

# Paradigmes et Interprétation

## Objets et classes

Julien Provillard

[julien.provillard@univ-cotedazur.fr](mailto:julien.provillard@univ-cotedazur.fr)

# OBJETS

# Introduction

- ❑ Le point sur la **programmation fonctionnelle**.
- ❑ La programmation fonctionnelle n'utilise pas d'**état**.

```
(define (area [s : Shape]) : Number  
  ... )
```

```
(define r (rectangle 10 20))
```

```
...  
(test (area r) 200)  
...  
(test (area r) 200)  
...
```

On est assuré que les  
résultats seront identiques.

Alternative : **programmation impérative**

# Introduction

- ❑ Le point sur la **programmation fonctionnelle**.
- ❑ La programmation fonctionnelle dispose de fonctions d'**ordre supérieur**.

```
(define (compose [f : ('a -> 'b)] [g : ('b -> 'c)]) : ('a -> 'c)  
  (lambda (x) (g (f x))))
```

Alternative : **programmation du premier ordre**

# Introduction

- ❑ Le point sur la **programmation fonctionnelle**.
- ❑ La programmation fonctionnelle est usuellement une **programmation orientée type de données**.

```
(define-type Shape  
  [rectangle (width : Number) (height : Number)]  
  [square (side : Number)]  
  [circle (radius : Number)])
```

```
(define (area [s : Shape]) : Number  
  (type-case Shape s  
    [(rectangle w h) ... ]  
    [(square s) ... ]  
    [(circle r) ... ]))
```

Une fonction qui agit différemment selon la variante de la donnée qui lui est transmise.

# Introduction

- ❑ Le point sur la **programmation fonctionnelle**.
- ❑ La programmation fonctionnelle est usuellement une **programmation orientée type de données**.

```
(define (map [f : ('a -> 'b)] [l : (Listof 'a)]) : (Listof 'b)
  (cond
    [(empty? l) empty]
    [(cons? l) (cons (f (first l)) (map f (rest l)))]))
```

Une fonction qui agit différemment selon la variante de la donnée qui lui est transmise.

Alternative : **programmation orientée objet**

# Programmation orientée donnée vs orientée objet

## Orientée type de données :

❑ Une opération avec des variantes.

```
(define area [s : Shape] : Number
  (type-case Shape s
    [(rectangle w h) ...]
    [(square s) ...]
    [(circle r) ...]))

(define perimeter [s : Shape] : Number
  (type-case Shape s
    [(rectangle w h) ...]
    [(square s) ...]
    [(circle r) ...]))
```

## Orienté objet :

❑ Une opération sur une variante.

```
class Rectangle extends Shape {
  int area() { ... }
  int perimeter() { ... }
}

class Square extends Shape {
  int area() { ... }
  int perimeter() { ... }
}

class Circle extends Shape {
  int area() { ... }
  int perimeter() { ... }
}
```

# Programmation orientée donnée vs orientée objet

## Orientée type de données :

- ☐ Une opération avec des variantes.

```
(define area [s : Shape] : Number
  (type-case Shape s
    [(rectangle w h) ]
```

### Nouvelle opération :

- **Nouvelle** fonction

### Nouvelle variante :

- **Modification** des fonctions

```
[(rectangle w h) ...]
[(square s) ...]
[(circle r) ...]))
```

## Orienté objet :

- ☐ Une opération sur une variante.

```
class Rectangle extends Shape {
  int area() { ... }
  int perimeter() { ... }
```

### Nouvelle opération :

- **Modification** des objets

### Nouvelle variante :

- **Nouvel** objet

```
class Circle extends Shape {
  int area() { ... }
  int perimeter() { ... }
}
```



# Programmation orientée donnée vs orientée objet

- ☐ La programmation fonctionnelle peut être orientée type de données ou orientée objet.
- ☐ On peut représenter des objets par des fonctions.

# Des fonctions en tant qu'objets

```
(define-type-alias Shape (-> Number))
```

Représentation simpliste d'un objet

```
(define (rectangle w h) : Shape  
  (lambda () (* w h)))
```

Constructeurs d'objets

```
(define (square s) : Shape  
  (lambda () (* s s)))
```

```
(define r (rectangle 10 20))  
(r) --> 200
```

```
(define c : Shape  
  (let ([r 10])  
    (lambda () (* 3.1415927 (* r r)))))  
(c) --> 314.15927
```

On peut définir de nouveaux objets à la volée

# Objets avec plusieurs opérations

- ❑ Avec la représentation actuelle, les objets ne peuvent disposer que d'une seule opération.

```
(define-type-alias Shape (-> Number))
```

```
(define r (rectangle 10 20))  
(r) --> 200
```

- ❑ Si on veut disposer de plusieurs opérations, il faut pouvoir opérer une sélection.

```
(define-type-alias Shape (Symbol -> Number))
```

```
(define r (rectangle 10 20))  
(r 'area) --> 200  
(r 'perimeter) --> 60
```

