

ENS Rennes  
L3 Informatique, parcours R & I  
Programmation avancée, 2<sup>e</sup> semestre 2015–2016

---

TD 8 : Évaluation statique et clôtures

Luc Bougé

1<sup>er</sup> avril 2016 \*

## Introduction

Le TD précédent nous a permis de découvrir le langage Lisp et son algorithme d'interprétation implémenté en Caml. Cet algorithme utilisait le principe de la *liaison dynamique*. Un objet est évalué dans l'environnement présent au moment de son évaluation. Nous avons vu que cette approche permet de traiter de manière très élégante la récursivité. En effet, il n'est pas nécessaire qu'une fonction soit définie pour que son appel soit inclus dans le code d'une autre fonction. C'est grâce à cette propriété remarquable que le langage Lisp est immédiatement devenu célèbre à une époque où les langages classiques (Fortran, Cobol) ne permettaient pas de manipuler des fonctions récursives. Lisp et les langages interprétés ont d'ailleurs été associés à la récursivité jusque dans les années 1980, alors qu'il est aujourd'hui clair que ces deux notions sont indépendantes.

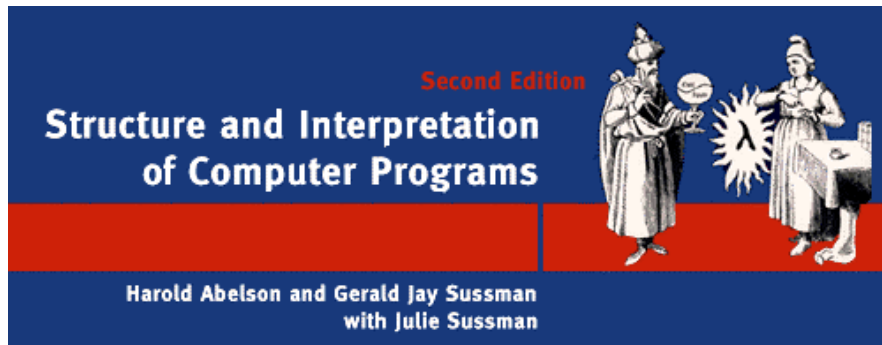


Ce principe de liaison dynamique conduit à un interprète simple, (assez) intuitif et efficace. Par contre, il est peu adapté à des développements logiciels de grande taille. En effet, le comportement d'une fonction dépend de son environnement, de manière plus précise de la valeur de ses variables libres au moment de son exécution. Ceci contredit fondamentalement l'approche de programmation par module, puisque le comportement d'un module ne doit être complètement déterminé par son code interne, indépendamment de son environnement. En sémantique, on parle d'un manque de *compositionnalité*. L'exemple le plus connu de grand projet logiciel développé en Lisp est *Emacs*, mais il a récemment introduit une notion de liaison statique, appelée ici *liaison lexicale*, pour assurer sa stabilité. Voir

<https://www.emacswiki.org/emacs/DynamicBindingVsLexicalBinding>

---

\*Id: td.tex 2026 2016–04–01 11:08:27Z bouge



Since April 1, 2011 (not an April fools joke), Emacs 24 has merged the lexical binding branch mentioned below. The trunk code is moving along and becoming lexical-binding-tolerant.

L'objectif de ce TD est de découvrir le principe de la liaison statique ou lexicale et de transformer l'interprète pour l'implémenter. Nous utilisons dans un premier temps l'interprète d'exercice pour ensuite examiner sa mise en œuvre dans les langages réels : Common Lisp, Scheme, Caml, etc.

La référence pour ce TD est le livre *Structure and Interpretation of Computer Programs*, Abelson, Sussman, and Sussman, MIT Press. Il est en ligne à l'adresse <https://mitpress.mit.edu/sicp/>. Le matériel de ce TD est discuté en section 3.2 *The Environment Model of Evaluation* sous une forme un peu différente.

