Programmation fonctionnelle

Cours n°1 : Introduction à Racket et premières fonctions

Licence 1 Informatique, Université Paris 8

19 Septembre 2022

Organisation et évaluation

- 1h30 de cours en salle **J001** les lundis de 15h à 16h30, et 3h en salle machine **A162** le mercredi de 15h à 18h pour des TD et TP.
  - Cours étalé sur 10 semaines, a priori pas de récupération envisagée sur les dernières semaines.
  - **Présence obligatoire** aux TP (sauf justification), puisque les fichiers déposés à l'issue des TP seront pris en compte dans l'évaluation.
  - Supports sur **Moodle** (`https://moodle.univ-paris8.fr/`), discussion sur le canal **Mattermost** (`https://talk.up8.edu`) appelé *L1A Programmation fonctionnelle (2022-2023)*.
- **Évaluation du cours:** La note finale pour ce cours sera composée de 3 notes:
  - **(30 %)** 3 interrogations écrites courtes ($\sim$ 15 minutes) réalisées en début de certains TP. Vous serez prévenus des dates assez longtemps à l'avance.
  - **(30 %)** Une note sur le rendu de certains TP (pas tous, uniquement les meilleurs) sélectionnés.
  - **(40 %)** Durant la dernière séance de TP, un **TP test** sur toutes les connaissances acquises durant le semestre.
- La note finale sera calculée suivant le principe du max suivant:

$$\max(\textbf{Note TP test}, \textbf{40\% Note TP test + 30\% Notes TP + 30 \% Notes interros})$$

Paradigmes de programmation

- Il existe plusieurs façons d'aborder la programmation, au travers de différents **paradigmes de programmation.**
- **Exemple : Programmation orientée objet.** On représente ce qu'on souhaite modéliser sous forme d'objets.
  - Par exemple, une voiture sera un objet de type voiture qui aura 4 roues, un moteur, un kilométrage; qui sont des attributs de l'objet.
  - La voiture peut également accélérer, ralentir. Ce sont des fonctions de l'objet.
  - Un objet sera alors représenté par une classe qui regroupera les attributs et les fonctions de cet objet. C'est par exemple en Python, ou en C++.
- **Exemple : Programmation impérative.** Chaque programme est constitué de séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme, par exemple changer la valeur d'une variable.
  - Ce paradigme sera utilisé au second semestre dans le cours de *Programmation impérative* en langage *C*.

Qu'est ce que la programmation fonctionnelle ?

- La **programmation fonctionnelle** est un autre paradigme de programmation, qu'on associe souvent à la programmation de type **déclarative**, en opposition à **impérative**. Il consiste à ne considérer que des évaluations de fonctions prédéfinies, dépendant uniquement des arguments choisis.
  - Dans les années 1930, avant l'existence des ordinateurs, le mathématicien Alonzo Church développa un langage (ou système formel) appelé lambda calcul, permettant de définir les notions de fonction et d'application sous forme de logique combinatoire.
  - Alan Turing développa les machines de Turing et la théorie de la calculabilité en 1958.
  - John Mac Carthy créa le langage Lisp, réalisant ainsi la première solution pratique liant un langage de programmation et le lambda calcul; dans les années 1970 apparaissent Lisp Machine Lisp, Scheme et NIL (New Implementation of Lisp).
  - Gerald Jay Sussman et Guy Lewis Steele créent Scheme dans le but de simplifier Lisp avec une syntaxe très simple et peu de mots clés.
  - Racket est une évolution de Scheme, développé par le groupe Programming Language Team de Rice University.
  - Références d'aide pour le langage Racket : `https://racket-lang.org/` et `https://docs.racket-lang.org/`.

Prog. fonctionnelle vs prog. impérative

- L'utilisation d'un langage fonctionnel implique pour les programmeurs une sémantique plus stable : pour un même problème résolu par deux programmeurs, les solutions finales en fonctionnel seront plus proches que les solutions d'homologues programment en impératif.