# Objets avec plusieurs opérations

```
(define-type-alias Shape (Symbol -> Number))

(define (rectangle w h) : Shape
  (lambda (op)
    (cond
      [(equal? op 'area) (* w h)]
      [(equal? op 'perimeter) (* 2 (+ w h))])))

(define (square s) : Shape
  (lambda (op)
    (cond
      [(equal? op 'area) (* s s)]
      [(equal? op 'perimeter) (* 4 s)])))

(define r (rectangle 10 20))
(r 'area) --> 200
(r 'perimeter) --> 60
```

# Objets avec plusieurs opérations

- ❑ Et si les fonctions ont des signatures différentes ?
- ❑ Il faut utiliser une version non typée du langage :  
#lang plait #:untyped
- ❑ On va en profiter pour généraliser la représentation d'un objet.
- ❑ Un objet sera représenté par une liste associative entre le nom de ses fonctions et leur valeur.
- ❑ Par exemple, on pourrait représenter une forme par une liste :

```
(list (pair 'area f-area) ; f-area : (-> Number)
      (pair 'bigger-than? f-bigger-than?) ; f-bigger-than? : (Number -> Boolean)
      ... ) ; autres fonctions
```

# Objets avec plusieurs opérations

□ Comment récupérer la définition d'une fonction ?

```
(list (pair 'area f-area) ; f-area : (-> Number)
      (pair 'bigger-than? f-bigger-than?) ; f-bigger-than? : (Number -> Boolean)
      ... ) ; autres fonctions
```

```
(define (find l name)
  (cond
    [(empty? l) (error 'find "not found")]
    [(equal? (fst (first l)) name) (snd (first l))]
    [else (find (rest l) name)]))
```

# Objets avec plusieurs opérations

## □ Comment évaluer une fonction ?

```
(list (pair 'area f-area) ; f-area : (-> Number)
      (pair 'bigger-than? f-bigger-than?) ; f-bigger-than? : (Number -> Boolean)
      ... ) ; autres fonctions
```

```
(define (rectangle w h)
  (list (pair 'area (lambda () (* w h)))
        (pair 'bigger-than? (lambda (n) (> (* w h) n)))))
```

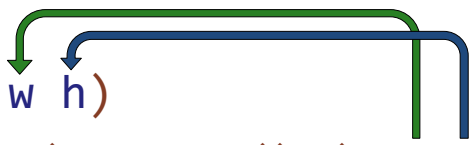
```
(define (square s)
  (list (pair 'area (lambda () (* s s)))
        (pair 'bigger-than? (lambda (n) (> (* s s) n)))))
```

```
(define r (rectangle 10 20))
((find r 'area)) --> 200
((find r 'bigger-than?) 100) --> #t
```

# Champs d'un objet

- ❑ Les valeurs stockées dans un objet sont ses **champs**, les fonctions qui lui sont associées sont ses **méthodes**.

```
(define (rectangle w h)
  (list (pair 'area (lambda () (* w h)))
        (pair 'bigger-than? (lambda (n) (> (* w h) n)))))
```



Les champs sont stockés dans chacune des clôtures lexicales, il serait plus raisonnable de les centraliser.

- ❑ Les objets formés à partir de rectangle ont `w` et `h` pour champs, et `area` et `bigger-than?` pour méthodes.
- ❑ Mais comment une méthode accède aux champs de l'objet ?
- ❑ Les deux expressions `lambda` définissent des fonctions internes à rectangles, `w` et `h` sont dans leur clôture lexicale.



## Champs d'un objet

- ☐ Tous comme les méthodes, les champs vont être stockés dans une liste associative. Un objet sera donc une paire de listes associatives.
- ☐ On va accéder aux champs d'un objet via une macro `get` qui aura le rôle d'un **accesseur**.
- ☐ On appellera une méthode d'un objet en lui **envoyant un message** via la macro `send`.
- ☐ Une méthode peut accéder aux **membres** (champs et méthodes) de l'objet qui la contient. Les méthodes possèdent un paramètre `this` qui référence l'objet englobant (comme en Python).
- ☐ Les constructeurs seront des macros pour empêcher la capture des champs dans les clôtures lexicales.

# Les objets 'soft' complets.

```
(define-syntax-rule (rectangle w-ini h-ini)
  (pair (list (pair 'w w-ini) (pair 'h h-ini))
        (list (pair 'area (lambda (this) (* (get this w) (get this h))))
              (pair 'bigger-than? (lambda (this n) (> (send this area) n))))))
```

```
(define-syntax-rule (get obj-expr field-id)
  (find (fst obj-expr) 'field-id))
```

```
(define-syntax-rule (send obj-expr method-id arg ...)
  (let ([obj obj-expr])
    ((find (snd obj) 'method-id) obj arg ...)))
```

```
(define r (rectangle 10 20))
(send r bigger-than? 100) --> #t
```



ordresup-oo.rkt

# Fonctions vs objets

- ☐ Pour l'instant, nous avons vu comment obtenir des objets dans un langage autorisant l'ordre supérieur.
- ☐ Cette démarche reste assez peu naturelle.
- ☐ Nous allons chercher à interpréter un langage orienté objet de manière classique sans utiliser les clôtures lexicales.

