Programmation Fonctionnelle

Séance 1

Alexandros Singh

Université Paris 8

1er octobre 2023

Exemple d'algorithme récursif

Problème

Trouver le minimum d'une liste de nombres entiers positifs.

Penser de manière récursive :

- Cas de base : le minimum d'une liste d'un élément est l'unique élément.
- Appel récursif : Pour une liste de taille au moins égale à 2, le minimum se trouve soit en tête de liste, soit en queue de liste.

Problème

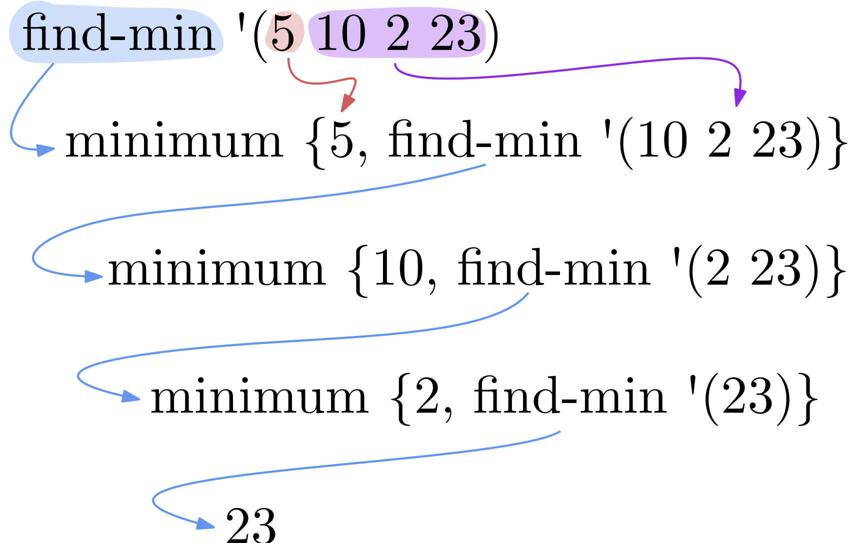
Trouver le minimum d'une liste de nombres entiers positifs.

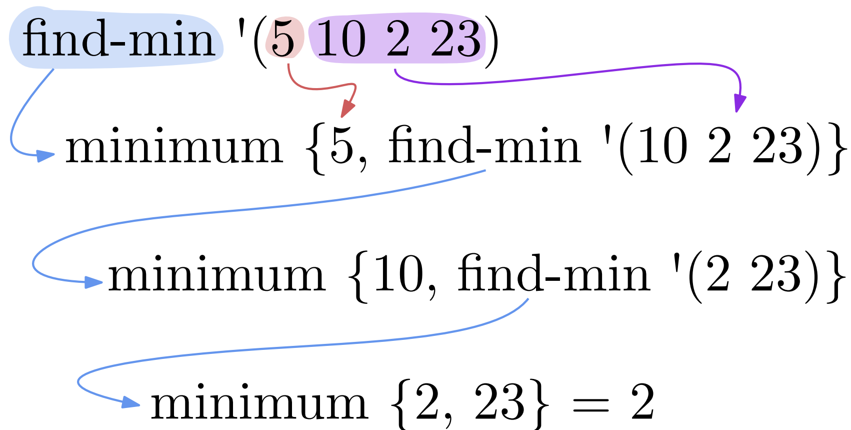
Penser de manière récursive :

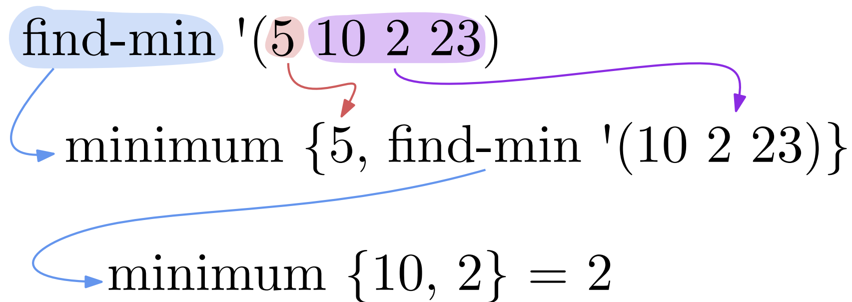
- Cas de base : le minimum d'une liste d'un élément est l'unique élément.
- Appel récursif : Pour une liste de taille au moins égale à 2, le minimum se trouve soit en tête de liste, soit en queue de liste.