- Cette qualité d'expression sémantique peut également devenir source de blocage pour certains programmeurs, obligeant à une présentation plus formelle et plus systématique permettant la bonne exécution du programme.

- En programmation fonctionnelle, on a donc un style de code rigide, avec une programmation fonctionnant par application de fonctions mathématiques immuables dans le temps, indépendant de l'état courant.

- La prog. fonctionnelle permet d'assurer l'absence d'effet de bords et donc de débordements en mémoire, ainsi que de remplacer des expressions et valeurs sans modification de comportement.

Prog. fonctionnelle vs prog. impérative

- **Un exemple fonctionnel vs impératif :**

Un programme impératif

```
1
2   int sommeNombre(int Nombre)
3   {
4     int somme = 0;
5     for (int i=0; i<=Nombre; i++)
6     {
7       somme += i;
8     }
9     return somme;
10  }
```

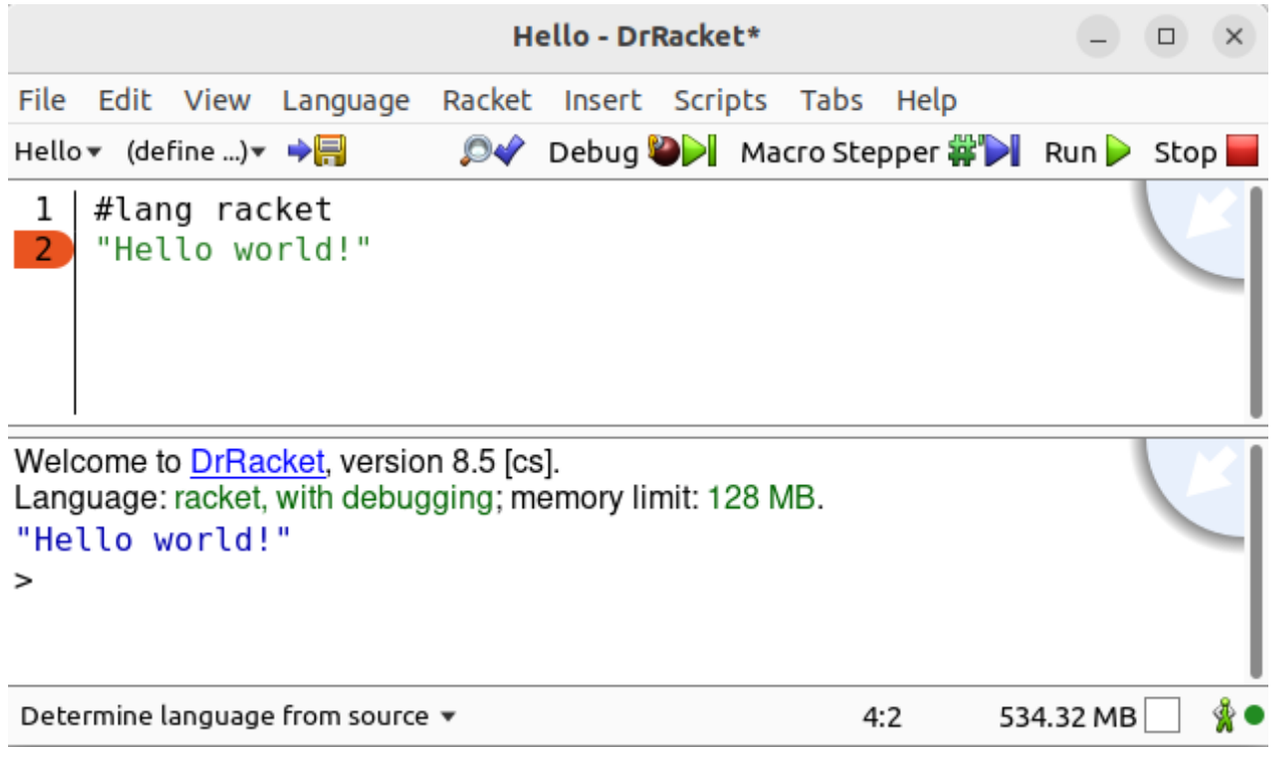
Un programme fonctionnel

```
1  (define somme
2    (lambda (N)
3      (if (= N 0)
4          0
5          (+ N (somme (- N 1))))))
```