# Grammaire : couche objet

```
<Exp> ::= ...  
        | {object {<Field>*} <Method>*}  
        | {get <Exp> <Symbol>}  
        | {send <Exp> <Symbol> <Exp>}  
<Field> ::= [<Symbol> <Exp>]  
<Method> ::= [<Symbol> <Exp>]
```

Évalué à la création de l'objet.

Ici, `arg` et `this` font référence à l'objet appelant (s'il existe).

Évalué lorsque la méthode est appelée.

On simplifie en considérant que les méthodes ont un unique argument.

On peut faire référence à cet argument via le mot clé `arg`.

On peut faire référence à l'objet englobant via le mot clé `this`.

# Exemples

```
<Exp> ::= ...  
      | {object {<Field>*} <Method>*}  
      | {get <Exp> <Symbol>}  
      | {send <Exp> <Symbol> <Exp>}
```

```
<Field> ::= [<Symbol> <Exp>]
```

```
<Method> ::= [<Symbol> <Exp>]
```

```
{object  
  {[x 1] [y 2]}}
```

# Exemples

```

<Exp>      ::= ...
              | {object {<Field>*} <Method>*}
              | {get <Exp> <Symbol>}
              | {send <Exp> <Symbol> <Exp>}

<Field>    ::= [<Symbol> <Exp>]
<Method>   ::= [<Symbol> <Exp>]

{object
  {[x 1] [y {+ 1 1}]}}
```

# Exemples

```

<Exp>      ::= ...
              | {object {<Field>*} <Method>*}
              | {get <Exp> <Symbol>}
              | {send <Exp> <Symbol> <Exp>}

```

```

<Field>    ::= [<Symbol> <Exp>]

```

```

<Method>   ::= [<Symbol> <Exp>]

```

```

{get {object
    {[x 1] [y {+ 1 1}]}}
  x}
--> 1

```

# Exemples

```
<Exp>      ::= ...  
              | {object {<Field>*} <Method>*}  
              | {get <Exp> <Symbol>}  
              | {send <Exp> <Symbol> <Exp>}  
<Field>    ::= [<Symbol> <Exp>]  
<Method>   ::= [<Symbol> <Exp>]
```

```
{object  
  {}  
  [incr {+ arg 1}]}
```



# Exemples

```

<Exp>      ::= ...
              | {object {<Field>*} <Method>*}
              | {get <Exp> <Symbol>}
              | {send <Exp> <Symbol> <Exp>}

<Field>    ::= [<Symbol> <Exp>]
<Method>   ::= [<Symbol> <Exp>]

{send {object
    {[x 1] [y {+ 1 1}]}
    [dist {+ {get this x} {get this y}}]}
dist 0}
--> 3

```

# Objets vs fonctions

❑ On a vu que les fonctions permettaient de simuler les objets.

❑ Les objets peuvent-ils simuler les fonctions ?

```
{{lambda {x} x} 1}
```



```
{send {object  
  {}  
  [f arg]}  
f 1}
```

# Objets vs fonctions

- ❑ On a vu que les fonctions permettaient de simuler les objets.
- ❑ Les objets peuvent-ils simuler les fonctions ?

```
{{{lambda {x} {lambda {y} {+ x y}}}} 1} 2}
```



```
{send {send {object
  {}
  [fx {object
    {[x arg]}
    [fy {+ {get this x} arg}]}}}
fx 1}
fy 2}
```

Sauvegarde du contexte

Pour généraliser automatiquement,  
il faudrait faire de la compilation.

# Grammaire complète

```
<Exp> ::= this
        | arg
        | <Number>
        | {+ <Exp> <Exp>}
        | {* <Exp> <Exp>}
        | {object {<Field>*} <Method>*}
        | {get <Exp> <Symbol>}
        | {send <Exp> <Symbol> <Exp>}
<Field> ::= [<Symbol> <Exp>]
<Method> ::= [<Symbol> <Exp>]
```

# Exemples comparatifs

```
{send {object
  {[x 1] [y 2]}
  [dist {+ {get this x}
           {get this y}}]}
dist 0}
```

```
class Posn {
  ...
  int x, y;
  int dist() {
    return this.x + this.y;
  }
}

new Posn(1, 2).dist();
```

