Programmation fonctionnelle - PG104 -

COURS 3

Myriam Desainte-Catherine et David Renault

January 27, 2020

Connaissez-vous cette personnalité? Oui Non

- Richard Stallman (Free Software Foundation) – rms

Environnement et extensibilité (RMS - emacs-lisp)

- Fonction A
  - Appelle la fonction B
- Fonction B
  - Appelle la fonction C
- Fonction C

Environnement et extensibilité (RMS - emacs-lisp)

- Fonction A — Extension utilisateur
  - Appelle la fonction B
- Fonction B — Système standard
  - Appelle la fonction C
- Fonction C — Extension utilisateur

Environnement et extensibilité (RMS - emacs-lisp)

- Fonction A — Extension utilisateur
  - Appelle la fonction B
- Fonction B — Système standard
  - Appelle la fonction C – via une table
- Fonction C — Extension utilisateur

Environnement et extensibilité (RMS - emacs-lisp)

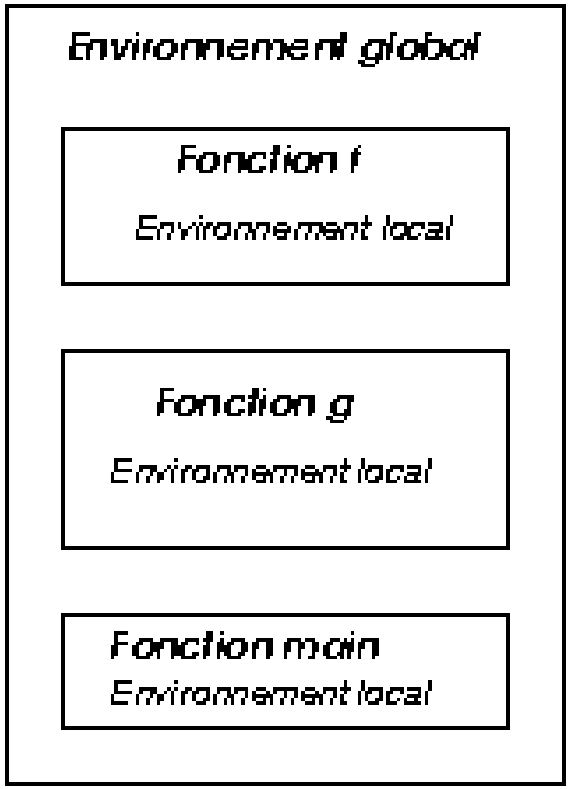
- Fonction A — Extension utilisateur
  - Définit la variable FOO
  - Appelle la fonction B
- Fonction B — Système standard
  - Appelle la fonction C – via une table
- Fonction C — Extension utilisateur
  - Utilise la variable FOO

Environnement et extensibilité (RMS - emacs-lisp)

- Fonction A — Extension utilisateur
  - Définit la variable FOO
  - Appelle la fonction B – avec paramètre FOO
- Fonction B — Système standard
  - Appelle la fonction C – avec paramètre FOO
- Fonction C — Extension utilisateur
  - Utilise la variable FOO

Organisation des environnements : imbrication

En langage C



- **Environnement global** : fabriqué au moyen de déclarations situées en dehors de toute fonction.
- **Environnements locaux** : fabriqués au moyen de
  - Déclarations dans des blocs {}
  - Paramètres des fonctions
- Comment une déclaration est-elle recherchée dans l'environnement?
- Existe-t-il plusieurs modèles de stratégies? lexical / dynamique
- Quel lien entre le fonctionnement des environnements et le paradigme fonctionnel?

Organisation des environnements : stratégie de recherche

Problème: L'occurrence d'une variable peut correspondre à plusieurs déclarations dans l'environnement. Laquelle faut-il choisir?

Exemples en C : Que vaut i lors du calcul de f(3)?

Paramètre d'une fonction

```
int i=0;
int
f(int i)
{
  return i;
}
```

3 0 2

Variable locale à une fonction

```
int i=0;
int f(int x)
{
  int i=3;
  return i*x;
}
```

3 0 2

Variable libre d'une fonction : non défini dans l'environnement local.

```
int i=0;
int
int f(int x)
{
  return i*x;
}
```

3 0 2

Appel à la fonction f

```
int g()
{
  int i=2;
  return f(3);
}
```

Symboles et liaisons

Définitions

- Un **symbole** est un identificateur, c'est-à-dire un nom symbolique.
- Une **liaison** est une entité, c'est-à-dire un objet nommé résidant dans la mémoire, donc l'association d'un symbole avec un emplacement mémoire contenant une valeur.

