Programmation Fonctionnelle

Séance 0

Alexandros Singh

Université Paris 8

21 septembre 2023

La programmation fonctionnelle est généralement décrite comme un paradigme de programmation qui met l'accent sur les éléments suivants :

- Les fonctions comme des données qui peuvent être liées à des identifiants, passées comme arguments, renvoyées par d'autres fonctions, etc.
- Construction de programmes par l'application et la composition de fonctions.

Pour s'adapter à ce paradigme, les langages de programmation fonctionnels possèdent certaines des caractéristiques suivantes :

- Ils favorisent les fonctions pures : celles qui ne dépendent pas de l'état du programme et qui n'ont pas d'effets de bord. Leur seul but est de renvoyer des valeurs, pas modifier des variables, de changer la mémoire, etc.
- Au lieu de boucles (qui utilisent des variables mutables), ils favorisent les fonctions récursives : fonctions définies en termes d'elles-mêmes.
- Les systèmes de typage : certains langages (par exemple OCaml que vous verrez l'année prochaine) sont équipés de systèmes de types statiques qui permettent d'éviter certaines erreurs avant l'exécution du programme. Racket est un langage non typé (avec "typage dynamique", ou même "unityped") et les erreurs ne sont détectées que pendant l'exécution.

Problème

Calculez la somme des éléments d'une liste donnée.

Algorithme itératif, avec des effets de bord

Idée : Parcourir la liste à l'aide d'une boucle, calculer et conserver les sommes partielles sur une variable que nous mettons à jour à chaque étape de la boucle, retourner cette variable.

Algorithme récursif, pas d'effets de bord

Nous ne pouvons pas utiliser de boucles et nous ne pouvons pas stocker les produits partiels.

Problème

Calculez la somme des éléments d'une liste donnée.

Algorithme itératif, avec des effets de bord

Idée : Parcourir la liste à l'aide d'une boucle, calculer et conserver les sommes partielles sur une variable que nous mettons à jour à chaque étape de la boucle, retourner cette variable.

Algorithme récursif, pas d'effets de bord

Idée :

- Pour les listes vides, la somme est de 0.
- La somme des éléments d'une liste est le premier élément plus la somme des éléments du reste de la liste.

Problème

Calculez la somme des éléments d'une liste donnée.

Algorithme itératif, avec des effets de bord

Idée : Parcourir la liste à l'aide d'une boucle, calculer et conserver les sommes partielles sur une variable que nous mettons à jour à chaque étape de la boucle, retourner cette variable.

Algorithme récursif, pas d'effets de bord

Idée :

- Cas de base : Pour les listes vides, la somme est de 0.
- Récursion : La somme des éléments d'une liste est le premier élément plus la somme des éléments du reste de la liste.

Notez ce motif de cas de base/recursion !

Problème

Calculez la somme des éléments d'une liste donnée.

Algorithme itératif, avec des effets de bord

Pseudocode :

```
def somme(l):
  sommePartielle = 0
  for i from 0 to length(l):
    sommePartielle += l[i]
  return sommePartielle
```

Algorithme récursif, pas d'effets de bord

Pseudocode :

```
def somme(l):
  if isEmpty(l):
    return 0
  else:
    return head(l)+somme(tail(l))
```

- Les fonctions pures sont référentiellement transparentes : nous pouvons remplacer leurs invocations par les valeurs résultantes sans affecter le programme. Cela facilite la mémoïsation.
- Les appels indépendants à des fonctions pures sont “thread-safe”, et peuvent être effectués dans *n'importe quel ordre (ou même en parallèle)*.
- Les techniques fonctionnelles peuvent être utilisées en dehors des langages purement fonctionnels.
- Les systèmes de types, quand ils sont présents, permettent de raisonner sur les programmes (vérifier leur validité, les optimiser, etc.).
- Liens avec l'informatique théorique et la logique (théorie de la preuve, théorie des types, sémantique catégorique, etc.)
- C'est sympa !

- Racket est un dialecte de Lisp et possède de nombreuses variantes telles que Typed Racket et Lazy Racket. [1]
- Sa syntaxe est minimale :

```
> (println "Hello world!") ;les parenthèses servent à délimiter
"Hello world!"
> (+ 2 2) ;la notation préfixe est utilisée
4
> (define x 5) ;liaison des identificateurs aux valeurs via define
> (+ 3 (+ x 2))
10
```

1. N'oubliez donc pas d'inclure `#lang racket` dans vos programmes pour spécifier que vous travaillez en Racket standard

Plus sur les fonctions

- Les fonctions anonymes sont définies comme suit :

```
> (lambda (x y) (+ x y))
```

- L'application se fait en écrivant d'abord la fonction (notation préfixe), puis ses arguments

```
> ((lambda (x y) (+ x y)) 2 2)
4
```

- Comme les fonctions sont des données, nous pouvons les lier à des identificateurs pour créer des fonctions nommées :

```
> (define myAdd (lambda (x y) (+ x y)))
> (myAdd 10 20)
30
```

Plus sur les fonctions

- On peut également écrire :

```
> (define (myAdd x y) (+ x y))
> (myAdd 10 20)
30
```

- Encore une fois, les fonctions sont des données ! Nous pouvons les passer comme arguments. Par exemple, voici une fonction `myCompose` qui prend deux fonctions `f` et `g`, les compose et les applique à `x` :

```
> (define myCompose (lambda (f g x) (f (g x))))
> (myCompose (lambda (x) (+ x 3)) (lambda (x) (* x 2)) 5)
13
> (+ (* 5 2) 3) ;pour vérifier
13
```