# Exemples comparatifs

```
{send {object
  {[x 1] [y 2] [z 3]}
  [dist {+ {get this x}
           {+ {get this y}
               {get this z}}}]
  [scal {+ {* {get this x}
              {get arg x}}
           {* {get this y}
              {get arg y}}}]}}

scal {object
  {[x 1] [y 2]}
  [dist {+ {get this x}
           {get this y}}]}}
```

```
class Posn { ... }

class Posn3D extends Posn {
  ...
  int z;
  int dist() {
    return this.x + this.y + this.z;
  }
  int scal(Posn p) {
    return this.x * p.x + this.y * p.y;
  }
}

new Posn3D(1, 2, 3).scal(new Posn(4, 5));
```

# Représentation

```
(define-type Exp
  [thisE]
  [argE]
  [numE (n : Number)]
  [plusE (l : Exp) (r : Exp)]
  [multE (l : Exp) (r : Exp)]
  [objE (fields : (Listof (Symbol * Exp)))
        (methods : (Listof (Symbol * Exp)))]
  [getE (object : Exp) (field : Symbol)]
  [sendE (object : Exp) (method : Symbol) (arg : Exp)])
```

# Représentation

```
(define-type Value  
  [numV (n : Number)]  
  [objV (fields : (Listof (Symbol * Value)))  
        (methods : (Listof (Symbol * Exp)))])
```



# Interpréteur : exemples

`interp : (Exp -> Value)`

## □ Expression arithmétique

```
(test (interp {+ 1 2})  
      (numV 3))
```

# Interpréteur : exemples

`interp : (Exp -> Value)`

## □ Objet

```
(test (interp {object {}})  
      (objV empty empty))
```

# Interpréteur : exemples

`interp : (Exp -> Value)`

## Objet

```
(test (interp {object {[x {+ 1 2}]}  
                  [incr {+ arg 1}]}})  
      (objV (list (pair 'x (numV 3)))  
            (list (pair 'incr (plusE (argE) (numE 1)))))))
```

# Interpréteur : exemples

`interp : (Exp -> Value)`

## □ Accès au champs

```
(test (interp {get {object {[x {+ 1 2}]}  
                      [incr {+ arg 1}]}  
          x}))  
(numV 3))
```

# Interpréteur : exemples

`interp : (Exp -> Value)`

## □ Appel de méthode

```
(test (interp {send {object {[x {+ 1 2}]}  
                    [incr {+ arg 1}]}  
            incr 3}))  
(numV 4))
```

# Interpréteur : exemples

`interp : (Exp -> Value)`

## ☐ Corps d'une méthode

```
(test (interp {+ arg 1})  
  ... )
```

- ☐ On a besoin de connaître les valeurs de `arg` et `this`.
- ☐ On peut mettre en place un environnement comme précédemment.
- ☐ Mais ce sont les seuls identificateurs qui existent, on peut juste ajouter deux paramètres à `interp` pour connaître leur valeur.

# Interpréteur : exemples

```
interp : (Exp Value Value -> Value)
```



## ❑ Corps d'une méthode

```
(test (interp {+ arg 1})  
  ... )
```

- ❑ On a besoin de connaître les valeurs de `arg` et `this`.
- ❑ On peut mettre en place un environnement comme précédemment.
- ❑ Mais ce sont les seuls identificateurs qui existent, on peut juste ajouter deux paramètres à `interp` pour connaître leur valeur.

# Interpréteur : exemples

`interp : (Exp Value Value -> Value)`

## □ Corps d'une méthode

```
(test (interp {+ arg 1})  
      (objV empty empty)  
      (numV 3))  
(numV 4))
```



# Interpréteur : exemples

```
interp : (Exp Value Value -> Value)
```

## □ Corps d'une méthode

```
(test (interp {get this x})  
      (objV (list (pair 'x (numV 3)))  
              empty)  
      (numV 0))  
(numV 3))
```

# Implémentation

```
(define (interp [e : Exp] [this-v : Value] [arg-v : Value]) : Value
  (type-case Exp e
    ...
    [(thisE) this-v]
    [(argE) arg-v]
    [(numE n) (numV n)]
    [(plusE l r) (num+ (interp l this-v arg-v) (interp r this-v arg-v))]
    [(multE l r) (num* (interp l this-v arg-v) (interp r this-v arg-v))]
    ... ))
```

# Implémentation

```
(define (interp [e : Exp] [this-v : Value] [arg-v : Value]) : Value
  (type-case Exp e
    ...
    [(objE fields methods)
     (objV (map (lambda (fd)
                  (pair (fst fd)
                        (interp (snd fd) this-v arg-v)))
                  fields)
            methods))]
    ... ))
```

# Implémentation

```
(define (interp [e : Exp] [this-v : Value] [arg-v : Value]) : Value
  (type-case Exp e
    ...
    [(getE obj fd)
     (type-case Value (interp obj this-v arg-v)
       [(objV fields methods) (find fd fields)]
       [else (error 'interp "not an object")])])
    ... ))
```