Exemple

```
int
g(int i)
{
  return i;
}
```

```
int
f(int n)
{
  int i=1;
  return i+n;
}
```

Dans un programme un même symbole peut apparaître dans plusieurs liaisons. De même, en C, un identificateur peut aussi servir à nommer plusieurs entités. Plusieurs stratégies de recherche ont été implémentées dans les langages de programmation.

Environnements global et locaux

L'environnement est formé de **liaisons** *symbole* $\longrightarrow$ *valeur*. Les symboles ne sont pas typés (non déclarés), mais leurs valeurs le sont. Il s'agit d'un **typage dynamique**.

Environnement global : la forme **define** au top-level

- Variables : (**define** $\langle v \rangle$ $\langle e \rangle$)
- Fonctions : (**define** ($\langle f \rangle$ $\langle p_1 \rangle$ $\langle p_2 \rangle$ ... $\langle p_n \rangle$) $\langle e_1 \rangle$ $\langle e_2 \rangle$ ... $\langle e_n \rangle$)
- Résultat non spécifié par la norme

Une définition établit une liaison entre une variable et un objet résultant de l'évaluation de l'expression, cet objet pouvant être une fonction.

Exemple

```
> (define a 0)
> (define (f x) x)
> (define g (lambda(x) (* 2 x)))
> a
0
> (f 1)
1
> (g 1)
```

1 2 error

Définitions et évaluation

- Quelle est la forme correcte pour définir une fonction : `(define g(x) x)` ou `(define (f x) x)`
- Quelle est la forme correcte pour appliquer une fonction : `f(1)` ou `(f 1)`
- Le symbole *e* dans cette définition est-il relié à une fonction ou bien à un numérique ? `(define e (* 2 4))`
- Le symbole *e* dans cette définition est-il relié à une fonction ou bien à un numérique ? `(define (e) (* 2 4))`
- Soient les définitions suivantes : `(define (f) 1)` `(define (g x) (* 2 x))` Parmi les expressions suivantes lesquelles sont valides :
  - `(f 1)` `(f)` `(f 1 2)`
  - `(g 1)` `(g)` `(g #t)`

Définitions et évaluation

- Quelle est la forme correcte pour définir une fonction : `(define (f x) x)`
- Quelle est la forme correcte pour appliquer une fonction : `f(1)` ou `(f 1)`
- Le symbole *e* dans cette définition est-il relié à une fonction ou bien à un numérique ? `(define e (* 2 4))`
- Le symbole *e* dans cette définition est-il relié à une fonction ou bien à un numérique ? `(define (e) (* 2 4))`
- Soient les définitions suivantes : `(define (f) 1)` `(define (g x) (* 2 x))` Parmi les expressions suivantes lesquelles sont valides :
  - `(f 1)` `(f)` `(f 1 2)`
  - `(g 1)` `(g)` `(g #t)`

Définitions et évaluation

- Quelle est la forme correcte pour définir une fonction : `(define (f x) x)`
- Quelle est la forme correcte pour appliquer une fonction : `(f 1)`
- Le symbole *e* dans cette définition est-il relié à une fonction ou bien à un numérique ? `(define e (* 2 4))`
- Le symbole *e* dans cette définition est-il relié à une fonction ou bien à un numérique ? `(define (e) (* 2 4))`
- Soient les définitions suivantes : `(define (f) 1)` `(define (g x) (* 2 x))` Parmi les expressions suivantes lesquelles sont valides :
  - `(f 1)` `(f)` `(f 1 2)`
  - `(g 1)` `(g)` `(g #t)`

Définitions et évaluation

- Quelle est la forme correcte pour définir une fonction : `(define (f x) x)`
- Quelle est la forme correcte pour appliquer une fonction : `(f 1)`
- Le symbole *e* dans cette définition est-il relié à une fonction ou bien à un numérique ? `(define e (* 2 4))` numérique
- Le symbole *e* dans cette définition est-il relié à une fonction ou bien à un numérique ? `(define (e) (* 2 4))`
- Soient les définitions suivantes : `(define (f) 1)` `(define (g x) (* 2 x))` Parmi les expressions suivantes lesquelles sont valides :
  - `(f 1)` `(f)` `(f 1 2)`
  - `(g 1)` `(g)` `(g #t)`