- **En impératif :**
  - Plusieurs affectations de la variable **sum** avant d'obtenir le résultat final, et **sum +=x** ne retourne pas de valeur, mais a pour effet de bord de modifier la valeur contenue dans la variable sum.
  - **sum** est une variable mutable qui peut changer tout au long de l'exécution du programme.
  - On donne des ordres à l'ordinateur en lui disant à chaque étape quelle valeur il faut affecter à telle variable; notre programme dit comment calculer la somme des éléments de 1 à N.
- **En fonctionnel :**
  - Aucune affectation.
  - On dit à l'ordinateur ce qu'est la somme des éléments de la liste de manière plus déclarative : si la liste est vide on ne fait rien; sinon on ajoute la dernière valeur à la somme (déjà calculée) des valeurs précédentes.
  - C'est un exemple de fonction **récursive**, omniprésente en programmation fonctionnelle.

DrRacket

- **DrRacket** (prononcé « Docteur Racket ») est un environnement de développement permettant de programmment en Racket.



- La partie supérieure présente le programme (dont les lignes sont numérotées), la partie inférieure présente l'exécution du programme.

- Attention au langage: cliquer sur *Déterminer le langage à partir de la source*, puis *Choisir langage* et être sûr d'être en langage Racket.

- La première ligne *#lang racket* pourrait être omise, mais on l'écrira toujours par vigilance.

Nombres en Racket

- Les principaux types de nombres sont entiers, rationnels, complexes, décimaux (ou flottants simple ou double précision), hexadécimaux (base 16), octodécimaux (base 8) ou binaires (base 2).
- Les fractions sont réduites, les nombres en notation scientifique voient leur décimale déplacée pour réduire leur nombre de zéros, et les nombres hexadécimaux, octodécimaux et binaires sont réduits en base 10.
- Les constantes mathématiques i, π ou encore e sont accessibles via i, pi et e.
- **Exemples :**

Programme

```
1 #lang racket
2 1
3 1+2i
4 3.14
5 pi
6 0.0602e+23
7 #b010101
8 #x29
```

Résultat

```
1
1+2i
3.14
3.141592653589793
6.02e+21
21
41
```

Nombres en Racket

- La virgule est représentée par un « . » selon les standards internationaux.
- Attention aux nombres rationnels : la représentation des nombres rationnels implique l'écriture $1/2$, qui est à ne pas confondre avec le flottant 0.5. Le première est dit du type **exact** tandis que le second est **inexact**.
- Plus généralement, les nombres sont soit **exacts** (nombres entiers, rationnels), soit **inexacts** (décimaux et complexes).
- Il existe deux autres préfixes : $\#e$ pour les nombres exacts, et $\#i$ pour les nombres inexacts : ainsi, le nombre $1 + 0i$ sera considéré comme exact, alors que $\#i1 + 0i$ sera considéré inexact.
- Lorsque nous voudrons comparer des nombres entre eux, il faudra bien faire attention à leur type, entre exact et inexact.

Fonctions, opérations

- Une **fonction** en Racket est un programme dépendant de **paramètres**, et retournant un **résultat**. Ces deux données peuvent être de différents types: nombre, caractère, chaîne de caractères (string), liste, booléen (vrai/faux), etc.

- **Syntaxe de l'appel à une fonction :**
  - Parenthèse ouvrante
  - Nom de la fonction
  - Espace
  - Premier argument
  - Espace
  - Deuxième argument
  - Etc.
  - Parenthèse fermante

(NomFct Arg1 Arg2 ... Argn)

- Attention, il faut donc donner à une fonction le bon nombre d'arguments, et du bon type !

