

Paradigmes et Interprétation

Environnement et ordre supérieur

Julien Provillard

julien.provillard@univ-cotedazur.fr

LIAISON LOCALE

Langage actuel

```
<Exp> ::= <Number>  
        | <Symbol>  
        | {+ <Exp> <Exp>}  
        | {* <Exp> <Exp>}  
        | {<Symbol> <Exp>}
```

Grammaire sous forme de Backus-Naur

□ Implémentation dans le type `Exp`.

```
(define-type Exp  
  [numE (n : Number)]  
  [idE (s : Symbol)]  
  [plusE (l : Exp) (r : Exp)]  
  [multE (l : Exp) (r : Exp)]  
  [appE (fun : Symbol) (arg : Exp)])
```

Identificateurs locaux

```
<Exp> ::= <Number>  
      | <Symbol>  
      | {+ <Exp> <Exp>}  
      | {* <Exp> <Exp>}  
      | {<Symbol> <Exp>}  
      | {let {[<Symbol> <Exp>]} <Exp>} New !
```

Identificateurs locaux

```
<Exp> ::= <Number>
        | <Symbol>
        | {+ <Exp> <Exp>}
        | {* <Exp> <Exp>}
        | {<Symbol> <Exp>}
        | {let {[<Symbol> <Exp>]} <Exp>}
```

```
{let {[x {+ 1 2}]}
  {+ x x}}
--> 6
```

Identificateurs locaux

```

<Exp> ::= <Number>
        | <Symbol>
        | {+ <Exp> <Exp>}
        | {* <Exp> <Exp>}
        | {<Symbol> <Exp>}
        | {let {[<Symbol> <Exp>]} <Exp>}
  
```

```

{+ {let {[x {+ 1 2}]}}
  {+ x x}}
1}
--> 7
  
```

Identificateurs locaux

```

<Exp> ::= <Number>
        | <Symbol>
        | {+ <Exp> <Exp>}
        | {* <Exp> <Exp>}
        | {<Symbol> <Exp>}
        | {let {[<Symbol> <Exp>]} <Exp>}
  
```

```

{+ {let {[x {+ 1 2}]}}
  {+ x x}}
{let {[x {+ 3 4}]}}
  {+ x x}}
--> 20
  
```

Identificateurs locaux

```

<Exp> ::= <Number>
        | <Symbol>
        | {+ <Exp> <Exp>}
        | {* <Exp> <Exp>}
        | {<Symbol> <Exp>}
        | {let {[<Symbol> <Exp>]} <Exp>}
  
```

```

{+ {let {[x {+ 1 2}]}}
  {+ x x}}
{let {[y {+ 3 4}]}}
  {+ y y}}
--> 20
  
```


Identificateurs locaux

```
<Exp> ::= <Number>
        | <Symbol>
        | {+ <Exp> <Exp>}
        | {* <Exp> <Exp>}
        | {<Symbol> <Exp>}
        | {let {[<Symbol> <Exp>]} <Exp>}
```

```
{let {[x {+ 1 2}]}}
  {let {[x {+ 3 4}]}}
    {+ x x}}
--> 14
```

Identificateurs locaux

```
<Exp> ::= <Number>
        | <Symbol>
        | {+ <Exp> <Exp>}
        | {* <Exp> <Exp>}
        | {<Symbol> <Exp>}
        | {let {[<Symbol> <Exp>]} <Exp>}
```

```
{let {[x {+ 1 2}]}
  {let {[y {+ 3 4}]}
    {+ x x}}}}
--> 6
```

Identificateurs locaux

```

<Exp> ::= <Number>
        | <Symbol>
        | {+ <Exp> <Exp>}
        | {* <Exp> <Exp>}
        | {<Symbol> <Exp>}
        | {let {[<Symbol> <Exp>]} <Exp>}
  
```

```

{let {[x {+ 1 2}]}
  {let {[x {+ x 4}]}
    {+ x x}}}
--> 14
  
```

Analyse syntaxique

```
(define-type Exp
```

```
...
```

```
[letE (s : Symbol) (rhs : Exp) (body : Exp)])
```

```
(define (parse [s : S-Exp]) : Exp
```

```
(cond
```

```
...
```

```
[(s-exp-match? `{let [{SYMBOL ANY}] ANY} s) ; `{let {[s rhs]} body}
```

```
(let ([sl (s-exp->list s)]) ; (list `let `[s rhs] `body)
```

```
(let ([subst (s-exp->list (first (s-exp->list (second sl))))]) ; (list `s `rhs)
```

```
(letE (s-exp->symbol (first subst))
```

```
(parse (second subst))
```

```
(parse (third sl))))]
```

```
... ))
```