Définitions et évaluation

- Quelle est la forme correcte pour définir une fonction : `(define (f x) x)`
- Quelle est la forme correcte pour appliquer une fonction : `(f 1)`
- Le symbole *e* dans cette définition est-il relié à une fonction ou bien à un numérique ? `(define e (* 2 4))` numérique
- Le symbole *e* dans cette définition est-il relié à une fonction ou bien à un numérique ? `(define (e) (* 2 4))` fonction
- Soient les définitions suivantes : `(define (f) 1)` `(define (g x) (* 2 x))` Parmi les expressions suivantes lesquelles sont valides :
  - `(f 1)` `(f)` `(f 1 2)`
  - `(g 1)` `(g)` `(g #t)`

Définitions et évaluation

- Quelle est la forme correcte pour définir une fonction : `(define (f x) x)`
- Quelle est la forme correcte pour appliquer une fonction : `(f 1)`
- Le symbole *e* dans cette définition est-il relié à une fonction ou bien à un numérique ? `(define e (* 2 4))` numérique
- Le symbole *e* dans cette définition est-il relié à une fonction ou bien à un numérique ? `(define (e) (* 2 4))` fonction
- Soient les définitions suivantes : `(define (f) 1)` `(define (g x) (* 2 x))` Parmi les expressions suivantes lesquelles sont valides :
  - `(f)`
  - `(g 1)` `(g)` `(g #t)`

Définitions et évaluation

- Quelle est la forme correcte pour définir une fonction : `(define (f x) x)`
- Quelle est la forme correcte pour appliquer une fonction : `(f 1)`
- Le symbole *e* dans cette définition est-il relié à une fonction ou bien à un numérique ? `(define e (* 2 4))` numérique
- Le symbole *e* dans cette définition est-il relié à une fonction ou bien à un numérique ? `(define (e) (* 2 4))` fonction
- Soient les définitions suivantes :

  ```
  (define (f) 1)
  (define (g x) (* 2 x))
  ```

  Parmi les expressions suivantes lesquelles sont valides :

  - `(f)`
  - `(g 1)` `(g #t)` ?

Environnements locaux - Forme **let**

Ils sont fabriqués avec les formes **let**, **let*** , **letrec**, et par des définitions au moyen de la forme **define** dans le corps des fonctions.

La forme **let**

(**let** (⟨l_1⟩
⟨l_2⟩
...
⟨l_n⟩)
⟨e⟩)

- ⟨l_i⟩ est une liaison : (⟨s_i⟩ ⟨o_i⟩)
- ⟨s_i⟩ est un symbole (id. de variable)
- ⟨o_i⟩ une valeur d'initialisation
- ⟨e⟩ est une expression
- Résultat de l'évaluation de l'expression ⟨e⟩ dans l'environnement créé

L'évaluation des valeurs d'initialisation est effectuée en premier puis les variables locales sont créées. Ce qui implique que les valeurs des variables locales définies dans un let ne sont pas utilisées dans l'évaluation des expressions d'initialisation.

Environnements locaux - Forme **let***

La forme **let***

(**let*** (⟨l_1⟩
⟨l_2⟩
...
⟨l_n⟩)
⟨e⟩)

- ⟨l_i⟩ est une liaison : (⟨s_i⟩ ⟨o_i⟩)
- ⟨s_i⟩ est un symbole (id. de variable)
- ⟨o_i⟩ une valeur d'initialisation
- ⟨e⟩ est une expression
- Résultat de l'évaluation de l'expression ⟨e⟩ dans l'environnement créé

L'évaluation des expressions d'initialisation est effectuée après la création des variables locales.

