe qui doivent se familiariser avec l'outil.

La configuration de l'interprète DrRacket fait apparaître deux fenêtres : la fenêtre supérieure est destinée à l'édition de programmes, les résultats des exécutions étant donnés dans la fenêtre inférieure. Vous pouvez taper directement des programmes dans la fenêtre inférieure pour les faire évaluer immédiatement, ce n'est toutefois pas une très bonne technique car il n'y a aucun moyen de « rattraper » les fautes de frappe, ni d'éditer à nouveau ce qui vient d'être tapé. Votre premier exemple de programme écrit en Scheme, constituant le fichier for-lc-1.scm joint à la distribution, est reproduit à la figure 1.

4. De même, si cette séquence de commandes échoue et que vous possédez des répertoires .plt-scheme ou .racket dans votre *home directory*, supprimez les, puis ré-essayez.

5. Nous présentons toutes nos excuses pour cet anglicisme, en invoquant les circonstances atténuantes et la clémence du jury : ce mot fait quasiment partie de la terminologie utilisée en Informatique.

```
#!r7rs

(import (scheme base) (scheme inexact) (scheme write))
;; Importation des bibliothèques prédéfinies nécessaires. Pour l'instant, retenez simplement que vous
;; aurez toujours besoin de base. De même, nous importerons toujours write pour réaliser des
;; affichages à l'écran. Nous vous signalerons l'importation d'autres bibliothèques le cas échéant : ici,
;; inexact, car nous utilisons quelques fonctions trigonométriques et logarithmiques. Comme vous
;; l'avez deviné, ceci est un commentaire.

(write (+ 2023 (* (+ 22 1) (/ 14 (- 9 2))))) ; (*)

(newline)
(write "Hello world!")
(newline)
(display "Hello world!")
(newline)
(write ((lambda (x) (* x x)) 2024))          ; (**)
(newline)

(define crash                                ; (***) Pour bien montrer que Scheme n'évalue
  (lambda () (/ 0)))                         ; pas sous la forme spéciale lambda.

(define derive-wrt
  (lambda (f1 h)
    (lambda (x) (/ (- (f1 (+ x h)) (f1 x)) h))))

(write ((derive-wrt (lambda (x) (* x x)) 0.0001) 2))
(newline)
(write ((derive-wrt + 0.0001) 2))            ; (****)

(define fact
  (lambda (n) (if (zero? n) 1 (* n (fact (- n 1))))))

(newline)
(write (fact 6))
(newline)

(define pi (* 4 (atan 1)))                   ; atan est la fonction Arc tangente en Scheme.

(define power
  (lambda (U I f0) (* U I (cos (* 2 pi f0)))))

(define fq->power
  (lambda (f0)
    (let ((coefficient (cos (* 2 pi f0))))
      (lambda (U I) (* coefficient U I)))))
```

Figure 1 : Votre premier programme en Scheme (fichier for-lc-1.scm).

La première ligne — « `#!r7rs` » — indique qu'il s'agit d'un programme écrit suivant les conventions de la toute nouvelle version *R7RS* [6]. La deuxième indique l'importation de quelques modules de la bibliothèque initiale. Suivent des expressions à évaluer, l'introduction de défini-

6. ***R**evised*^{7} ***R**eport on the Algorithmic Language **S**cheme*. C'est parce que nous employons cette version que l'installation d'un *package supplémentaire* est nécessaire, comme nous le montrons au § 0.2. Ce *package* a été en principe installé sur les machines des salles de travaux pratiques fonctionnant sur le réseau Linux.

tions — éventuellement récursives, voir l'exemple de la fonction factorielle `fact` — n'en étant qu'un cas particulier. Toutes les expressions sont évaluées, en séquence, mais seules les expressions auxquelles s'applique un ordre d'affichage apparaissent à l'écran [7]. La figure 1 donne des exemples d'utilisation des procédures `write` et `display`, cette dernière étant plutôt pertinente pour l'affichage de messages explicatifs. *Attention !!* rappelons que ces procédures `write` et `display` ont un *effet*, mais qu'elles ne retournent pas de résultat. Mentionnons aussi qu'elles n'effectuent pas le moindre passage à la ligne, pour cela, utiliser la procédure `newline` — qui elle non plus ne retourne pas de résultat — comme cela vous est montré dans la figure 1.