Une présentation très détaillée de l'évaluation par liaison dynamique, par liaison statique ou lexicale et l'évaluation paresseuse se trouve dans le livre de Christian Queinnec, *Les langages LISP*, InterEditions. Christian Queinnec est le spécialiste français du domaine. Le livre est aussi disponible en anglais sous le titre *Lisp in Small Pieces*.

## 1 Clôtures

Examinons ce qui se passe pour les variables libres des fonctions dans l'interprète à liaison dynamique de la dernière fois.

```
LISP? (setq a 0)
```

```
SET: a = 0
```

```
LISP? (setq f (lambda () a))
```

```
SET: f = (lambda () a)
```

```
LISP? (f)
```

```
0
```

```
LISP? (setq a 1)
```

```
SET: a = 1
```

```
LISP? (f)
```

La valeur du symbole `a` qui est utilisée est celle active au moment de l'évaluation de la fonction. Ce principe d'évaluation est appelé *liaison dynamique*.

On peut refaire l'expérience en activant le mode debug :

```
LISP? (setq a 2)
```

```
SET: a = 2
```

```
LISP? (f)
```

```
0 --> (f) | "a" = 2; "a" = 1; "f" = (... ..); "a" = 0;
```

```
1 --> f | "a" = 2; "a" = 1; "f" = (... ..); "a" = 0;
```

```
1 <-- (lambda () a)
```

```
1 --> a | "a" = 2; "a" = 1; "f" = (... ..); "a" = 0;
```

```
1 <-- 2
```

```
0 <-- 2
```

La sémantique de la fonction `f` n'est donc pas *compositionnelle*. Elle ne dépend pas seulement de la définition de `f`, mais aussi de la valeur de `a` au moment de son exécution. Cette valeur est imprévisible au moment de la définition de la fonction. De plus, elle peut varier entre des exécutions successives de la fonction.

Une solution serait par exemple de remplacer syntaxiquement, *lexicalement*, le symbole `a` par sa valeur au moment de la définition de `f`. Mais cette approche est beaucoup trop compliquée car il n'y a aucune raison que ce symbole soit *explicite* dans le texte de la fonction. Il se pourrait par exemple qu'il soit synthétisé au cours de l'exécution de `f`, à la manière de la fonction (**gensym**) par exemple.

Une idée plus intéressante est de faire une photographie (*snapshot*) de l'environnement au moment de la définition de `f`, de l'associer à cette définition et de restaurer cet environnement lors de l'exécution en remplacement de l'environnement courant. Ainsi, la fonction sera bien exécutée dans son environnement de définition. Sa sémantique ne dépendra que de son texte, *indépendamment* de son environnement d'exécution. Ce principe d'évaluation est appelé *liaison lexicale*.

L'association d'un environnement à un objet est appelé une *clôture*. On dit que l'environnement a été *capturé* lors de la définition de la création de la clôture.

Voici par exemple ce que ça donne avec l'interprète `MLlisp_static`.

```
MLlisp_static? (define a 1)
```

```
DEFINE: a = 1
```

```
MLlisp_static? (define f (lambda () a))
```

```
DEFINE: f = (closure (lambda () a) <env>)
```

```
MLlisp_static? (define a 2))
```

```
DEFINE: a = 2
```

```
MLlisp_static? (f)
```

```

0 --> (f) | "a" = 2; "f" = (... ..); "a" = 1;
1 --> f | "a" = 2; "f" = (... ..); "a" = 1;
1 <-- (closure (lambda () a) "a" = 1; )
1 --> a | "a" = 1;
1 <-- 1
0 <-- 1

```

1

On peut noter que l'évaluation de `f` est lancée dans l'environnement dans lequel `a` vaut 2. Mais lorsque la clôture est ouverte, l'environnement capturé au moment de la définition de `f` est restauré : `a` vaut 1. C'est bien cet environnement qui est utilisé pour évaluer `a`.