Exemple

```
> (let ((a 2)
        (b 3)
        (c 0))
    (- (* b b)
       (* 4 a c)))
9
```

```
> (let* ((a 2)
         (b 3)
         (c a))
    (- (* b b)
       (* 4 a c)))
-7
```

Différencier les expressions let et let*

Quels sont les résultats des expressions suivantes?

```
(define b 0)
(let ((b 1)
      (c 2))
  (+ b c))
```

2 ou 3

```
(define b 0)
(let ((b 1)
      (c b))
  (+ c b))
```

1 ou 2

```
(define b 0)
(let* ((b 1)
       (c b))
  (+ c b))
```

1 ou 2

Différencier les expressions let et let*

Quels sont les résultats des expressions suivantes?

```
(define b 0)
(let ((b 1)
      (c 2))
  (+ b c))
```

3

```
(define b 0)
(let ((b 1)
      (c b))
  (+ c b))
```

1 ou 2

```
(define b 0)
(let* ((b 1)
       (c b))
  (+ c b))
```

1 ou 2

Différencier les expressions let et let*

Quels sont les résultats des expressions suivantes?

```
(define b 0)
(let ((b 1)
      (c 2))
  (+ b c))
```

3

```
(define b 0)
(let ((b 1)
      (c b))
  (+ c b))
```

1

```
(define b 0)
(let* ((b 1)
       (c b))
  (+ c b))
```

1 ou 2

Différencier les expressions let et let*

Quels sont les résultats des expressions suivantes?

```
(define b 0)
(let ((b 1)
      (c 2))
  (+ b c))
```

3

```
(define b 0)
(let ((b 1)
      (c b))
  (+ c b))
```

1

```
(define b 0)
(let* ((b 1)
       (c b))
  (+ c b))
```

2

Environnements locaux - Forme **letrec**

La forme **letrec**

(**letrec** (⟨l_1⟩
⟨l_2⟩
...
⟨l_n⟩)
⟨e⟩)

- ⟨l_i⟩ est une liaison : (⟨s_i⟩ ⟨o_i⟩)
- ⟨s_i⟩ est un symbole (id. de variable)
- ⟨o_i⟩ une lambda-expression
- ⟨e⟩ est une expression
- Résultat de l'évaluation de l'expression ⟨e⟩ dans l'environnement créé

L'évaluation des expressions d'initialisation permet les définitions récursives.

Exemple

```
> (letrec ((fact
            (lambda (n)
              (if (zero? n)
                  1
                  (* n (fact (sub1 n)))))))
    (fact 2))
> 2
```

Stratégies de recherche d'une liaison

Pour chercher la liaison correspondant à l'occurrence d'un symbole dans une expression, la recherche commence par l'environnement dans lequel apparaît l'expression. Si l'occurrence apparaît dans le corps d'une fonction et qu'aucune liaison ne correspond en local (cas d'une variable libre), deux stratégies existent.

Stratégie lexicale – Lexical scope

La **stratégie lexicale** consiste à remonter les environnements locaux englobants du plus proche jusqu'à l'environnement global.
La première liaison dont le nom de symbole correspond est retenue. Cette stratégie s'applique aussi à l'évaluation du corps d'une fonction lors d'une application. En effet, celui-ci est évalué dans l'environnement englobant de la fonction, dit **environnement lexical**.
Cette stratégie correspond au langage C et aux langages impératifs en général et au langage Scheme.

Stratégie dynamique – Dynamic scope

Pour chercher la liaison correspondant à l'occurrence d'un symbole dans une expression située dans le corps d'une fonction, la **stratégie dynamique** consiste à rechercher sa liaison dans l'**environnement dynamique**, c'est-à-dire l'environnement d'application de la fonction.
Cette stratégie correspond par exemple à LaTeX, et beaucoup de lisp dont emacs-lisp. Common-Lisp implémente les deux stratégies.

Portées lexicales et dynamiques

Racket : lexical

```
> (define i 0)
> (define (f x) (* x i))
> (f 3)
0
> (let ((i 2)) (f 3))
0
> (let ((j 0))
    (let ((g (lambda(x)
               (* x j))))
      (let ((j 3))
        (g 3))))
0
```

emacs-lisp : dynamique

```
> (defvar i 0)
> (defun f(x) (* x i))
> (f 3)
0
> (let ((i 2)) (f 3))
6
> (let ((j 0))
    (flet ((g (x)
             (* x j)))
      (let ((j 2))
        (g 3))))
6
```

Common Lisp : dynamique

```
> (defvar i 0); variable speciale
> (defun f(x) (* x i))
> (f 3)
0
> (let ((i 2)) (f 3))
6
```