En ce qui concerne l'expression (*) :

— entraînez-vous à utiliser les commandes de l'éditeur pour voir quelles sont les parenthèses appariées :

  ⋆ cliquez à gauche d'une parenthèse ouvrante pour voir toute la partie d'expression qu'elle ouvre,

  ⋆ de même, cliquez à droite d'une parenthèse fermante pour voir toute la partie d'expression qu'elle referme ;

— voyez aussi comment fonctionne l'indentation : ligne après ligne, à chaque frappe de la touche RETURN ; il est aussi possible d'indenter des expressions *a posteriori* à l'aide des commandes Reindent et « Reindent all » du menu Racket, situé au-delà de la fenêtre de DrRacket, dans la bordure supérieure de l'écran ;

— retour à l'expression (*) précédente : décomposez-en bien les différentes phases « à la main » et vérifiez que vous trouvez le même résultat.

Vous pouvez aussi évaluer quelques expressions de votre cru — nous rappelons que les parenthèses doivent être strictement appariées et les règles d'espacement scrupuleusement respectées. Remarquer l'importance de ces règles d'espacement en tapant `(+ 7 2)`, puis `(+7 2)`, et expliquez ce qui se passe dans ce deuxième cas. De même, évaluez d'abord l'expression `(- +2)`, puis l'expression `(- + 2)` et expliquez ce qui se passe dans les deux cas. Remarquez également que le signe « - » pour un nombre négatif doit être accolé aux chiffres qu'il précède.

2 Compléments

2.1 Compléments d'Arithmétique

Remarquez que les fonctions Scheme `+` et `*` admettent un nombre quelconque d'arguments, en évaluant :

```
(+)               ⟹ ??
(+ 6 10 2023 1)   ⟹ ??
(*)               ⟹ ??
(* 12 10 2023 2)  ⟹ ??
```

Quant aux fonctions Scheme `-` et `/`, elles admettent un nombre quelconque, mais non nul d'arguments. Évaluez :

7. Pour toutefois vérifier que toutes les expressions sont évaluées, il suffit de glisser délibérément une expression erronée, par exemple, une division par zéro. Nous reviendrons sur l'évaluation des expressions en Scheme à l'aide de l'exemple de la fonction `crash` donnée dans la figure 1.

```
(-)             ⟹ ??
(- 1)           ⟹ ??
(- 7 2 2023 6)  ⟹ ??
```

La fonction Scheme `/` est la fonction de division *exacte*. Observez son comportement lorsque la division de nombres entiers tombe « juste » :

```
(/ 6 3) ⟹ ??
```

et dans un autre cas :

```
(/ 3 2) ⟹ ??
```

mais nous n'explorerons plus avant cette possibilité — l'emploi de nombres *rationnels* — que plus tard. Cette fonction `/` peut également être employée avec des nombres décimaux, auquel cas son résultat est décimal :

```
(/ 3.0 2.0) ⟹ ??
```

Observez que le point décimal est *contagieux* :

```
(+ 1 1.1) ⟹ ??
(/ 3.0 2) ⟹ ??
```

Ne pas confondre « `/` » qui est la fonction de division exacte avec « `quotient` » qui est la fonction de quotient entier :

```
(/ 4 3)         ⟹ ??
(quotient 4 3)  ⟹ ??
```

la fonction `remainder` donnant le reste de la division euclidienne :

```
(remainder 4 3) ⟹ ??
```

Le résultat de la fonction `remainder` a toujours le signe du dividende ; il existe également une fonction `modulo`, mais le résultat qu'elle retourne a toujours le signe du diviseur.