*On utilise ici `define` au lieu de `setq` pour des raisons expliquées plus loin.*

On peut comparer ce comportement à celui de Caml :

```

# let a = 1;;
val a : int = 1
# let f = fun () -> a;;
val f : unit -> int = <fun>
# f();;
- : int = 1
# let a = 2;;
val a : int = 2
# f();;
- : int = 1

```

Dans l'interprète à liaison lexicale, la valeur d'une  $\lambda$ -expression n'est donc pas l'expression elle-même comme dans l'interprète à liaison dynamique, mais bien la clôture constituée de l'expression et de l'environnement courant.

```

MLlisp_static? (define a 0)
DEFINE: a = 0

MLlisp_static? (lambda f () a)
(closure (lambda f () a) <env>)

```

Dans cet interprète, cette clôture est implémentée par une liste dont le premier symbole est `closure`.



*L'environnement n'est pas un objet du monde Lisp. Il est ici injecté dans le monde Lisp, un peu à la manière des fonctions prédéfinies `subr`, mais il n'y a aucun moyen de le manipuler explicitement ou de le représenter.*

## 2 Clôture et directives de définition

Réfléchissons maintenant à la nature de l'environnement qui est capturé.

Une première idée serait de copier l'environnement au moment de la définition de la

clôture, de manière qu'il soit complètement indépendant des évolutions ultérieures de l'environnement. Ce serait possible, mais évidemment très coûteux.

Une seconde idée est de garder juste une référence (un pointeur) vers l'environnement courant, mais en laissant celui-ci évoluer ultérieurement. C'est l'approche habituelle.

Ceci exige alors de bien spécifier l'action des directives de définition. Comment est-ce qu'une définition modifie l'environnement ?

Notre interprète distingue deux types de modification. La directive `define` ajoute une liaison en tête de la liste courante des liaisons. Le reste de l'environnement n'est pas modifié.

```
MLlisp_static? (define a 1)
DEFINE: a = 1
```

```
MLlisp_static? (printenv)
"a" = 1; ()
```

```
MLlisp_static? (define a 2)
DEFINE: a = 2
```

```
MLlisp_static? (printenv)
"a" = 2; "a" = 1; ()
```

Par contre, la directive `setq` modifie physiquement l'environnement courant. Elle met à jour la liaison *active*, c'est-à-dire la plus *récente*, du symbole considéré. S'il n'y a pas de liaison active, elle crée une liaison comme le ferait la directive `define`.

```
MLlisp_static? (printenv)
"a" = 2; "a" = 1; ()
```

```
MLlisp_static? (setq a 3)
SET: a = 3
```

```
MLlisp_static? (printenv)
"a" = 3; "a" = 1; ()
```



*Il n'y a aucun moyen de modifier les liaisons de l'environnement qui ne sont pas actives.*

Il est donc impossible de définir des fonctions récursives avec `define`. En effet, l'environnement qui est capturé lors de la définition ne contient pas la liaison engendrée par cette définition !

```
MLlisp_static? (define fact (lambda (n) (if (= n 0) 1 (* n (fact (- n 1))))))
DEFINE: fact = (closure (lambda (n) (if (... ..) 1 (... ..))) <env>)
```

```
MLlisp_static? (fact 2)
0 --> (fact 2) | "fact" = (... ..);
1 --> 2 | "fact" = (... ..);
```

```

1 <-- 2
1 --> fact | "fact" = (... ..);
1 <-- (closure (lambda (n) (if (... ..) 1 (... ..)))) )
1 --> (if (= n 0) 1 (* n (fact (... ..)))) | "n" = 2;
2 --> (= n 0) | "n" = 2;
3 --> 0 | "n" = 2;
3 <-- 0
3 --> n | "n" = 2;
3 <-- 2
3 --> = | "n" = 2;
3 <-- <subr>
2 <-- ()
2 --> (* n (fact (- n 1))) | "n" = 2;
3 --> (fact (- n 1)) | "n" = 2;
4 --> (- n 1) | "n" = 2;
5 --> 1 | "n" = 2;
5 <-- 1
5 --> n | "n" = 2;
5 <-- 2
5 --> - | "n" = 2;
5 <-- <subr>
4 <-- 1
4 --> fact | "n" = 2;