Common Lisp : lexical

```
> (let ((j 0)); variable lexicale
    (flet ((g (x)
             (* x j)))
      (let ((j 2))
        (g 3))))
0
```

Portée dynamique en LaTeX

- Style book : saute 2 pages avant la table of contents
- Style report : saute 1 page avant la table of contents
- On souhaite ne sauter qu'une page dans le style book
- Commande `cleardoublepage` saute 2 pages
- Commande `clearpage` saute 1 page
- Commande `renewcommand` redéfinit une commande

Exemple

```
%% creation d'un bloc avec environnement local
\begingroup
\renewcommand{\cleardoublepage}{\clearpage}
\tableofcontents
\endgroup           %% sortie du bloc,
...
\cleardoublepage    %% \cleardoublepage restauree
```

Portée et durée de vie en Scheme

Portée lexicale

La portée d'une liaison est la partie du code source dans laquelle il est possible de l'utiliser.

- Les liaisons globales ont une portée égale à tout le programme.
- Les liaisons locales ont une portée limitée à la forme de définition let.

Durée de vie

La durée de vie d'un objet correspond à la période de l'exécution d'un programme comprise entre la création de cet objet et sa destruction.

- Les objets définis globalement ont une durée de vie égale à celle du programme.
- Les objets définis localement ont une durée de vie potentiellement égale à celle du programme.

Les environnements : portée

- La stratégie lexicale implique que la portée d'une variable peut être déterminée lors de : la lecture ou l'exécution du programme
- Scheme est un langage lexical ou dynamique
- Dans la définition de la fonction **f**, la variable **a** est dite : libre ou liée
- Le résultat de l'expression suivante est : 1 ou 0

```
(let ((a 1))
  (f 0))
```

Exemple

```
(define a 0)
(define (f n)
  (if (zero? n)
      a
      0))
```

Les environnements

- La stratégie lexicale implique que la portée d'une variable peut être déterminée lors de : la lecture du programme
- Scheme est un langage lexical ou dynamique
- Dans la définition de la fonction **f**, la variable **a** est dite : libre ou liée
- Le résultat de l'expression suivante est : 1 ou 0

```
(let ((a 1))
  (f 0))
```

Exemple

```
(define a 0)
(define (f n)
  (if (zero? n)
      a
      0))
```

Les environnements

- La stratégie lexicale implique que la portée d'une variable peut être déterminée lors de : la lecture du programme
- Scheme est un langage lexical
- Dans la définition de la fonction **f**, la variable **a** est dite : libre ou liée
- Le résultat de l'expression suivante est : 1 ou 0

```
(let ((a 1))
  (f 0))
```

Exemple

```
(define a 0)
(define (f n)
  (if (zero? n)
      a
      0))
```

Les environnements

- La stratégie lexicale implique que la portée d'une variable peut être déterminée lors de : la lecture du programme
- Scheme est un langage lexical
- Dans la définition de la fonction **f**, la variable **a** est dite : libre
- Le résultat de l'expression suivante est : 1 ou 0

```
(let ((a 1))
  (f 0))
```

Exemple

```
(define a 0)
(define (f n)
  (if (zero? n)
      a
      0))
```

Les environnements

- La stratégie lexicale implique que la portée d'une variable peut être déterminée lors de : la lecture du programme
- Scheme est un langage lexical
- Dans la définition de la fonction **f**, la variable **a** est dite : libre
- Le résultat de l'expression suivante est : 0

```
(let ((a 1))
  (f 0))
```

Exemple

```
(define a 0)
(define (f n)
  (if (zero? n)
      a
      0))
```

Paradigme fonctionnel et environnements

Équivalence des formes Let et Lambda

La forme **let** équivaut à l'application d'une fonction construite avec la forme **lambda**. Les symboles définis correspondent aux paramètres formels de la fonction, et les expressions associées aux symboles définis correspondent aux arguments de l'application.

```
(let ((j 0))
  (* x j))
```

```
((lambda(j) (* x j)) 0)
```

Questions

La forme **let** est-elle fonctionnelle? oui ou non

La forme **let*** est-elle fonctionnelle? oui ou non

Environnements et paradigme fonctionnel

L'organisation des environnements et des stratégies de recherche des liaisons sont des mécanismes qui ont été ajoutés au lambda-calcul pour en faire un langage de programmation opérationnel. Mais quels sont les liens entre ces mécanismes et le paradigme fonctionnel?