Le nombre π ayant été défini en utilisant la relation $\tan \frac{\pi}{4} = 1$ (*cf.* figure 1), donnez les expressions Scheme correspondant aux expressions mathématiques suivantes et faites-les calculer par l'interprète :

$$4.1 * 9.8 + \frac{2\pi}{3} \qquad \exp \pi + \ln \sin 2.1 + \sin(\frac{1}{\pi} + \pi)$$

Pour ce faire, notez que les fonctions exponentielle, sinus et logarithme népérien se notent respectivement `exp`, `sin`, et `log` en Scheme.

2.2 Nos premières fonctions

Revenons à l'expression (**) de la figure 1, voici, à partir de cette expression, quelques variations à évaluer — certaines peuvent être erronées [8], auquel cas il importe de voir pourquoi — :

8. Si une expression erronée a été tapée dans l'éditeur, vous pouvez en désactiver l'évaluation en la sélectionnant, puis en la commentant au moyen des commandes du menu Racket. L'effet inverse est obtenu par la commande Uncomment de ce même menu. La commande « Comment Out with a Box (Commenter à l'aide d'une boîte) » permet de court-circuiter globalement plusieurs lignes consécutives, alors que la commande « Comment Out with Semi-Colons Commenter à l'aide de points-virgules) » permet un contrôle plus fin, ligne par ligne. Vous pouvez bien sûr essayer les deux techniques.

```
((lambda (x) (* x x)) 2023)                ⟹ ??
((lambda (x y) (* x y y)) 2023)            ⟹ ??
((lambda (x y) (* x y y)) 2023 2022)       ⟹ ??
((lambda (x y) (* x y y)) 2023 2022 2021)  ⟹ ??
```

Comprenez bien la différence qui existe entre *valeur*, d'une part, et *fonction* qui retourne une valeur, d'autre part, par l'évaluation des trois expressions suivantes :

```
2023              ⟹ ??
(lambda () 2023)  ⟹ ??
((lambda () 2023)) ⟹ ??
```

Vous pouvez en outre constater cette différence entre fonction et application de fonction par l'exemple de la fonction prédéfinie `+` employée avec zéro argument :

```
+   ⟹ ??
(+) ⟹ ??
```

Maintenant, vous devriez pouvoir expliquer ce qui se produit avec les expressions suivantes — certaines sont erronées — :

```
(2023)                              ⟹ ??
((lambda () 2023))                  ⟹ ??
(((lambda () 2023)))                ⟹ ??
(((lambda () (lambda () 2023))))    ⟹ ??
```

Enfin, ne pas oublier que Scheme n'évalue pas sous la forme spéciale `lambda`, comme cela vous est suggéré par l'évaluation de l'expression (***).

3 Nos premières définitions

3.1 Dérivation

Nous avons donné dans la figure 1 la définition d'une fonction `derive-wrt` de calcul numérique de la dérivée d'une fonction f_1 au moyen de la formule :

$$x \mapsto \frac{f_1(x+h) - f_1(x)}{h}$$

f_1 étant une fonction à un argument numérique et retournant un nombre, h un nombre réel non nul jugé suffisamment petit. Un exemple de dérivée de la fonction d'élévation au carré vous est donné ; essayez d'autres exemples. Vérifiez que l'appel de `derive-wrt` avec une fonction dont le nombre d'arguments formels requis est différent de 1 provoque une erreur :

```
((derive-wrt (lambda (x y) (+ (* 2 x) y)) 0.0001) 2) ⟹ ??
```

À l'inverse, pourquoi l'expression (****) est-elle licite et produit un résultat ?

3.2 Puissance électrique

Nous rappelons d'abord la formule bien connue en Électricité :

$$\mathcal{P}(U, I, f) = U\, I\, \cos 2\pi f$$

qui donne la puissance en Watts d'une portion de circuit, étant donné la différence de potentiel efficace U de la portion de circuit en Volts, l'intensité efficace I en Ampères, la fréquence f du courant alternatif en Hertz. Quant à la notation que nous venons d'adopter, elle est destinée à mieux faire ressortir que le volume est fonction (au sens mathématique de ce mot) de U, I et f. Nous vous avons aussi rappelé dans la figure 1 la fonction `power`, réalisant ce calcul et écrite en cours — la fonction cosinus se note `cos` en Scheme. Essayer l'expression :