```

Lisp **error!** Undefined variable: fact

Vous pouvez comparer cette situation à ce qui se passe en Caml.

```

# let fact = fun n -> 0;;
val fact : 'a -> int = <fun>
# let fact = fun n -> if n = 0 then 1 else n * fact(n - 1);;
val fact : int -> int = <fun>
# fact 2;;
- : int = 0

```

Le symbole fact interne à la définition récursive est associée à la valeur de fact active au moment de la définition.

Heureusement, une particularité de notre implémentation des clôtures va nous permettre de contourner cette difficulté. En effet, la capture de l'environnement est simplement une référence vers l'environnement courant. Ainsi, les modifications faites plus tard à l'environnement pourront être visibles à *l'intérieur de la clôture*.

```

MLlisp_static? (define a 0)
DEFINE: a = 0

MLlisp_static? (define f (lambda () a))
DEFINE: f = (closure (lambda () a) <env>)

MLlisp_static? (f)

```

```

0 --> (f) | "f" = (... ..); "a" = 0;
1 --> f | "f" = (... ..); "a" = 0;
1 <-- (closure (lambda () a) "a" = 0; )
1 --> a | "a" = 0;
1 <-- 0
0 <-- 0

```

0

```
MLlisp_static? (setq a 1)
```

```
SET: a = 1
```

```
MLlisp_static? (printenv)
```

```
"f" = (... ..); "a" = 1; ()
```

```
MLlisp_static? (f)
```

```

0 --> (f) | "f" = (... ..); "a" = 1;
1 --> f | "f" = (... ..); "a" = 1;
1 <-- (closure (lambda () a) "a" = 1; )
1 --> a | "a" = 1;
1 <-- 1
0 <-- 1

```

1

La liaison du symbole a a été modifiée dans la partie de l'environnement capturée lors de la définition de f. Le résultat de (f) est bien 1 et non plus 0.

Grâce à cette particularité, il est maintenant possible de définir des fonctions récursives. Il suffit d'utiliser la directive setq !

L'idée est de créer une première définition du symbole fact. Il ne sert qu'à réserver sa place dans l'environnement. On l'appelle un *placeholder*. Ensuite, la clôture de la  $\lambda$ -expression est créée. L'environnement capturé contient la liaison de fact avec cette première définition. Finalement, la directive setq va modifier physiquement cette définition en remplaçant la valeur du symbole fact par cette nouvelle définition. Cette modification est visible à l'intérieur de la clôture puisqu'elle concerne une liaison qui a été faite avant la clôture.

```
MLlisp_static? (define fact ())
```

```
DEFINE: fact = ()
```

```
MLlisp_static? (setq fact (lambda (n) (if (= n 0) 1 (* n (fact (- n 1))))))
```

```
SET: fact = (closure (lambda (n) (if (... ..) 1 (... ..))) <env>)
```

```
MLlisp_static? (fact 10)
```

```
3628800
```



*L'environnement est maintenant une structure circulaire !*

On peut comparer ce comportement avec celui de let rec en Caml :

```
# let rec fact = fun n -> if n = 0 then 1 else n * fact(n - 1);;
val fact : int -> int = <fun>
# fact 10;;
- : int = 3628800
```

---

Voici donc le mystère du let rec enfin éclairci, ô futur chercheurs !

**Question 2.1** *Et maintenant, à vous de jouer pour des fonctions mutuellement récursives :*

```
# let rec f = fun n -> if n = 0 then 0 else g (n - 1)
  and g = fun n -> if n = 0 then 1 else f (n - 1);;
val f : int -> int = <fun>
val g : int -> int = <fun>
# f 1000;;
- : int = 0
# g 1000;;
- : int = 1
```

---

### 3 Application : continuations

La programmation par continuation repose crucialement sur le principe de la liaison lexicale. En effet, la définition d'une continuation doit capturer la valeur des variables libres au moment de cette définition, et non dépendre de leurs valeurs lors de son exécution.