Plus sur les fonctions

- Le corps des fonctions peut contenir plus d'une expression :

```
> (define greet (lambda (name)
    (println (string-append "Hi, " name))
    (println (string-append "Bye, " name))
  ))
> (greet "Alex")
"Hi, Alex"
"Bye, Alex"
```

- La valeur renvoyée est *la valeur de la dernière expression* :

```
> (define addTen (lambda (x)
    (+ x 20) ;évaluée, mais le résultat n'est jamais utilisé ailleurs
    (+ x 10)
  ))
> (addTen 5)
15
```

Liaisons, portée

Comme nous l'avons vu, une façon de lier des identifiants à des valeurs est :

```
(define id expr)
```

Un identifiant peut être écrit avec n'importe quel caractère Unicode, à l'exception des espaces blancs, des chaînes de chiffres uniquement et des caractères réservés suivants :

```
( ) [ ] { } " , ' ` ; # | \
```

Par convention, les id sont généralement écrits en minuscules, avec des traits d'union entre les mots [2] :

```
un-identifiant
un-identifiant-beaucoup-plus-long
```

2. Également connu sous le nom de "kebab case" !

Portée (scope)

La portée (scope) d'une liaison est la partie d'un programme où la liaison est valide.

Les liaisons `define` au toplevel sont globales :

```
> (define myConstant 5)
> (define addConstant (lambda (x) (+ x myConstant)))
> (addConstant 10)
15
```

`lambda` lie également les identificateurs. Les liaisons les plus internes sont prioritaires :

```
> (define x 10)
> ((lambda (x) (+ x 1)) 5)
6
> x ;ici x est toujours 10
10
```

Une autre façon de lier localement les identificateurs, qui nous donne plus de contrôle, est `let` :

```
(let ([id expr] ...) body ...+)
```

La portée d'un `id` dans `let` est `body`.

```
> (let ([me "Alexandros"]
        [myself "Alekos"]
        [I "Alex"])

        (list me myself I))
'("Alexandros" "Alekos" "Alex")
```

Nous pouvons mélanger les méthodes pour lier les identificateurs comme nous le souhaitons :

```
> (define f (lambda (x)
    (let ([y 10])
          (+ x y))))
> (f 5)
15
```

```
> (define my-constant 1)
> (define f (lambda (x)
   (let ([x my-constant])
         x)))
> (f 20)
1
```

Quote

Par défaut, les expressions sont évaluées :

```
> (+ 2 2)
4
```

Pour empêcher l'évaluation, nous pouvons utiliser `quote` ou faire précéder une expression par ' :

```
> (quote (+ 2 2))
'(+ 2 2)
> '(+ 2 2)
'(+ 2 2)
```

Ce concept est souvent utilisé dans le contexte de la métaprogrammation (pour laquelle la famille Lisp est célèbre).

Cons, car, cdr

Les paires, qui sont immutables, sont définies par `cons`

```
> (cons 1 2)
'(1 . 2)
```

`car` et `cdr` extraient respectivement le premier et le deuxième élément :

```
> (define a-pair (cons 1 2))
> (car a-pair)
1
> (cdr a-pair)
2
```

Listes

Une liste est définie récursivement :

- C'est soit la constante `null`,
- Soit une paire dont la deuxième valeur est une liste.

```
> null ; liste vide
'()
> (cons 1 null)
'(1)
> (cons 1 (cons 2 null)) ;attention: pas la même chose que (cons 1 2)
'(1 2)
> (cons 1 (cons 2 (cons 3 null)))
'(1 2 3)
```

`list` construit une liste de ses arguments :

```
>(list 1 2 3)
'(1 2 3)
```

Listes

Les listes étant constituées de paires, nous pouvons utiliser `car` pour obtenir leur premier élément et `cdr` pour obtenir le reste :

```
> (car (list 1 2 3))
1
> (cdr (list 1 2 3))
'(2 3)
```

Nous pouvons encore utiliser `cons` pour ajouter un élément au début d'une liste et `append` pour joindre plusiers listes :

```
> (define my-list (list 1 2 3))
> (cons 0 my-list)
'(0 1 2 3)
> (append my-list (list 4 5 6) (list 7 8 9 10))
'(1 2 3 4 5 6 7 8 9 10)
```

Booléens et conditionnels

Les constantes booléennes sont `#t, #f`. Quelques fonctions manipulant les booléens sont :

```
> (not #t) ;inversion booléenne
#f
> (and #t #f) ; conjonction booléenne
#f
> (or #t #f) ; disjonction booléenne
#t
```

Syntaxe pour `if` :

```
(if expression-à-tester
    expression-si-vrai
    expression-si-faux)
```

Dans les conditionnelles, toutes les valeurs non-`#f` sont traitées comme vraies :

```
> (if "blabla" 0 1)
0
```

Pour les chaînes de conditionnelles, utilisez `cond` :

```
> (cond ((< 2 0) (println "Deux est négatif"))
        ((= 2 0) (println "Deux est égal à zéro"))
        (else (println "Deux est plus grand que zéro")))
"Deux est plus grand que zéro"
```

Somme des éléments d'une liste en Racket

Nous pouvons maintenant combiner tout ce que nous avons appris pour écrire l'algorithme en Scheme :

```
> (define somme (lambda (l)
    (if (null? l) 0
        (+ (car l) (somme (cdr l))))))
> (somme (list 1 2 3 4 5))
15
```

où `null?` donne `#t` si une liste est vide, sinon `#f`. Vous voyez comme c'est élégant ? Remarquez, aussi, que l'algorithme reflète la définition récursive des listes avec ses deux cas.
Nous verrons beaucoup d'autres exemples de ce schéma dans la suite du cours...