`(power 220 7 50)` $\Longrightarrow$ `??`

Comme nous l'avons fait remarquer durant le cours, la fréquence est en pratique très souvent fixée à une valeur donnée et ne varie pas : en particulier, elle est fixée à 50 Hz pour les installations domestiques. Ce qui nous a mené à l'écriture d'une fonction `fq->power` réalisant :

$$f_0 \mapsto (\mathcal{P}_{f_0} : (U, I) \mapsto U\ I\ \cos 2\pi f_0)$$

— la fonction $\mathcal{P}_{f_0}$ étant parfois notée $\mathcal{P}(\cdot, \cdot, f_0)$ dans certains ouvrages mathématiques — cette fonction `fq->power` vous est elle aussi rappelée dans la figure 1. Vous pouvez remarquer que cette fonction « factorise » un certain nombre de calculs en utilisant la possibilité de *clôture lexicale* du langage Scheme, et vous pouvez définir une fonction `power-50` comme suit :

```
(define power-50 (fq->power 50))
```

et retrouver le résultat déjà rencontré :

`(power-50 220 7)` $\Longrightarrow$ `??`
`((fq->power 50) 220 7)` $\Longrightarrow$ `??`

3.3 Dilatation des corps physiques

Dans sa « philosophie », cet exemple est analogue au traitement du calcul de la puissance électrique d'une portion de circuit. Lorsqu'on étudie la dilatation des corps en Physique, on démontre que le volume V à la température t — exprimée en degrés Celsius — et le volume V_0 du même corps à la température de 0°C sont reliés par l'égalité suivante :

$$V(t, V_0, \lambda_b) = V_0(1 + \lambda_b t)$$

les volumes étant donnés en mètres cubes et le coefficient de dilatation λ_b — donné en $^{\circ}C^{-1}$ — étant fonction de la matière qui constitue le corps considéré [9]. Quant à la notation que nous venons d'adopter, elle est destinée à mieux faire ressortir que le volume V est *fonction* — au sens mathématique de ce mot — des arguments t, V_0 et λ_b.

$\Longrightarrow$ Définir une fonction `volume-at` telle que l'évaluation de l'expression :

`(volume-at` t V_0 $lambda_b$`)`

— où t, V_0 et $lambda_b$ sont des nombres réels — retourne le volume qu'occupe un corps à la température t, son volume à 0°C étant V_0, et son coefficient de dilatation $lambda_b$. Essayer :

`(volume-at 30 1 1e-5)` $\Longrightarrow$ `??`

9. Pour un corps solide (resp. liquide), il est de l'ordre de $10^{-5}\,^{\circ}C^{-1}$ (resp. $10^{-3}\,^{\circ}C^{-1}$).

— « `1e-5` » étant mis pour « 10^{-5} » (attention : *aucune* espace de part et d'autre du signe « - »).

Si l'on considère une matière donnée, alors le coefficient de dilatation peut être vu comme une constante, et le volume n'est plus fonction que de la température et du volume à 0°C. Ce que les mathématiciens traduisent par la notation suivante :

$$V_{\lambda_b}(t, V_0) = V_0(1 + \lambda_b t)$$

$\Longrightarrow$ Donner la fonction Scheme :

`body->volume-at`

qui réalise $\lambda_b \mapsto V_{\lambda_b}$: remarquez bien que `body->volume-at` est une fonction à *un seul* argument formel, et qu'un résultat *quelconque* de `body->volume-at` est une fonction à deux arguments formels :

```
(define body->volume-at
  (lambda (lambda-b)
    (lambda (t V0) ...)))
```

$\Longrightarrow$ Utiliser cette fonction `body->volume-at` pour définir une fonction `gas-volume-at` calculant le volume d'un gaz à une température donnée, lorsqu'on connaît son volume à 0°C. Le coefficient de dilatation de tout gaz [10] est :

$$\alpha = \frac{1}{273.15}$$

et dès lors :

```
(define gas-volume-at (body->volume-at ...))
```

Évaluez :

```
(gas-volume-at 30 1) ⟹ ??
```

— le volume, en mètres cubes, et à 30°C, d'un mètre cube de gaz à 0°C. Revenons à la fonction `body->volume-at`, nous pouvons voir qu'elle nous permet de dériver aisément des fonctions de calcul de volume lorsque le coefficient de dilatation est connu.