Implémentation

```

(define-type Exp
  [numE (n : Number)]
  [idE (s : Symbol)]
  [plusE (l : Exp) (r : Exp)]
  [multE (l : Exp) (r : Exp)]
  [appE (fun : Symbol) (arg : Exp)]
  [letE (s : Symbol) (rhs : Exp) (body : Exp)])

(define (interp [e : Exp] [fds : (Listof FunDef)]) : Number
  (type-case Exp e
    ...
    [(letE s rhs body) (interp (subst (numE (interp rhs fds))
                                       s
                                       body)
                                fds)]))

```

Substitution

□ Comment bien substituer ?

- Substituer x par 1 dans $\{\text{let } \{[y \ 2]\} \ x\}$ doit donner $\{\text{let } \{[y \ 2]\} \ 1\}$.
- Substituer x par 1 dans $\{\text{let } \{[y \ x]\} \ y\}$ doit donner $\{\text{let } \{[y \ 1]\} \ y\}$.
- Substituer x par 1 dans $\{\text{let } \{[x \ y]\} \ x\}$ doit donner $\{\text{let } \{[x \ y]\} \ x\}$.
- Substituer x par 1 dans $\{\text{let } \{[x \ x]\} \ x\}$ doit donner $\{\text{let } \{[x \ 1]\} \ x\}$.

```
(define (subst [what : Exp] [for : Symbol] [in : Exp]) : Exp
  (type-case Exp in
    ...
    [(letE s rhs body)
     (letE s
       (subst what for rhs)
       (if (equal? for s) body (subst what for body))))])
```

ENVIRONNEMENT

Coût de la substitution

```
{let {[x1 1]}  
  {let {[x2 2]}  
    {let {[x3 3]}  
      ...  
        {let {[xn n]}  
          {+ x1 {+ x2 {+ x3 { ... xn} ... }}}}}}}}
```

❑ On traverse n fois le code source qui est lui-même de taille n !

Inefficace et contre-intuitif !

Environnement

- ❑ On cherche à retarder la substitution.
- ❑ On part d'un environnement vide qui va s'enrichir.



```
{let {[x1 1]}  
  {let {[x2 2]}  
    {let {[x3 3]}  
      ...  
      {let {[xn n]}  
        {+ x1 {+ x2 {+ x3 { ... xn}}}}}}}}}
```

Environnement

- ❑ On cherche à retarder la substitution.
- ❑ On part d'un environnement vide qui va s'enrichir.

x1 = 1

```
{let {[x2 2]}  
  {let {[x3 3]}  
    ...  
    {let {[xn n]}  
      {+ x1 {+ x2 {+ x3 { ... xn}}}}}}}
```

Environnement


- ❑ On cherche à retarder la substitution.
- ❑ On part d'un environnement vide qui va s'enrichir.

$x_n = n \quad \dots \quad x_3 = 3 \quad x_2 = 2 \quad x_1 = 1$


$\{+ \ x_1 \ \{+ \ x_2 \ \{+ \ x_3 \ \{ \ \dots \ x_n \} \} \} \}$

- ❑ Le prochain appel à `interp` va aboutir à évaluer x_1 .
- ❑ On va alors substituer l'identificateur par sa valeur dans l'environnement.

Environnement



```
(interp {let {[x 1]}
  {let {[x {+ x 1}]}}
  x}})
```



```
(interp {let {[x {+ x 1}]}}
  x})
```



```
(interp x)
--> 2
```

- ❑ On utilise toujours la première occurrence trouvée dans l'environnement pour la substitution.

Implémentation

- ❑ L'interpréteur prend un argument supplémentaire pour l'environnement.

```
interp : (Exp Env (Listof FunDef) -> Number)
```

- ❑ On doit définir de nouveaux types et de nouvelles primitives pour gérer l'environnement.

```
bind : (Symbol Number -> Binding)
```

```
mt-env : Env
```

```
extend-env : (Binding Env -> Env)
```

```
lookup : (Symbol Env -> Number)
```

Implémentation

□ Liaisons

```
(define-type Binding  
  [bind (name : Symbol) (val : Number)])
```

□ Environnements

```
(define-type-alias Env (Listof Binding))  
(define mt-env empty)  
(define extend-env cons)
```

Cette implémentation n'est pas efficace.
On l'utilise par souci de simplicité .