L'exécution de cet algorithme se fait en deux phases :

- La phase descendante : Appels récursifs jusqu'à ce que nous arrivions à la liste vide. vide. Chacun de ces appels entraîne la création d'un nouveau cadre dans la pile d'appels.
- La phase remontante : Nous calculons le résultat pour le cas de base et puis nous l'utilisons pour calculer le résultat du dernier appel effectué. On continue ainsi, en dépilant la pile, jusqu'au premier appel qui donne le résultat final.


find-min '(5 10 2 23)
minimum {5, find-min '(10 2 23)}
minimum {10, find-min '(2 23)}
minimum {2, find-min '(23)}
23


find-min '(5 10 2 23)
minimum {5, find-min '(10 2 23)}
minimum {10, find-min '(2 23)}
minimum {2, 23} = 2


find-min '(5 10 2 23)
minimum {5, find-min '(10 2 23)}
minimum {10, 2} = 2

find-min '(5 10 2 23)

$\text{minimum } \{5, 2\} = 2$

$$\text{find-min '(5 10 2 23)} = 2$$

Pour implémenter notre algorithme en Racket, nous avons besoin des fonctions `car`, `cdr`, `null?` que nous avons déjà vues.
Nous avons aussi besoin d'une fonction qui joue le rôle de *minimum* : Racket fournit une fonction `min` qui prend un nombre arbitraire de nombres (réels) et donne le minimum de ses arguments.

```
(define find-min (lambda (l)
    (if (null? (cdr l))
        (car l)
        (min (car l) (find-min (cdr l))))))
```

Chaque nouvel appel de fonction nécessite l'ajout d'un nouveau cadre de pile (stack frame) à la pile d'appels (call stack).

find-min '(5 10 2 23)

minimum {5, find-min '(10 2 23)}

minimum {10, find-min '(2 23)}

minimum {2, find-min '(23)}

23

Pile d'appels

find-min '(5 10 2 23)

Chaque nouvel appel de fonction nécessite l'ajout d'un nouveau cadre de pile (stack frame) à la pile d'appels (call stack).

find-min '(5 10 2 23)

minimum {5, find-min '(10 2 23)}

minimum {10, find-min '(2 23)}

minimum {2, find-min '(23)}

23

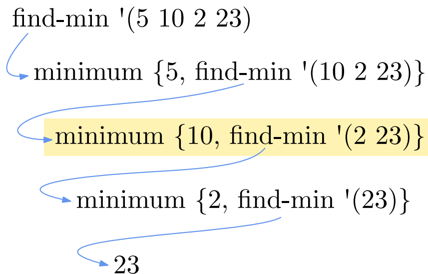
Pile d'appels

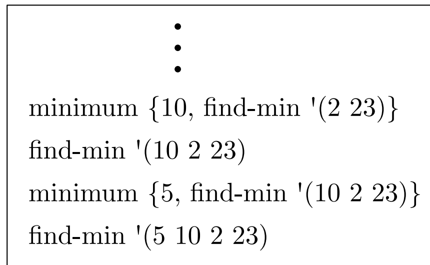
find-min '(10 2 23)

minimum {5, find-min '(10 2 23)}

find-min '(5 10 2 23)

Chaque nouvel appel de fonction nécessite l'ajout d'un nouveau cadre de pile (stack frame) à la pile d'appels (call stack).





Cependant, si nous pouvions garantir que chaque nouvel appel est uniquement vers `find-min`, nous pourrions garder un seule cadre dans la pile et modifier son contenu à chaque nouvel appel.

Récursion Terminale

Une fonction à récursivité terminale est une fonction où la dernière instruction à être évaluée est l'appel récursif.

Notre fonction `find-min` est-elle récursive ?

```
1 (define find-min (lambda (l)
2     (if (null? (cdr l))
3         (car l)
4         (min (car l) (find-min (cdr l))))))
```

Notre fonction `find-min` est-elle récursive ?

```
(define find-min (lambda (l)
    (if (null? (cdr l))
        (car l)
        (min (car l) (find-min (cdr l))))))
```

Non, car le dernier appel qu'elle fait est vers `min`, pas vers elle-même.
Cependant, il est impossible de trouver le minimum d'une liste sans comparer ses éléments - nous ne pouvons donc pas nous débarrasser complètement de `min` !

La solution consiste à introduire une nouvelle variable, un accumulateur, pour stocker les résultats partiels.
Cette technique est fréquemment utilisée avec les fonctions récursives terminales.

```
define find-min-tr (lambda (l acc) ...)
```

Où `acc` est notre accumulateur, qui contient le minimum que nous avons trouvé jusqu'à présent.

Comment devrions-nous réécrire notre cas de base et notre appel récursif ?

- Cas de base : le minimum d'une liste vide est ...
- Appel récursif : pour une liste non vide ...