# Implémentation

```
(define (interp [e : Exp] [this-v : Value] [arg-v : Value]) : Value
  (type-case Exp e
    ...
    [(sendE obj mtd arg)
     (let ([obj-v (interp obj this-v arg-v)])
       (type-case Value obj-v
         [(objV fields methods)
          (interp (find mtd methods)
                   obj-v
                   (interp arg this-v arg-v))]
         [else (error 'interp "not an object")])]))
    ... ))
```

# Fonction de recherche

```
(define (find [name : Symbol]
              [pairs : (Listof (Symbol * 'a))]) : 'a
  (cond
    [(empty? pairs) (error 'find "not found")]
    [(equal? name (fst (first pairs))) (snd (first pairs))]
    [else (find name (rest pairs))]))
```

# CLASSES

# Terminologie

```
(define (numV n)
  (list (pair 'number (lambda () ... ))
        (pair 'apply (lambda (arg) ... ))))

(define (closV par body env)
  (list (pair 'number (lambda () ... ))
        (pair 'apply (lambda (arg) ... ))))
```

- ❑ Les résultats de `numV` et `closV` sont des **objets** (soft).
- ❑ Les fonctions `apply` et `number` sont des **méthodes**.
- ❑ Les identificateurs `n`, `par`, `body`, `env` sont des **champs**.
- ❑ Ensemble, les champs et les méthodes d'un objet sont ses **membres**.
- ❑ Les fonctions `numV` et `closV` sont des **constructeurs**  
... et s'approchent de la notion de **classe**.



# Classes

□ Les classes ont deux rôles principaux :

- La création dynamique d'objets

```
class Snake {  
    ...  
}  
new Snake("Jormungandr", 100);
```

- L'héritage

```
class Python extends Snake {  
    ...  
}
```

- Héritage des membres
- Appels statiques de méthodes (à bien différencier des appels de méthodes statiques)

# Classes : liaisons dynamiques et statiques

```
class Snake implements Animal {  
    ...  
    boolean isPredatorOf(Animal a) {  
        return a.getSpeed() <= this.speed  
            && a.getWeight() <= this.weight / 2;  
    }  
}  
  
class Python extends Snake {  
    ...  
    boolean isPredatorOf(Animal a) {  
        return this.canSuffocate(a)  
            && super.isPredatorOf(a);  
    }  
}
```

Appels dynamiques de méthodes, la méthode réellement appelée dépend de l'objet courant.

Appel statique de méthode, la méthode appelée est la méthode `isPredatorOf` de la classe `Snake`.

# Grammaires pour les classes avec appels statiques

```
<Class> ::= {class <Symbol> {<Field>*} <Method>*}  
<Field> ::= <Symbol>  
<Method> ::= [<Symbol> <Exp>]  
<Exp> ::= arg  
        | this  
        | <Number>  
        | {+ <Exp> <Exp>}  
        | {* <Exp> <Exp>}  
        | {new <Symbol> <Exp>*}  
        | {get <Exp> <Symbol>}  
        | {send <Exp> <Symbol> <Exp>}  
        | {ssend <Exp> <Symbol> <Symbol> <Exp>}
```

# Exemples d'utilisation

```
{class Posn
  {x y}
  [dist {+ {get this x}
           {get this y}}]
  [addDist {+ {send this dist 0}
             {send arg dist 0}}]]}

{send {new Posn 1 2}
  addDist
  {new Posn 3 4}}
```

```
class Posn {
  ...
  int x, y;
  int dist() {
    return this.x + this.y;
  }
  int addDist(Posn p) {
    return this.dist() + p.dist();
  }
}

new Posn(1, 2).addDist(new Posn(3, 4));
```

# Exemples d'utilisation

```
{class Posn ... }
```

```
{class Posn3D
  {x y z}
  [dist {+ {get this z}
           {ssend this Posn dist arg}}]
  [addDist {ssend this Posn addDist arg}]]}
```

```
{send {new Posn3D 1 2 3}
      addDist
      {new Posn 4 5}}
```

```
class Posn { ... }
```

```
class Posn3D extends Posn {
  int z;
  int dist() {
    return this.z + super.dist();
  }
  int addDist(Posn p) {
    return super.addDist();
  }
}
```

```
new Posn3D(1, 2, 3).addDist(new Posn(4, 5));
```

# Représentation des objets

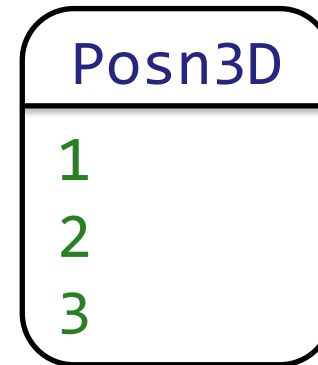
❑ Comment récupérer le code de la méthode `dist` lors de l'appel

```
{send {new Posn3D 1 2 3} dist 0}
```

❑ En fait, que doit être la valeur de `{new Posn3D 1 2 3}` ?

❑ Un objet qui contient désormais

- Une référence à sa classe
- La liste des valeurs de ses champs




❑ Les noms des champs et les méthodes sont recherchés directement dans la classe.