La portée lexicale correspond-elle au paradigme fonctionnel?

- Caractéristiques de la programmation fonctionnelle
  - Fondement provenant du λ-calcul
  - Programmation par applications de fonctions
  - Transparence référentielle
- Reformulation de la question
  - La portée lexicale implique-t-elle la transparence référentielle?
  - La portée lexicale implique-t-elle des constructions qui peuvent être simulés par des applications de fonctions?

Portée et transparence référentielle

Les deux applications (f 3) sont effectuées dans deux environnements différents. Le premier lie i avec 0 et le second lie i avec 2.

Portée lexicale : Scheme

Les résultats des deux applications sont-ils les mêmes? oui ou non

```
> (define i 0)
> (define (f x) (* x i))
> (f 3)
> (let ((i 2)) (f 3) )
```

Portée dynamique : emacs-lisp

Les résultats des deux applications sont-ils les mêmes? oui ou non

```
(defvar i 0)
(defun f(x) (* x i))
(let ((i 2)) (f 3) )
(f 3)
```

Portée et transparence référentielle

Les deux applications (f 3) sont effectuées dans deux environnements différents.
Le premier lie i avec 0 et le second lie i avec 2.

Portée lexicale : Scheme

Les résultats des deux applications sont les mêmes : transparence référentielle.

```
> (define i 0)
> (define (f x) (* x i))
> (f 3) -> 0
> (let ((i 2)) (f 3) ) -> 0
```

Portée dynamique : emacs-lisp

Les résultats dépendent de l'environnement : pas de transparence référentielle.

```
(defvar i 0)
(defun f(x) (* x i))
(let ((i 2)) (f 3) ) -> 6
(f 3) -> 0
```

Portée et formulation fonctionnelle – exercice

Méthode

On utilise la formulation fonctionnelle du **let** pour étudier le phénomène de portée en calculant le résultat sur les exemples précédents..

```
(let ((i 0))
  (let ((f (lambda(x)
             (* x i))))
    (let ((i 2))
      (f 3))))
```

```
((lambda(i)
  (let ((f (lambda(x)
             (* x i))))
    (let ((i 2))
      (f 3))))
 0)
```

Portée et formulation fonctionnelle

Méthode

On utilise la formulation fonctionnelle du **let** pour étudier le phénomène de portée en calculant le résultat sur les exemples précédents..

```
((lambda(i)
  (let ((f (lambda(x)
             (* x i))))
    (let ((i 2))
      (f 3)))
 0)
```

```
((lambda(i)
   ((lambda(f)
      (let ((i 2))
        (f 3)))
    (lambda(x) (* x i))))
 0)
```

Portée et formulation fonctionnelle

Méthode

On utilise la formulation fonctionnelle du **let** pour étudier le phénomène de portée en calculant le résultat sur les exemples précédents..

```
((lambda(i)
   ((lambda(f)
     (let ((i 2))
       (f 3)))
     (lambda(x) (* x i))))
 0)
```

```
((lambda(i)
   ((lambda(f)
      ((lambda(i)
         (f 3))
       2))
     (lambda(x) (* x i))))
 0)
```

Portée et formulation fonctionnelle

Méthode

On utilise la formulation fonctionnelle du **let** pour étudier le phénomène de portée en calculant le résultat sur les exemples précédents..

```
((lambda(i)
   ((lambda(f)
     (let ((i 2))
       (f 3)))
     (lambda(x) (* x i))))
 0)
```

```
((lambda(i)
   ((lambda(f)
      ((lambda(i)
         (f 3))
       2))
     (lambda(x) (* x i))))
 0)
```

$$(\lambda\, i\, .\, (\lambda\, f\, .\, (\lambda\, i\, .\, f\, 3)\, 2)\, (\lambda\, x\, .\, *\, x\, i))\, 0$$