Comment devrions-nous réécrire notre cas de base et notre appel récursif ?

- Cas de base : le minimum d'une liste vide est ce qui se trouve dans l'accumulateur.
- Appel récursif : pour une liste non vide ...

Comment devrions-nous réécrire notre cas de base et notre appel récursif ?

- Cas de base : le minimum d'une liste vide est ce qui se trouve dans l'accumulateur.
- Appel récursif : pour une liste non vide ...
  - Nous mettons à jour l'accumulateur : sa nouvelle valeur est le minimum entre la tête de liste et la valeur actuellement stockée.
  - Nous continuons à explorer le reste de la liste de manière récursive.

Implémentation en Racket :

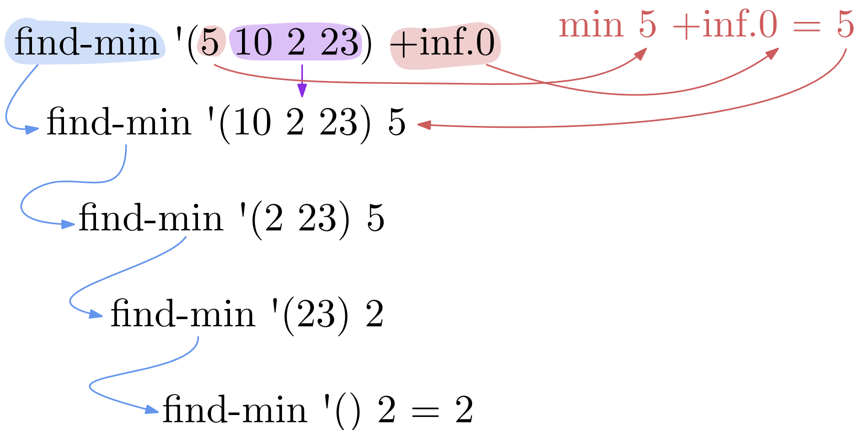
```
(define find-min-tr (lambda (l acc)
    (if (null? l)
        acc
        (find-min-tr (cdr l) (min (car l) acc)))))
```

Il reste un dernier détail à régler : pour appeler cette fonction, nous devons initialiser l'accumulateur. Quelle est la valeur appropriée pour ce faire ?

Implémentation en Racket :

```
1 (define find-min-tr (lambda (l acc)
2     (if (null? l)
3         acc
4         (find-min-tr (cdr l) (min (car l) acc)))))
```

Il reste un dernier détail à régler : pour appeler cette fonction, nous devons initialiser l'accumulateur. Quelle est la valeur appropriée pour ce faire ? La constante +inf !


find-min '(5 10 2 23) +inf.0
min 5 +inf.0 = 5
find-min '(10 2 23) 5
find-min '(2 23) 5
find-min '(23) 2
find-min '() 2 = 2

find-min '(5 10 2 23) +inf.0

find-min '(10 2 23) 5

find-min '(2 23) 5

find-min '(23) 2

find-min '() 2 = 2

Pile d'appels

find-min '(5 10 2 23) +inf.0

Récursion Terminale

find-min '(5 10 2 23) +inf.0

find-min '(10 2 23) 5

find-min '(2 23) 5

find-min '(23) 2

find-min '() 2 = 2

Pile d'appels

find-min '(10 2 23) 5

find-min '(5 10 2 23) +inf.0

find-min '(10 2 23) 5

find-min '(2 23) 5

find-min '(23) 2

find-min '() 2 = 2

Pile d'appels

find-min '(2 23) 5

Récursion Terminale

find-min '(5 10 2 23) +inf.0

find-min '(10 2 23) 5

find-min '(2 23) 5

find-min '(23) 2

find-min '() 2 = 2

Pile d'appels

find-min '(23) 2

Remarquez que, contrairement à la version précédente, il n'est pas nécessaire de dépiler pour "remonter". Comme il n'y a qu'un seul élément dans la pile à un moment donné, une fois l'appel final effectué, le résultat est renvoyé.

```
find-min '(5 10 2 23) +inf.0
 find-min '(10 2 23) 5
  find-min '(2 23) 5
   find-min '(23) 2
    find-min '() 2 = 2
```

Pile d'appels

```
find-min '() 2
```

Benchmark

Nous obtenons donc des programmes plus rapides et qui conservent de l'espace dans la pile d'appels :

```
> (define big-list (shuffle (range 10e6)))
> (time (find-min big-list))
cpu time: 961 real time: 964 gc time: 793
0
> (time (find-min-tr big-list +inf.f))
cpu time: 308 real time: 309 gc time: 201
0.0
```