# Représentation des objets et des classes

```
(define-type Value  
  [numV (n : Number)]  
  [objV (class-name : Symbol)  
        (field-values : (Listof Value))])
```

```
(define-type Class  
  [classE (field-names : (Listof Symbol))  
          (methods : (Listof (Symbol * Exp)))]])
```

```
interp : (Exp (Listof (Symbol * Class)) Value Value -> Value)
```



La liste des classes définies

# Retour sur les exemples

```
(define Posn-class
  (pair 'Posn
    (classE
      (list 'x 'y)
      (list (pair 'dist (plusE (getE (thisE) 'x) (getE (thisE) 'y)))
        (pair 'addDist (plusE (sendE (thisE) 'dist (numE 0))
          (sendE (argE) 'dist (numE 0))))))))))
```

```
(define Posn3D-class
  (pair 'Posn3D
    (classE
      (list 'x 'y 'z)
      (list (pair 'dist (plusE (getE (thisE) 'z)
        (ssendE (thisE) 'Posn 'dist (argE))))
        (pair 'addDist (ssendE (thisE) 'Posn 'addDist (argE))))))))))
```



# Retour sur les exemples

□ On se donne une fonction enveloppe pour illustrer les tests.

```
(define (interp-posn e)
  (interp e (list Posn-class Posn3D-class) (objV 'Object empty) (numV 0)))

(test (interp-posn (newE 'Posn (list (numE 1) (numE 2))))
      (objV 'Posn (list (numV 1) (numV 2))))
```

# Retour sur les exemples

```
(define Posn-class
  (pair 'Posn
    (classE
      (list 'x 'y)
      (list (pair 'dist (plusE (getE (thisE) 'x) (getE (thisE) 'y)))
        (pair 'addDist (plusE (sendE (thisE) 'dist (numE 0))
          (sendE (argE) 'dist (numE 0))))))))))
```

```
(define Posn12 (newE 'Posn (list (numE 1) (numE 2))))
```

```
(test (interp-posn (sendE Posn12 'dist (numE 0)))
      (numV 3))
```

# Retour sur les exemples

```
(test (interp-posn (sendE Posn345 'addDist Posn12))  
      (numV 15))
```

# Implémentation

```
(define (interp [e : Exp] [classes : (Listof (Symbol * Class))]  
          [this-v : Value] [arg-v : Value]) : Value  
  (let ([interp-r (lambda (e) (interp e classes this-v arg-v))])  
    (type-case Exp e  
      ...  
      [(thisE) this-v]  
      [(argE) arg-v]  
      [(numE n) (numV n)]  
      [(plusE l r) (num+ (interp-r l) (interp-r r))]  
      [(multE l r) (num* (interp-r l) (interp-r r))]  
      ... )))
```

Fonction enveloppe pour les  
appels récurifs à `interp`.

# Implémentation

```
(define (interp [e : Exp] [classes : (Listof (Symbol * Class))]  
            [this-v : Value] [arg-v : Value]) : Value  
  (let ([interp-r (lambda (e) (interp e classes this-v arg-v))])  
    (type-case Exp e  
      ...  
      [(newE class-name fd-exprs)  
       (let ([c (find class-name classes)])  
         (if (= (length fd-exprs) (length (classE-field-names c)))  
             (objV class-name (map interp-r fd-exprs))  
             (error 'interp "wrong fields count")))]  
      ... )))
```

# Implémentation

```
(define (interp [e : Exp] [classes : (Listof (Symbol * Class))]  
            [this-v : Value] [arg-v : Value]) : Value  
  (let ([interp-r (lambda (e) (interp e classes this-v arg-v))])  
    (type-case Exp e  
      ...  
      [(getE obj fd)  
       (type-case Value (interp-r obj)  
         [(objV class-name fd-vals)  
          (let ([c (find class-name classes)])  
            (find fd (map2 pair  
                          (classE-field-names c)  
                          fd-vals)))]  
         [else (error 'interp "not an object")]])]  
      ... )))
```

# Implémentation

```

(define (interp [e : Exp] [classes : (Listof (Symbol * Class))]
  [this-v : Value] [arg-v : Value]) : Value
  (let ([interp-r (lambda (e) (interp e classes this-v arg-v))])
    (type-case Exp e
      ...
      [(sendE obj mtd arg)
       (let ([obj-v (interp-r obj)])
         (type-case Value obj-v
           [(objV class-name fd-vals)
            (call-method class-name mtd classes obj-v (interp-r arg))]
           [else (error 'interp "not an object")])])
      [(ssendE obj class-name mtd arg)
       (call-method class-name mtd classes (interp-r obj) (interp-r arg))]
      ... )))

```

# Implémentation

```
(define (call-method [class-name : Symbol]
                    [method-name : Symbol]
                    [classes : (Listof (Symbol * Class))]
                    [this-v : Value]
                    [arg-v : Value]) : Value
  (type-case Class (find class-name classes)
    [(class field-names methods)
     (interp (find method-name methods) classes this-v arg-v)]))
```



# Bilan

- ☐ Nous avons défini un langage avec classes.
- ☐ La création dynamique d'objets est possible via l'instruction `new`.
- ☐ On a une notion de sous-classe via l'utilisation de l'appel statique de méthode (`ssend`).
- ☐ En avons-nous fini ?
  