Portée et formulation fonctionnelle en λ-calcul

- $(\lambda\ i\ .\ (\lambda\ f\ .\ (\lambda\ i\ .\ f\ 3)\ 2)\ (\lambda\ x\ .\ *\ x\ i))\ 0$
- $(\lambda\ i\ .\ (\lambda\ f\ .\ (\lambda\ i\ .\ f\ 3)\ 2)\ (\lambda\ x\ .\ *\ x\ i))\ 0$
- $\longrightarrow (\lambda\ f\ .\ (\lambda\ i\ .\ f\ 3)\ 2)\ (\lambda\ x\ .\ *\ x\ 0)$
- $(\lambda\ f\ .\ (\lambda\ i\ .\ f\ 3)\ 2)\ (\lambda\ x\ .\ *\ x\ 0)$
- $\longrightarrow (\lambda\ i\ .\ (\lambda\ x\ .\ *\ x\ 0)\ 3)\ 2$
- $(\lambda\ i\ .\ (\lambda\ x\ .\ *\ x\ 0)\ 3)\ 2$
- $\longrightarrow (\lambda\ x\ .\ *\ x\ 0)\ 3$
- $(\lambda\ x\ .\ *\ x\ 0)\ 3$
- $\longrightarrow * 3\ 0$
- $\longrightarrow 0$

Portée et formulation fonctionnelle en scheme

```
((lambda(i)
   ((lambda(f)
      ((lambda(i)
         (f 3))
       2))
     (lambda(x) (* x i))))
   0)
->
   ((lambda(f)
      ((lambda(i)
         (f 3))
       2))
     (lambda(x) (* x 0)))
```

Portée et formulation fonctionnelle en scheme

```
((lambda(f)
   ((lambda(i)
      (f 3))
    2))
 (lambda(x) (* x 0)))
```

->

```
((lambda(i)
   ((lambda(x) (* x 0)) 3))
 2)
```

Portée et formulation fonctionnelle en scheme

```
((lambda(i)
   ((lambda(x) (* x 0)) 3))
  2)
->
((lambda(x) (* x 0)) 3)
```

Portée et formulation fonctionnelle en scheme

```
((lambda(x) (* x 0)) 3)
->
(* 3 0)
->
0
```

Portée lexicale / dynamique – Conclusion

Portée lexicale et pureté fonctionnelle

- La portée lexicale respecte la transparence référentielle en associant à chaque fonction l'environnement lexical définissant leurs variables libres
- La portée lexicale correspond au calcul purement fonctionnel et les constructions d'environnements locaux peuvent être simulées par des applications de fonctions.

Portée dynamique et extensibilité

- La portée dynamique apporte davantage d'extensibilité que la portée lexicale.

Extensibilité – cas lexical (RMS - emacs-lisp)

- Fonction A — Extension utilisateur
  - Définit la variable FOO
  - Appelle la fonction B – avec paramètre FOO
- Fonction B — Système standard
  - Appelle la fonction C – avec paramètre FOO
- Fonction C — Extension utilisateur
  - Utilise la variable FOO

La solution consistant à utiliser un paramètre pour faire passer la variable FOO depuis la fonction A vers la fonction C revient à modifier l'appel de la fonction C dans la fonction B et en conséquence, cet appel étant générique, tous les appels aux autres commandes.

Extensibilité – cas dynamique (RMS - emacs-lisp)

- Fonction A — Extension utilisateur
  - Définit la variable FOO
  - Appelle la fonction B – variable à portée dynamique
- Fonction B — Système standard
  - Appelle la fonction C
- Fonction C — Extension utilisateur
  - Utilise la variable FOO

La solution consistant à utiliser une variable à portée dynamique dans la fonction A n'implique aucun changements pour la fonction B ni pour les autres commandes existantes.

Portée dynamique – Richard Stallman

Some language designers believe that dynamic binding should be avoided, and explicit argument passing should be used instead. Imagine that function A binds the variable FOO, and calls the function B, which calls the function C, and C uses the value of FOO. Supposedly A should pass the value as an argument to B, which should pass it as an argument to C.

This cannot be done in an extensible system, however, because the author of the system cannot know what all the parameters will be. Imagine that the functions A and C are part of a user extension, while B is part of the standard system. The variable FOO does not exist in the standard system; it is part of the extension. To use explicit argument passing would require adding a new argument to B, which means rewriting B and everything that calls B. In the most common case, B is the editor command dispatcher loop, which is called from an awful number of places.

What's worse, C must also be passed an additional argument. B doesn't refer to C by name (C did not exist when B was written). It probably finds a pointer to C in the command dispatch table. This means that the same call which sometimes calls C might equally well call any editor command definition. So all the editing commands must be rewritten to accept and ignore the additional argument. By now, none of the original system is left!