Fonctions, opérations

- Les opérations usuelles $+, -, \times, \div$ ne dérogent pas à la règle : ce sont des fonctions qui prennent 2 arguments (ou plus en fait !) et qui renvoient un nombre. Ainsi, pour représenter $5 + 3$, on va écrire

$$(+\ 5\ 3)$$

- On parle de notation **préfixée** (aussi notation polonaise, de par son introduction en 1924 par Jan Lukasiewicz). Ce choix de notation est utilisé en Racket, mais n'est pas propre à tous les langages de programmation fonctionnelle.

- **Exemples :**

Programme

```
#lang racket
(+ 12 2 3)
(* 2 7)
(+ (* 2 5) (- 3 1))
(* 5)
```

Fonctions, opérations

- Les opérations usuelles $+, -, \times, \div$ ne dérogent pas à la règle : ce sont des fonctions qui prennent 2 arguments (ou plus en fait !) et qui renvoient un nombre. Ainsi, pour représenter $5 + 3$, on va écrire

$$(+\ 5\ 3)$$

- On parle de notation **préfixée** (aussi notation polonaise, de par son introduction en 1924 par Jan Lukasiewicz). Ce choix de notation est utilisé en Racket, mais n'est pas propre à tous les langages de programmation fonctionnelle.

- **Exemples :**

Programme	Résultat
`#lang racket`	
`(+ 12 2 3)`	17
`(* 2 7)`	14
`(+ (* 2 5) (- 3 1))`	12
`(* 5)`	5

Fonctions utiles

- Voici d'autres fonctions prenant des nombres en paramètres :
  - **(expt 2 3)** renvoie la puissance 2^3, donc 8;
  - **(sqrt 3)** renvoie la racine $\sqrt{3}$ (sous forme de flottant, donc une valeur approchée de la valeur exacte);
  - **(quotient 13 3)** retourne le quotient de la division euclidienne de 13 par 3, donc 4 puisque $13 = 4 \times 3 + 1$;
  - **(remainder 13 3)** retourne le reste de la division euclidienne de 13 par 3, donc 1;
  - **(modulo 13 3)** renvoie la valeur du premier paramètre modulo le second, soit 1;
  - **(add1 4)** (resp. **(sub1 4)**) ajoute 1 (resp. enlève 1) à la valeur passée en paramètre;
  - **(max 3 5)** (resp. **(min 3 5)**) renvoie le maximum (resp. le minimum) des deux fonctions passées en paramètres;
  - **(gcd 6 4)** renvoie le plus grand commun diviseur de 6 et 4, à savoir 2;
  - **(lcm 6 4)** renvoie le plus petit multiple commun de 6 et 4, à savoir 12;
  - **(round 2.6)** renvoie l'entier le plus proche de 2.6, donc 3;
  - **(floor 2.6)** renvoie l'entier juste en dessous de 2.6, donc 2;
  - **(numerator 17/4)** (resp **(denominator 17/4)**) retourne numérateur (resp. dénominateur) de la fraction en paramètre;
  - **(random 10)** renvoie un nombre aléatoire entier dans l'ensemble $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$;
  - **(random)** renvoie un nombre réel aléatoire dans $[0, 1]$.

Prédicats

- Un **booléen** est un type de donnée qui a une valeur de vérité, c'est à dire qu'on peut déterminer si il est vrai ou faux. Par exemple, quand on écrit « $2 < 3$ », c'est une assertion qui est vraie.
- Les **prédicats** sont des fonctions qui peuvent renvoyer deux valeurs : vrai ou faux. En Racket, ces valeurs sont encodées par $\#t$ et $\#f$.
- En pratique, beaucoup de prédicats permettent de tester si certaines conditions sont vraies ou fausses, par exemple :
  - **(even? a)** permet de tester si le nombre a est pair, et donc renvoie $\#t$ si c'est le cas, $\#f$ sinon;
  - **(odd? a)** permet de tester si le nombre a est impair;
  - **(integer? a)** permet de tester si le nombre a est entier;
  - **(rational? a)** permet de tester si le nombre a est rationnel;
  - **(real? a)** permet de tester si le nombre a est réel;
  - **(complex? a)** permet de tester si le nombre a est complexe;
  - **(boolean? a)** permet de tester si l'argument a est un booléen, à savoir une assertion qui a une valeur vraie ou fausse;
  - **(number? a)** permet de tester si l'argument a est un nombre;
  - **(negative? a)** (resp. **(positive? a)**) permet de tester si l'argument a est un nombre negatif (resp. un nombre positif);
  - **(zero? a)** permet de tester si l'argument a est un nombre, égal à 0;
  - ... et pleins d'autres qui permettent d'autres tests, nous en verrons certaines.