- ☐ Pour l'instant on simule l'héritage mais notre langage lui-même n'a pas de notion d'héritage !

# Héritage

□ Pour que la classe `Posn3D` soit une sous-classe de la classe `Posn`, nous avons dû écrire :

```
{class Posn
  {x y}
  [dist {+ {get this x} {get this y}}]
  [addDist {* {send this dist 0} {send arg dist 0}}]}
```

```
{class Posn3D
  {x y z}
  [dist {+ {get this z} {ssend this Posn dist arg}}]
  [addDist {ssend this Posn addDist arg}]}
```

Duplication des champs.

Réécriture des méthodes héritées

# Héritage

❑ On souhaiterait pouvoir écrire à la place :

```
{class Posn extends Object
  {x y}
  [dist {+ {get this x} {get this y}}]
  [addDist {* {send this dist 0} {send arg dist 0}}]]}
```

```
{class Posn3D extends Posn
  {z}
  [dist {+ {get this z} {super dist arg}}]]}
```

❑ Et l'utiliser de manière transparente :

```
{send {new Posn3D 1 2 3} addDist {new Posn 4 5}}
```

# Langage utilisateur vs langage interne

## ☐ Langage utilisateur

```

<Class> ::= {class <Symbol> extends <Symbol>
             {<Field>*}
             <Method>*}
<Field> ::= <Symbol>
<Method> ::= [<Symbol> <Exp>]
<Exp>    ::= arg
           | this
           | <Number>
           | {+ <Exp> <Exp>}
           | {* <Exp> <Exp>}
           | {new <Symbol> <Exp>*}
           | {get <Exp> <Symbol>}
           | {send <Exp> <Symbol> <Exp>}
           | {super <Symbol> <Exp>}

```

## ☐ Langage interne

```

<Class> ::= {class <Symbol>
             {<Field>*}
             <Method>*}
<Field> ::= <Symbol>
<Method> ::= [<Symbol> <Exp>]
<Exp>    ::= arg
           | this
           | <Number>
           | {+ <Exp> <Exp>}
           | {* <Exp> <Exp>}
           | {new <Symbol> <Exp>*}
           | {get <Exp> <Symbol>}
           | {send <Exp> <Symbol> <Exp>}
           | {ssend <Exp> <Symbol> <Symbol> <Exp>}

```

# Compilation

Changement des appels à `super`  
en appels statiques `ssend`.

```
{class Posn extends Object
  {x y}
  [dist {+ {get this x} {get this y}}]
  [addDist {* {send this dist 0} {send arg dist 0}}]}
```

```
{class Posn3D extends Posn
  {z}
  [dist {+ {get this z} {super dist arg}}]}
```



```
{class Posn
  {x y}
  [dist {+ {get this x} {get this y}}]
  [addDist {* {send this dist 0} {send arg dist 0}}]}
```

Fusion des méthodes non  
redéfinies de la classe mère.

```
{class Posn3D
  {x y z}
  [dist {+ {get this z} {ssend this Posn dist arg}}]
  [addDist {ssend this Posn addDist arg}]}
```

Fusion avec les champs de la classe mère.

# Représentation

```
(define-type ClassS
  [classS (super-name : Symbol)
          (field-names : (Listof Symbol))
          (methods : (Listof (Symbol * ExpS)))]])

(define-type ExpS
  [thisS]
  [argS]
  [numS (n : Number)]
  [plusS (l : ExpS) (r : ExpS)]
  [multS (l : ExpS) (r : ExpS)]
  [newS (class : Symbol) (field-values : (Listof ExpS))]
  [getS (object : ExpS) (field : Symbol)]
  [sendS (object : ExpS) (method : Symbol) (arg : ExpS)]
  [superS (method : Symbol) (arg : ExpS)])
```

# Du langage utilisateur au langage interne

```
(define (exp-s->e [e : ExpS] [super-name : Symbol]) : Exp
  (let ([aux (lambda (e) (exp-s->e e super-name))])
    (type-case ExpS e
      [(thisS) (thisE)]
      [(argS) (argE)]
      [(numS n) (numE n)]
      [(plusS l r) (plusE (aux l) (aux r))]
      [(multS l r) (multE (aux l) (aux r))]
      [(newS c fd-vals) (newE c (map aux fd-vals))]
      [(getS o fd) (getE (aux o) fd)]
      [(sendS o mtd arg) (sendE (aux o) mtd (aux arg))]
      [(superS mtd arg) (ssendE (thisE) super-name mtd (aux arg))])))
```