3.4 *Interludio*

Enfin, sauvegardez les définitions et tests dans un fichier dont vous choisirez le nom (dans le menu File, sélectionnez « Save Definitions (sauvegarder les définitions) » ou « Save Definitions as... »). Puis quittez DrRacket (dans le menu File, sélectionnez Quit), relancez-le, et relisez le contenu de votre fichier (dans le menu File, sélectionnez « Open... » ou « Open Recent »). Vous pouvez déclencher l'évaluation en séquence de toutes les expressions présentes dans l'éditeur de DrRacket en cliquant sur le bouton « Racket > Run ».

10. C'est la loi de Louis Joseph Gay-Lussac (1778–1850), origine de la notion de *gaz parfait*. Cette formule ne s'applique qu'à pression constante, ce que nous supposerons être le cas. Quant à la valeur −273.15, c'est la température du *zéro absolu*, exprimée en degrés Celsius.

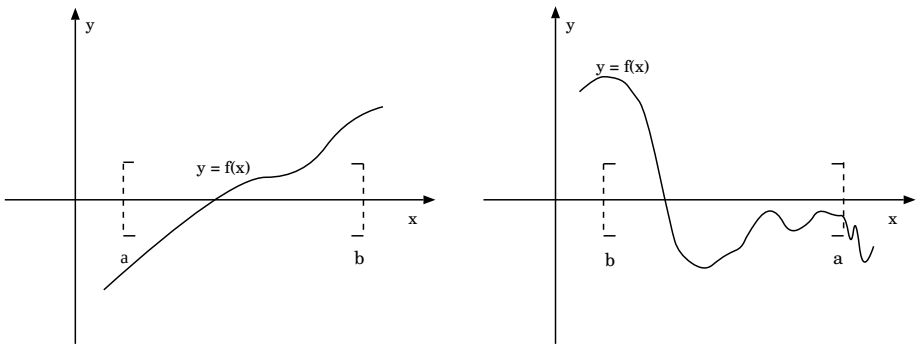

Figure 2 : Applications possibles pour la recherche par dichotomie.

---

4 Exercice final

4.1 *Adagio molto*

Soient x et y deux nombres réels, écrire les deux fonctions utilitaires suivantes :

— `average`, telle que l'évaluation de l'expression `(average x y)` retourne la moyenne de x et de y, soit $\dfrac{x+y}{2}$;

— `close-enough?`, telle que l'évaluation de l'expression `(close-enough? x y ε)` — où ε est un nombre réel jugé suffisamment petit — retourne la valeur logique `#t` si $|x-y| < \varepsilon$, la valeur `#f` sinon ; autrement dit, `close-enough?` est une fonction d'égalité entre nombres réels, à ε près.

Indication La fonction retournant la valeur absolue est `abs` en Scheme.

4.2 *Allegro agitato*

Soit $f_1 : \mathbb{R} \to \mathbb{R}$, et soient $a, b \in \mathbb{R}$, recherchons une solution de l'équation $f_1(x) = 0$ dans l'intervalle $[a, b]$, sachant :

— que la fonction f_1 est continue,

— que $f_1(a)$ et $f_1(b)$ sont de signes opposés,

ce qui implique l'existence d'au moins une solution. (Deux exemples de telles fonctions vous sont suggérés dans la figure 2.)

Supposons, sans perte de généralité, que $f_1(a) < 0$ et $f_1(b) > 0$. Soit m la moyenne de a et de b, calculons $f_1(m)$. Trois cas se présentent :

(i) $f_1(m) = 0$: m est solution de l'équation $f_1(x) = 0$;

(ii) $f_1(m) > 0$: étant donné que par ailleurs, $f_1(a) < 0$, il existe au moins une solution entre a et m ;

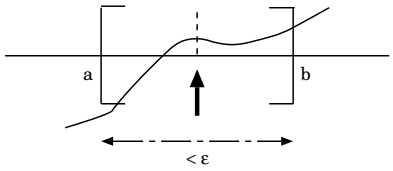

Figure 3 : Terminaison de la recherche par dichotomie.