Booléens et tables de vérité

- On peut construire des booléens "plus gros" à partir de certains booléens en les combinant à partir d'opérateurs logiques : **négation** (not en Racket), **et** (and en Racket) et **ou** (or en Racket).
- **Exemple :** Le booléen « $(2 < 3)$ **et** $(3 > 5)$ » est faux, puisque la première condition est vraie, mais la deuxième non. Pour qu'un **et** soit évalué à vrai, il faut que ses deux arguments soient vrais. En pratique, on utilise des **tables de vérité** pour déterminer la valeur de vérité d'un booléen construit à partir de **négation**, **et** et **ou**.

X	Y	X ET Y
Vrai	Vrai	Vrai
Vrai	Faux	Faux
Faux	Vrai	Faux
Faux	Faux	Faux

X	Y	X OU Y
Vrai	Vrai	Vrai
Vrai	Faux	Vrai
Faux	Vrai	Vrai
Faux	Faux	Faux

X	NON X
Vrai	Faux
Faux	Vrai

Définition

- La fonction **define** permet de définir des variables, et des fonctions qu'il est possible de réutiliser.

- **Définition de variables :**

```
#lang racket
(define a 5)
a
(define b 7)
(+ a b)
```

renvoie les valeurs 5 et 12.

- On affecte alors à la variable *a* la valeur 5, qu'elle gardera dans les évaluations faisant appel à *a*. On peut éviter la fonction **(quote a)** ou son équivalent **'a** pour empêcher l'évaluation.
  - Par exemple, **(quote a)** –> a et **(quote (+ 1 2))** –> (+ 1 2).

- Pour définir une fonction, on a deux syntaxes possibles:

```
#lang racket
(define (f x) (if (>= x 10) #t #f))
(f 9) (f 10) (f 11)
```

```
#lang racket
(define f (lambda (x) (if (>= x 10) #t #f)))
(f 9) (f 10) (f 11)
```

qui renvoient toutes deux *#f*, *#t*, *#t* sur les appels en ligne 2.

Exemples

- Voici deux manières différentes d'écrire une fonction qui prend un argument un nombre x, et renvoie $3 * x$.

```
1 | #lang racket
2 | (define (fois3 x)
3 |   (* 3 x))
4 |
5 | (fois3 3)
```

```
1 | #lang racket
2 | (define fois3b
3 |   (lambda (x)
4 |     (* 3 x)))
5 |
6 | (fois3b 3)
```

- On peut ajouter du commentaire dans l'éditeur Racket, à l'aide du « ; ». Cela est recommandé pour préciser a **spécification** de la fonction, c'est à dire quels sont ses types d'arguments, et quel type d'objet elle renvoie. Par exemple :

```
1 | #lang racket
2 | (define plus10
3 |   (lambda (n); n est un nombre entier, auquel
4 |             ; on va ajouter 10.
5 |     (+ n 10)))
6 |
7 |
8 | (plus10 5)
```

Définition et test

- Conseil : utiliser la deuxième option afin de bien faire la différence entre le nom de la fonction et ses arguments. Il faut utiliser le mot clé **lambda** suivi d'une liste contenant les arguments de la fonction. Voici la syntaxe complète :

```
(define fonction
        (lambda liste-des-paramètres
                instructions))
```

- Quelques remarques :
  - Il est également possible d'utiliser **lambda** en utilisant et définissant *in situ* les traitements recherchés, par exemple

```
#lang racket
((lambda (x) (if (>= x 10) #t #f)) 9)
```

    Ceci définit une fonction (sans nom) qui évalue l'instruction pour l'argument donné dans le lambda, ici x, qui sera évalué à 9 étant le deuxième argument.