Examinons d'abord un cas simple avec l'interprète à liaison dynamique.

```
LISP? (define print_cont (lambda (n k) (progn (print n) (newline) (k))))
SET: print_cont = (lambda (n k) (progn (print n) (newline) (k)))

LISP? (define k (lambda () (progn (print "Stop!") (newline))))
SET: k = (lambda () (progn (print Stop!) (newline)))

LISP? (print_cont 3 k)
3
Stop!
()
```

---

Lorsque la continuation est évaluée, le symbole `n` est lié à la valeur 3.

Supposons maintenant que ce symbole soit une variable libre de la continuation.

```
LISP? (define print_cont (lambda (n k) (progn (print n) (newline) (k))))
SET: print_cont = (lambda (n k) (progn (print n) (newline) (k)))

LISP? (define n 0)
SET: n = 0

LISP? (define k (lambda () (progn (print "n_□=□") (print n) (newline))))
```



```
SET: k = (lambda () (progn (print n = ) (print n) (newline))))
```

```
LISP? (k)
n = 0
()
```

```
LISP? (print_cont 3 k)
3
n = 3
()
```

---

Nous voyons ici que la variable libre de la continuation est *capturée* dans son environnement d'exécution. Ce comportement n'est pas correct. L'évaluation de la continuation ne doit pas dépendre du contexte dans lequel elle est appelée.

Il faut donc que cette variable libre retrouve la valeur qu'elle avait lors de la définition de la continuation. C'est bien le cas avec une évaluation à liaison lexicale.

```
MLlisp_static? (define print_cont (lambda (n k) (progn (print n) (newline) (k))))
DEFINE: print_cont = (closure (lambda (n k) (progn (... ...) (... ...) (...))) <env>)
```

```
MLlisp_static? (define n 0)
DEFINE: n = 0
```

```
MLlisp_static? (define k (lambda () (progn (print "n_=") (print n) (newline))))
DEFINE: k = (closure (lambda () (progn (... ...) (... ...) (...))) <env>)
```

```
MLlisp_static? (k)
n = 0
()
```

```
MLlisp_static? (print_cont 3 k)
3
n = 0
()
```

---

Examinons maintenant le cas plus complexe de la factorielle. Utilisons d'abord l'interprète à liaison dynamique.

```
LISP? (define fact (lambda (n k) (if (= n 0) (k 1) (fact (- n 1) (lambda (r) (k (* n r)))))))
SET: fact = (lambda (n k) (if (= n 0) (k 1) (fact (... ...) (... ...) (...))))
```

```
LISP? (define k (lambda (n) (progn (print "n_=") (print n) (newline))))
SET: k = (lambda (n) (progn (print n = ) (print n) (newline)))
```

```
LISP? (k 3)
n = 3
()
```

```
LISP? (fact 0 k)
n = 1
()
```

```
LISP? (fact 1 k)
;;; Loop... :- (
```

---


L'évaluation boucle car, au moment de l'appel de (fact 0), le symbole k est lié à

(lambda (r) (k (\* n r))).

Par le principe de la liaison dynamique, il s'agit donc d'une fonction récursive... qui ne termine pas.

**Question 3.1** Tracez l'évaluation pour vérifier ce phénomène déroutant.

Reprenons maintenant les mêmes définitions avec l'interprète à liaison lexicale.