□ Recherche dans un environnement

```
(define (lookup [n : Symbol] [env : Env]) : Number  
  (cond  
    [(empty? env) (error 'lookup "free identifier")]  
    [(equal? n (bind-name (first env))) (bind-val (first env))]  
    [else (lookup n (rest env))]))
```

Implémentation

□ Synopsis

```
(interp {let {[x 1]}  
        {let {[y {+ x 1}]}  
            {+ x y}}}  
mt-env)
```

```
(interp {let {[y {+ x 1}]}  
        {+ x y}}  
(extend-env (bind 'x 1) mt-env))
```

```
(interp {+ x y} (extend-env (bind 'y 2) (extend-env (bind 'x 1) mt-env)))  
--> 3
```

Implémentation

```

(define (interp [e : Exp] [env : Env] [fds : (Listof FunDef)]) : Number
  (type-case Exp e
    [(numE n) n]
    [(idE s) (lookup s env)]
    [(plusE l r) (+ (interp l env fds) (interp r env fds))]
    [(multE l r) (* (interp l env fds) (interp r env fds))]
    [(appE f arg)
     (let [(fd (get-fundef f fds))]
       (interp (fd-body fd)
                (extend-env (bind (fd-par fd) (interp arg env fds)) env)
                fds)))]
    [(letE s rhs body)
     (interp body
              (extend-env (bind s (interp rhs env fds)) env)
              fds)))]))

```


Implémentation

❑ En a-t-on fini ?

```
{define {f x} {+ x y}}
{define {g y} {f y}}
```


{interp {g 1}}


{interp {f y}}


{interp {+ x y}}

-- 

Ce n'est pas le résultat que l'on aurait
obtenu avec une substitution classique !

Implémentation

```

(define (interp [e : Exp] [env : Env] [fds : (Listof FunDef)]) : Number
  (type-case Exp e
    [(numE n) n]
    [(idE s) (lookup s env)]
    [(plusE l r) (+ (interp l env fds) (interp r env fds))]
    [(multE l r) (* (interp l env fds) (interp r env fds))]
    [(appE f arg)
      (let [(fd (get-fundef f fds))]
        (interp (fd-body fd)
          (extend-env (bind (fd-par fd) (interp arg env fds)) env)
          fds))]
    [(letE s rhs body)
      (interp body
        (extend-env (bind s (interp rhs env fds)) env)
        fds))]))
  
```

Où est l'erreur ?

Ici !



Implémentation

```

(define (interp [e : Exp] [env : Env] [fds : (Listof FunDef)]) : Number
  (type-case Exp e
    [(numE n) n]
    [(idE s) (lookup s env)]
    [(plusE l r) (+ (interp l env fds) (interp r env fds))]
    [(multE l r) (* (interp l env fds) (interp r env fds))]
    [(appE f arg)
     (let [(fd (get-fundef f fds))]
       (interp (fd-body fd)
                (extend-env (bind (fd-par fd) (interp arg env fds)) mt-env)
                fds)))]
    [(letE s rhs body)
     (interp body
              (extend-env (bind s (interp rhs env fds)) env)
              fds))]))
  
```

On doit réinitialiser l'environnement lors d'un appel de fonction.

Liaison : Terminologie

□ On a vu plusieurs structures **liantes** qui permettent d'introduire des identificateurs.

```
{let {[x 1]} ... }  
{define {f x} ... }
```

□ Un identificateur est dit **lié** s'il a été introduit par une de ces structures.

```
{let {[x 1]} ... x ... }  
{define {f x} ... x ... }
```

□ Un identificateur qui n'est pas lié est dit **libre**.

```
{let {[x 1]} ... y ... }  
{define {f x} ... y ... }
```

Identificateurs liés et libres

```
{let {[x 1]}  
  {let {[y x]}  
    {+ y {+ z x}}}}}
```

Identificateurs liés et libres

{let {[y x]}
{+ y {+ z x}}}}

Identificateurs liés et libres

```
{let {[x 1]}  
  {let {[x x]}  
    {+ y {+ z x}}}}}
```

Identificateurs liés et libres

{let { [x] [x] }
{+ y {+ z x} } } }

Identificateurs liés et libres

```
{define {double x} {+ x x}}
```

```
{double 3}
```

Portée

□ Portée lexicale (statique)

La liaison d'un identificateur n'agit que sur une portion du code source.