  - En définissant des fonctions, il apparaît des notions de fonction principale, et auxiliaire. On peut ainsi encapsuler un nouveau define à l'intérieur d'une fonction afin d'avoir une fonction locale (utilisable que dans les instructions de la fonction de départ), permettant d'éviter une multiplication de fonctions principales.

Définition et test

- **Exemple :** Dans le programme ci-dessous, la fonction g1 est la fonction principale qui utilise la fonction auxiliaire f, déclarée pour cette occasion. Cependant, on pouvait aussi encapsuler un nouveau define comme dans les définitions de g2 et g3 (équivalentes à g1) : dans le cas de g2 et g3, les fonctions h sont locales et ne sont donc utilisables que dans celles-ci.

```
#lang racket
(define (f x y) (* x y))
(define (g1 x) (f x x))

(define (g2 x)
  (define (h x y) (* x y))
  (h x x))

(define (g3 x)
  (define h (lambda (x y) (* x y)))
  (h x x))
```

Conditions

- On a vu précédemment le mot-clé **if**. Il permet de réaliser un **test**, renvoyant une certaine valeur selon le résultat de ce test.
- Voici la syntaxe d'une condition en Racket :

(if condition valeur-si-vrai valeur-si-faux)

- **Un exemple de fonction :** On veut tester si un nombre est un multiple de 3, si oui en renvoie vrai et sinon faux.

```
#lang racket
(define multiple3
  (lambda (n); n un nombre entier -> renvoie un booléen
    (if (= (modulo n 3) 0) #t #f)))


(multiple3 18)
(multiple3 29)
```

qui renvoie alors $\#t$ et $\#f$.

Conditions multiples

- Parfois, on va écrire une condition **if**, et dans les instructions de cette cette condition, on aura à refaire un test avec un nouveau **if**, et cela peut être encore le cas...
- Le mot clé **cond** permet d'éviter l'imbrication de plusieurs **if** en permettant de lister dans un ordre à bien choisir les différentes conditions qui seront testée. Voici la syntaxe, et son équivalent n'utilisant que des **if**:

```
(if test1 valeur1
   (if test2 valeur2
         ...
         valeurN) ...))
```

```
(cond  (test1 valeur1)
       (test2 valeur2)
       ...
       (else valeurN))
```

- Voici un exemple où l'utilisation du **cond** est utile, pour tester si un symbole est un nombre et ensuite si il est positif.

```
1 #lang racket
2 (define testnbpositif
3   (lambda (N)
4     (if (number? N)
5         (if (> N 0)
6             #t
7             #f)
8         #f)))
```

```
1 #lang racket
2 (define testnbpositif2
3   (lambda (N)
4     (cond ((not (number? N)) #f)
5           ((> N 0) #t)
6           (else #f))))
```

Ces deux fonctions sont équivalentes, mais la version utilisant **cond** est plus compacte.

Récursivité

- Une **fonction récursive** est une fonction qui fait appel à elle même pour s'exécuter. En pratique, pour appliquer la fonction sur un argument donné, on va supposer qu'on connaît le résultat de l'appel de la fonction sur une donnée « plus petite », pour ensuite reconstruire le résultat global.

- **Exemple :** On a une liste de nombres $L = (3\ \ 6{,}5\ \ 1\ \ -2\ \ 0\ \ 7)$ dont on cherche le minimum.

  - Supposons qu'on connaisse le minimum m de la liste privée de son premier élément, ici $m = -2$.

  - Pour connaître le minimum de la liste L, il suffit alors de comparer ce minimum m avec le premier élément, et ne garder que le plus petit.