# Du langage utilisateur au langage interne

```
(define (class-s->e [c : (Symbol * ClassS)]) : (Symbol * Class)
  (type-case ClassS (snd c)
    [(classS super-name fds mtds)
     (pair (fst c)
            (classE fds
                     (map (lambda (mtd)
                           (pair (fst mtd) (exp-s->e (snd mtd) super-name)))
                           mtds))))]))
```



# Compiler une classe

- ❑ On sait passer d'une classe du langage utilisateur vers une classe du langage interne.
- ❑ Il faut au préalable compiler les classes utilisateurs :
  - Fusionner les champs de la classe mère avec ceux de la classe courante.
  - Fusionner les méthodes qui n'ont pas été redéfinies.
- ❑ Mais ceci ne fonctionne que si la classe mère est elle-même déjà compilée !
- ❑ Nous allons supposer que c'est le cas et nous verrons comment l'assurer ultérieurement.

# Compiler une classe

```
(define (compile-class  
  [raw : (Symbol * ClassS)]  
  [compiled : (Listof (Symbol * ClassS))]) : (Listof (Symbol * ClassS))  
  ...  
  ... )
```

Classe à compiler

Renvoie la classe compilée ajoutée à la liste.

...

Liste des classes précédemment compilées,  
Contient par hypothèse la classe mère de la  
classe à compiler.

... )

# Compiler une classe

```
(define (compile-class
  [raw : (Symbol * ClassS)]
  [compiled : (Listof (Symbol * ClassS))]) : (Listof (Symbol * ClassS))
(type-case ClassS (snd raw)
  [(classS super-name fds mtds)
   (type-case ClassS (find super-name compiled)
     [(classS super-super-name super-fds super-mtds)
      (let ([c (classS super-name
                       (merge-fields fds super-fds)
                       (merge-methods mtds super-mtds))])
        (cons (pair (fst raw) c) compiled)))]))])
```

# Compiler une classe

❑ Pour fusionner les champs, il suffit de concaténer les listes associées.

```
(define (merge-fields fds super-fds) (append super-fds fds)) ; ordre important
```

❑ Pour fusionner les méthodes, on ajoute un appel à la méthode de la classe mère si elle n'est pas redéfinie dans la classe fille.

```
(define (merge-methods mtds super-mtds)
  (let* ([defined (map fst mtds)] ; méthodes de la classe fille brute
        [super-defined (map fst super-mtds)] ; méthodes de la classe mère compilée
        [not-redefined (filter (lambda (mtd-name) ; méthodes non redéfinie
                                (not (member mtd-name defined)))
                                super-defined)]
        [inherited-mtds (map (lambda (mtd-name) ; donc héritée via (super mtd arg)
                              (pair mtd-name (superS mtd-name (argS))))
                              not-redefined)])
    (append mtds inherited-mtds))) ; ordre indifférent
```

# Compiler les classes

- ❑ Pour compiler les classes, on les compile une par une en se servant des classes précédemment compilées. Puis, on passe au langage interne.

```
(define (compile-classes
  [raws : (Listof (Symbol * ClassS))]) : (Listof (Symbol * Class))
  (let ([compiled (foldl compile-class
    (list (pair 'Object (classS 'Object empty empty))
    raws)])
    (map class-s->e compiled)))
```

- ❑ Mais on doit assurer que les classes se compilent dans le bon ordre !
- ❑ Il faut réaliser un **tri topologique** sur la liste des classes avant de les compiler. C'est un tri qui assure qu'une classe mère apparaît toujours avant ses classes filles.
- ❑ Cela revient à fait un parcours (en profondeur ou en largeur) de la hiérarchie des classes.

# Compiler les classes

```
(define (topological-sort
  [from : (Symbol * ClassS)] ; racine du tri
  [classes : (Listof (Symbol * ClassS))]) ; liste de toutes les classes
: (Listof (Symbol * ClassS)) ; renvoie le tri topologique à partir de la racine
(let* ([children ; extraction des classes filles
      (filter (lambda (c)
                (equal? (classS-super-name (snd c))
                        (fst from)))
              classes)]
      [children-lists ; tri topologique récursif sur les classes filles
      (map (lambda (child) (topological-sort child classes))
           children)])
  (cons from (foldl append empty children-lists)))) ; agrégations des tris
```

# Compiler les classes

□ Pour compiler les classes, on les compile une par une en se servant des classes précédemment compilées. Puis, on passe au langage interne.

```
(define (compile-classes
  [raws : (Listof (Symbol * ClassS))]) : (Listof (Symbol * Class))
  (let* ([sorted (topological-sort
    (pair 'Object (classS 'Object empty empty))
    raws)]
    [compiled (foldl compile-class
      (list (first sorted)) ; Object pré-compilé
      (rest sorted))])
    (map class-s->e compiled)))
```