- **Locale.**

```
{let {[x 1]} ... }  
{define {f x} ... }
```

`x` a une portée locale.

- **Globale.**

```
{define {double x} {+ x x}}  
{define {quadruple x} {double {double x}}}
```

`double` et `quadruple` ont une portée globale.

□ Portée dynamique

La liaison d'un identificateur dépend du flot d'exécution du programme.

ORDRE SUPÉRIEUR

Valeur

- ❑ Quel est le résultat de l'évaluation d'une expression?
- ❑ Dans notre langage, seulement un nombre.
- ❑ De manière générale, beaucoup de choses sont possibles
 - Caractère
 - Chaîne de caractère
 - Booléen
 - Structure de donnée (liste, tableau, ...)
 - ...
- ❑ Le résultat d'une expression est une **valeur**.
- ❑ Une valeur peut être passée en argument d'une fonction ou liée à une variable.

Une fonction est-elle une valeur?

☐ Dans notre langage : Non!

☐ On peut définir une fonction

```
{define {double x} {+ x x}}
```

☐ On peut appeler une fonction

```
{double 3}
```

☐ On ne peut pas renvoyer une fonction

```
(interp (parse `double)
```

```
  mt-env
```

```
    (list (parse-fundef `{define {double x} {+ x x}})))
```

```
--> lookup : free identifier
```

Une fonction est-elle une valeur?

❑ En plait : Oui!

❑ Une fonction peut être le résultat d'une expression

```
(lambda (x) (+ x x))
```

```
(if (= n 1) first second)
```

❑ On peut utiliser une fonction comme argument d'une autre fonction

```
(map (lambda (x) (+ x x)) (list 1 2 3 4 5))
```

Quel est l'avantage?

❑ Permet d'abstraire facilement certains comportements avancés

`map, filter, foldl, ...`

❑ Introduit dans les versions les plus récentes de langages impératifs
(C++11, Java 8, ...)

❑ Conséquence dans notre langage : la clause `define` devient superflue

```
{define {double x} {+ x x}}  
{double 3}
```



```
{let {[double {lambda {x} {+ x x}}]}}  
  {double 3}}
```

Grammaire du langage

```
<Exp> ::= <Number>  
        | <Symbol>  
        | {+ <Exp> <Exp>}  
        | {* <Exp> <Exp>}  
        | {let {[<Symbol> <Exp>]} <Exp>}  
        | {lambda {<Symbol>} <Exp>} New !  
        | {<Symbol> <Exp>}
```


Exemple d'évaluations

- ❑ $0 \longrightarrow 0$
 - ❑ $x \longrightarrow \text{free identifier}$
 - ❑ $\{+ \ 1 \ 2\} \longrightarrow 3$
 - ❑ $\{* \ 1 \ 2\} \longrightarrow 2$
 - ❑ $\{\text{let } \{[x \ 1]\} \ \{+ \ x \ 2\}\} \longrightarrow \{+ \ 1 \ 2\} \longrightarrow 3$
 - ❑ $\{\text{lambda } \{x\} \ \{+ \ x \ x\}\} \longrightarrow \{\text{lambda } \{x\} \ \{+ \ x \ x\}\}$
 - ❑ $\{\text{let } \{[\text{double } \{\text{lambda } \{x\} \ \{+ \ x \ x\}\}]\} \ \{\text{double } \ 3\}\}$
 - ❑ L'interpréteur ne renvoie plus forcément un nombre.
- $(\text{interp } [e : \text{Exp}] \ \dots) : \text{Number}$ **incorrect !**
 $(\text{interp } [e : \text{Exp}] \ \dots) : \text{Value}$ **correct !**