(iii) $f_1(m) < 0$: il existe au moins une solution entre m et b.

Autrement dit : si nous n'avons pas trouvé une solution, nous pouvons réitérer le procédé sur un intervalle — soit $[a, m]$, soit $[m, b]$ — dont la longueur a été réduite de moitié. Cette méthode est appelée **recherche par dichotomie** de la solution de l'équation $f_1(x) = 0$ dans l'intervalle $[a, b]$.

Une question reste en suspens : *quand arrêter la recherche ?* Nous ne pouvons pas diviser indéfiniment la longueur de l'intervalle par 2 ; de plus, compte tenu des erreurs d'approximation des calculs sur les nombres réels, il est *très improbable* que nous trouvions une solution exacte comme nous l'avons envisagé dans le cas (i).

Nous fixons donc une précision ε. Si la longueur de l'intervalle $[a, b]$ est inférieure à ε, nous considérons que $\dfrac{a+b}{2}$ est racine à $\dfrac{\varepsilon}{2}$ près, comme nous le suggérons dans la figure 3. Sinon, le calcul peut encore être affiné et nous appliquons le procédé vu plus haut.

Écrire la fonction `look-for-root`, réalisant ce *modus operandi*. Par hypothèse, tout appel de cette fonction, de la forme `(look-for-root` f_1 *negative-point positive-point* ε`)` — où f_1 est une fonction à un argument formel et *epsilon* la précision — vérifie :

$$f_1(\textit{negative-point}) < 0 < f_1(\textit{positive-point})$$

Il est important de s'assurer que les appels récursifs de cette fonction sont en accord avec cette hypothèse. Comme exemple d'application, tenter la résolution de l'équation $x^2 - 2$ dans l'intervalle $[1, 2]$.

Remarque Pour éviter que les résultats soient donnés sous la forme de fractions rationnelles, spécifier les nombres réels des exemples en en utilisant le point décimal, par exemple, « `2.0` » pour « 2 ».

4.3 *Postludio*

Après la mise au point de la fonction `look-for-root` précédente, il s'agit maintenant de réaliser son enrobage, de telle sorte qu'un utilisateur puisse chercher la racine d'une fonction sans être contraint par les hypothèses nécessitées par la fonction `look-for-root`. Ce qui nous mène à une fonction `dichotomy`, telle que pour $f_1 \in \mathcal{F}_{\mathbb{R}\to\mathbb{R}}, r_1, r_2, \textit{epsilon} \in \mathbb{R}$ — f_1 étant supposée continue et *epsilon* étant la précision —, l'évaluation de l'expression :

`(dichotomy` f_1 r_1 r_2 *epsilon*`)`

commence par s'assurer que la recherche par dichotomie est applicable — si ce n'est pas le cas, afficher un message d'erreur, et retourner la valeur `#f`. Ensuite, on devra évaluer, selon les signes de $f_1(r_1)$ et $f_1(r_2)$:

— soit l'expression `(look-for-root` f_1 r_1 r_2 *epsilon*`)`,

— soit l'expression `(look-for-root` f_1 r_2 r_1 *epsilon*`)`,

Comme exemples d'application, tenter :

$$\begin{array}{lll} \text{la résolution de} & x^2 - 2 = 0 & \text{dans l'intervalle } [1,2], \\ \ldots\ldots\ldots\ldots & x^2 - 2 = 0 & \ldots\ldots\ldots\ldots\ [0,1], \\ \ldots\ldots\ldots\ldots & \cos x - \dfrac{x}{4} = 0 & \ldots\ldots\ldots\ldots\ [1,2]. \end{array}$$

Rendre ensuite la fonction `look-for-root` locale à la fonction `dichotomy`. Lorsque cette opération est effectuée, est-il possible de supprimer des arguments formels de la fonction locale `look-for-root` ? Si oui, lesquels ?