- **Question 1: Comment trouve-t-on le minimum pour la sous-liste ?**
  - En faisant le même raisonnement ! On va appeler notre fonction sur la sous-liste de celle-ci où on oublie le premier élément.

- **Question 2: Quand s'arrête-t-on ?**
  - Quand on ne peut plus parler de sous-liste, donc quand la liste a un seul élément. On parle de **cas d'arrêt**.

  - Génial, puisque dans ce cas particulier d'une liste à un seul élément, connaître le minimum est facile : c'est forcément cet élément ! On peut donc renvoyer quelque chose dans ce cas.

- Voici dont la fonction écrite en langage naturel:

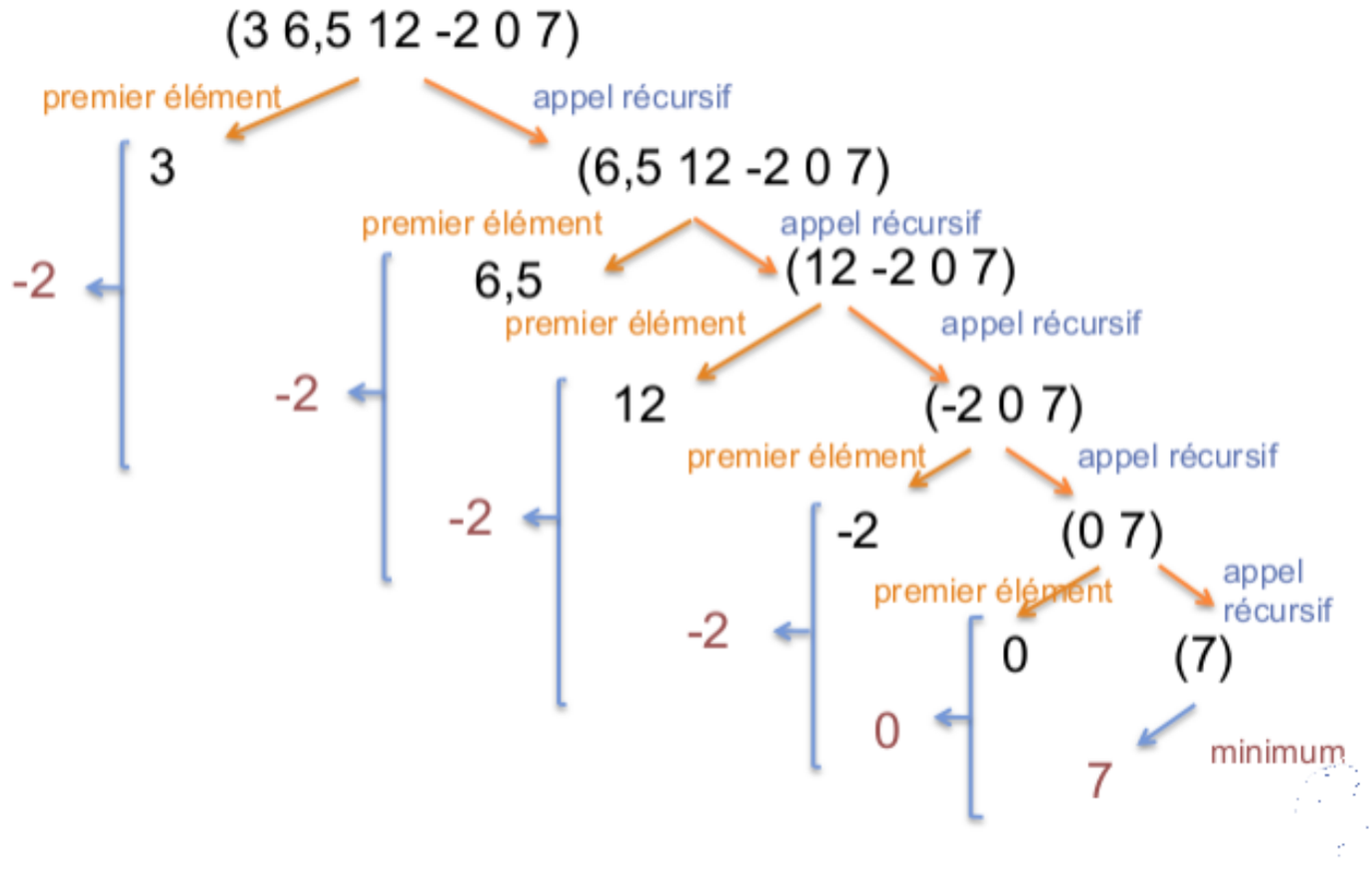
```
Définition de la fonction minimum (L)

Si vide?(reste(L)) Alors
        retourne premier(L)
Sinon
        Si premier(L) < minimum(reste(L)) Alors
                retourne premier(L)
        Sinon
                retourne minimum(reste(L))
        FinSi
FinSi
```