Exemple d'évaluations

- ❑ $0 \longrightarrow 0$
- ❑ $x \longrightarrow \text{free identifier}$
- ❑ $\{+ \ 1 \ 2\} \longrightarrow 3$
- ❑ $\{* \ 1 \ 2\} \longrightarrow 2$
- ❑ $\{\text{let } \{[x \ 1]\} \ \{+ \ x \ 2\}\} \longrightarrow \{+ \ 1 \ 2\} \longrightarrow 3$
- ❑ $\{\text{lambda } \{x\} \ \{+ \ x \ x\}\} \longrightarrow \{\text{lambda } \{x\} \ \{+ \ x \ x\}\}$
- ❑ $\{\text{let } \{[\text{double } \{\text{lambda } \{x\} \ \{+ \ x \ x\}\}]\} \ \{\text{double } \ 3\}\}$
 $\longrightarrow \{\{\text{lambda } \{x\} \ \{+ \ x \ x\}\} \ 3\}$
- ❑ Ce n'est pas une expression valide de notre langage !

Grammaire du langage

```
<Exp> ::= <Number>
        | <Symbol>
        | {+ <Exp> <Exp>}
        | {* <Exp> <Exp>}
        | {let {[<Symbol> <Exp>]} <Exp>}
        | {lambda {<Symbol>} <Exp>}
        | {<Exp> <Exp>}
```

New !**New !**

Evaluation

□ Maintenant que les appels de fonctions peuvent prendre une expression pour la fonction

```
{let {[double {lambda {x} {+ x x}}]} {double 3}}
```

➡

```
{{lambda {x} {+ x x}} 3}
```

➡

```
{+ 3 3}
```

 ➡ 6

□ De nouvelles erreurs peuvent survenir

- ```
{1 2}
```

 ➡ not a function
- ```
{+ 1 {lambda {x} x}}
```

 ➡ not a number

Représentation des expressions

```
(define-type Exp
  [numE (n : Number)]
  [idE (s : Symbol)]
  [appE (fun : Exp) (arg : Exp)]
  [plusE (l : Exp) (r : Exp)]
  [multE (l : Exp) (r : Exp)]
  [letE (s : Symbol) (rhs : Exp) (body : Exp)]
  [lamE (par : Symbol) (body : Exp)])

(test (parse `{lambda {x} {+ x 1}})
      (lamE 'x (plusE (idE 'x) (numE 1))))
```

Représentation des expressions

```
(define-type Exp
  [numE (n : Number)]
  [idE (s : Symbol)]
  [appE (fun : Exp) (arg : Exp)]
  [plusE (l : Exp) (r : Exp)]
  [multE (l : Exp) (r : Exp)]
  [letE (s : Symbol) (rhs : Exp) (body : Exp)]
  [lamE (par : Symbol) (body : Exp)])

(test (parse `{{lambda {x} {+ x 1}} 2})
      (appE (lamE 'x (plusE (idE 'x) (numE 1)))
            (numE 2)))
```

Fonctions et substitution

□ Si on applique la substitution brute

```
(interp {let {[y 1]}  
        {lambda {x} {+ x y}}})
```

devient

```
(interp {lambda {x} {+ x 1}})
```

□ Et avec un environnement?

Fonctions et environnement



```
(interp {let {[y 1]} {lambda {x} {+ x y}}})
```



```
(interp {lambda {x} {+ x y}})
```

☐ Que doit-on renvoyer?

☐ Un paramètre et un corps comme pour `lamE`?

Fonctions et environnement

❑ Que se passe-t-il alors?

```
(interp {let {[y 1]} {{lambda {x} {+ x y}} 2}})
```



```
(interp {{lambda {x} {+ x y}} 2}}
```



```
(interp {+ x y})
```

y = 1

x = 2

Un appel de fonction réinitialise l'environnement !

❑ Il nous manque quelque chose : une fonction doit se souvenir de son environnement de définition.

❑ C'est ce que l'on appelle une **clôture lexicale**!

Représentation des valeurs

```
(define-type Value
  [numV (n : Number)]
  [closV (par : Symbol) (body : Exp) (env : Env)])

(define-type Binding
  [bind (name : Symbol) (val : Value)])

(test (interp (parse `{let {[y 1]}
                        {lambda {x} {+ x y}}}})
      mt-env)
(closV 'x
  (plusE (idE 'x) (idE 'y))
  (extend-env (bind 'y (numV 1)) mt-env)))
```

Comment va se passer l'évaluation?



```
(interp {let {[y 1]} {{lambda {x} {+ x y}}} 2}})
```

- Pour l'argument :



```
y = (numV 1)
```

```
(interp 2) → (numV 2)
```

- Pour la fonction :



```
y = (numV 1)
```

```
(interp {lambda {x} {+ x y}})
```

```
→ (closV 'x
    (plusE (idE 'x) (idE 'y))
    (extend-env (bind 'y (numV 1)) mt-env)))
```

Comment va se passer l'évaluation?

```
(interp {let {[y 1]} {{lambda {x} {+ x y}} 2}})
```

- Pour l'application de la fonction :

```
x = (numV 2) ...
```

```
(interp {+ x y})
```

Valeur de l'argument lié
au nom du paramètre

Corps et paramètre de la fonction
récupéré dans la clôture lexicale

Comment va se passer l'évaluation?

```
(interp {let {[y 1]} {{lambda {x} {+ x y}}} 2}})
```

- Pour l'application de la fonction :

```
(interp {+ x y})
```

(Note: In the original image, the expression `x = (numV 2) y = (numV 1)` is highlighted in an orange box and is positioned between the two code snippets, with arrows pointing from it to the `x` and `y` in the second snippet.)

Ajouté à l'environnement
de la clôture lexicale

Valeur de l'argument lié
au nom du paramètre

Corps et paramètre de la fonction
récupéré dans la clôture lexicale

Interpréteur

```

(define (interp [e : Exp] [env : Env]) : Value
  (type-case Exp e
    [(numE n) (numV n)]
    [(idE s) (lookup s env)]
    [(plusE l r) (num+ (interp l env) (interp r env))]
    [(multE l r) (num* (interp l env) (interp r env))]
    [(lamE par body) ... ]
    [(appE f arg)
     ...
     ...
     ... ]
    [(letE s rhs body) ... ])))

```

Addition et multiplication

☐ Fonctions utilitaires

```
(define (num+ [l : Value] [r : Value]) : Value
  (if (and (numV? l) (numV? r))
      (numV (+ (numV-n l) (numV-n r)))
      (error 'interp "not a number")))
```

```
(define (num* [l : Value] [r : Value]) : Value
  (if (and (numV? l) (numV? r))
      (numV (* (numV-n l) (numV-n r)))
      (error 'interp "not a number")))
```

Addition et multiplication

□ Généralisation

```
(define (num-op [op : (Number Number -> Number)]  
          [l : Value] [r : Value]) : Value  
  (if (and (numV? l) (numV? r))  
      (numV (op (numV-n l) (numV-n r)))  
      (error 'interp "not a number")))
```

```
(define (num+ [l : Value] [r : Value]) : Value  
  (num-op + l r))
```

```
(define (num* [l : Value] [r : Value]) : Value  
  (num-op * l r))
```


Interpréteur

```
(define (interp [e : Exp] [env : Env]) : Value
  (type-case Exp e
    [(numE n) (numV n)]
    [(idE s) (lookup s env)]
    [(plusE l r) (num+ (interp l env) (interp r env))]
    [(multE l r) (num* (interp l env) (interp r env))]
    [(lamE par body) (closV par body env)]
    [(appE f arg)
     (type-case Value (interp f env)
       [(closV par body c-env)
        (interp body (extend-env (bind par (interp arg env)) c-env))]
       [else (error 'interp "not a function")])]
    [(letE s rhs body) (interp body (extend-env (bind s (interp rhs env)) env))]))
```

Espaces de noms

□ Avec l'interpréteur `env.rkt`:

```
{define {f x} {+ x x}}  
{let {[f 3]} {f f}}  
--> 6
```

□ Avec l'interpréteur `ordresup.rkt`:

```
{let {[f {lambda {x} {+ x x}}]}  
  {let {[f 3]}  
    {f f}}}  
--> interp : not a function
```

Espaces de noms

- ❑ Dans l'interpréteur `env.rkt`, les noms de fonctions et les noms d'identificateurs 'vivent' dans deux mondes différents.
- ❑ Dans l'interpréteur `ordresup.rkt`, les noms de fonctions et les noms d'identificateurs apparaissent tous dans l'environnement.
- ❑ Deux entités peuvent porter le même nom si elles existent dans des espaces de noms distincts.
- ❑ Notion présente dans de nombreux langages
 - Les package en Java.
 - Les namespace en C++ ou C#.
 - Les modules en Python ou Racket.
 - Les classes dans les langages objet.