- La première ligne représente le **cas d'arrêt**: si la liste ne contient qu'un élément, autrement dit que son reste est vide (ce qui est plus simple à tester), on retourne le premier élément.

- L'instruction **minimum(reste(L))** constitue **l'appel récursif** de la fonction minimum sur la liste reste(L).

Récursivité

- **Illustration :**



- Pour écrire une fonction récursive, il faut donc (dans cet ordre).
  - 1) Choisir sur quelle donnée plus petite faire l'appel récursif. En pratique, pour un entier n ce sera $n-1$; pour une liste L, ce sera L moins son premier élément etc.
  - 2) Déterminer comment passer du résultat de l'appel récursif au résultat cherché.
  - 3) Identifier le cas d'arrêt.
- C'est ici ce qu'on appelle de la récursivité **en remontant**. Nous verrons plus tard un autre type de récursivité.

Quelques fonctions récursives

- **Minimum d'une liste :**

```
#lang racket
(define minimum
  (lambda (L); L une liste -> renvoie un entier
    (if (null? (cdr L))
        (car L)
        (min (car L) (minimum (cdr L))))))
```

  - Il reste ici des fonctions mystérieuses, notamment **car** et **cdl**, que nous verrons plus tard...

- **Factorielle :** On calcule ici, pour en entier n, la quantité $n! := 1 \times 2 \times \cdots \times n$. Par convention, $0! = 1$.

```
#lang racket
(define facto
  (lambda (n); n un entier -> renvoie un nombre entier
    (if (= 0 n)
        1
        (* n (facto (- n 1))))))
```

Mémorisation

- Il est parfois nécessaire d'utiliser un même calcul ou une même fonction avec les mêmes arguments dans une expression : pour éviter l'exécution redondante d'appels de fonctions, il est possible d'utiliser des variables locales, liant les expressions entre elles.

- **Exemple :** Si on veut calculer $\sqrt{x^2+1}+\sqrt{x^2+2}$, on peut écrire

```
#lang racket
(define f
  (lambda (x)
    (+ (sqrt (+ 1 (* x x))) (sqrt (+ 2 (* x x))))))
```

- Toutefois, on calcule la quantité **(* x x)** deux fois, alors qu'on aurait pu le faire qu'une fois. La fonction **let** permet de définir une variable équivalente à ces appels et précise à l'utilisateur qu'il suffit de l'exécuter une seule fois et réutiliser la ou les valeurs retournées.

Mémorisation

- La syntaxe de **let** est la suivante:

```
(let (
       (ident1 val1)
       (ident2 val2)
       ...
       (identN valN)
     )
  corps
)
```

```
#lang racket
(define f
  (lambda (x)
    (let ((x2 (* x x)))
      (+ (sqrt (+ 1 x2)) (sqrt (+ 2 x2))))))
```

- **Remarques :**
  - Les parenthèses rouges à l'intérieur du **let** (séparant les différentes mémorisations) peuvent être remplacées par des crochets.
  - Il existe d'autres fonctions telles que **let*** et **letrec** héritées de Scheme, ou encore la fonction **local** de Racket; que nous utiliserons moins. Vous pouvez regarder la documentation.