 Il faut bien sûr faire attention à rendre la fonction fact récursive.

```
MLlisp_static? (define fact ())
DEFINE: fact = ()

MLlisp_static? (setq fact (lambda (n k) (if (= n 0) (k 1)
                                           (fact (- n 1) (lambda (r) (k (* n r)))))))
SET: fact = (closure (lambda (n k) (if (... ..) (... ..) (... ..))) <env>)

MLlisp_static? (define k (lambda (n) (progn (print "n_=") (print n) (newline))))
DEFINE: k = (closure (lambda (n) (progn (... ..) (... ..) (...))) <env>)

MLlisp_static? (k 3)
n = 3
()

MLlisp_static? (fact 0 k)
n = 1
()

MLlisp_static? (fact 10 k)
n = 3628800
()
```

---

Lors de l'appel récursif, de (fact 0), le symbole k est lié à (lambda (r) (k (\* n r))), mais lors de l'évaluation de cette expression, le symbole k sera bien lié à sa valeur initiale capturée lors de la définition de l'expression.

**Question 3.2** Tracez l'évaluation pour vérifier cette affirmation.

**Question 3.3** Étudiez le cas de la fonction Fibonacci.

**Question 3.4** Étudiez le cas de la fonction *append*.

**Question 3.5** Nous avons ici détaillé le cas des continuations. Un problème analogue se pose avec toutes les fonctionnelles. Le comportement des fonctions arguments des fonctionnelles ne doit pas dépendre des variables internes utilisées pour la programmation des fonctionnelles.


Mettez ce problème en évidence avec les fonctionnelles *map* et *reduce*.

## 4 Modification de l'interprète

Le but de cette section est de modifier pas à pas l'interprète à liaison dynamique pour obtenir un interprète à liaison lexicale.

### 4.1 Environnement

Le premier travail est d'augmenter le type des objets Lisp pour pouvoir manipuler l'environnement dans le monde Lisp. La définition se trouve dans le fichier Defs/types.ml.

 Nous aurons besoin de modifier physiquement les liaisons. Le champ *value* de *binding* doit donc être déclaré *mutable*.

```
type
  lisp_object =
  | Lisp_symbol of string
  | Lisp_string of string
  | Lisp_num of int
  | Lisp_subr of (lisp_object -> lisp_object)
  | Lisp_list of (lisp_object list)
  (** begin MLlisp_static *)
  | Lisp_env of environment
  (** end MLlisp_static *)

and

  (** begin MLlisp_static *)
  binding = {name: string; mutable value: lisp_object}
  (** end MLlisp_static *)

and

  environment = binding list
;;
```

Il faut définir dans le fichier Env/env.ml les fonctions nécessaires pour manipuler l'environnement.

```

let env_to_obj env = Lisp_env(env);;

let obj_to_env obj = match obj with
| Lisp_env(e) -> e
| _ -> error2 obj "Not_an_environment"
;;

let envp obj = match obj with
| Lisp_env(_) -> true
| _ -> false
;;

```

---

Il faut aussi ajouter une fonction pour modifier une liaison si elle existe.

```

let update_env_modify obj v (env:environment) =
  try
    let b = find_binding_env obj (env:environment)
    in b.value <- v; env
  with
    Lisp_no_such_binding -> extend_env obj v env
;;

```

---

## 4.2 Clôtures

Il faut ajouter dans le fichier Eval/closure.ml toutes les fonctions de gestion des clôture.

```

let lisp_closure = string_to_obj "closure";;

let make_closure body env =
  (cons lisp_closure
    (cons body
      (cons (env_to_obj env)
        nil)))
;;

let check_closure c =
  if not ((car c) = lisp_closure)
  then error2 c "Not_a_closure"
;;

let get_body_closure c =
  check_closure c;
  cadr c
;;

let get_env_closure c =
  check_closure c;

```

```
obj_to_env (caddr c)
;;
```

### 4.3 Évaluation

Il reste maintenant à modifier l'évaluateur dans le fichier Eval/eval.ml. Tout d'abord, il faut qu'un  $\lambda$ -expression soit évaluée en une clôture de l'environnement courant.

```
if (car obj) = lisp_lambda
then (make_closure obj env)
else if (car obj) = lisp_closure then obj
```



*Notez qu'une clôture est ici évaluée en elle-même.*

Il faut maintenant modifier la fonction **apply** pour prévoir le cas de l'application d'une clôture à une liste de valeurs. On extrait de la clôture son corps `new_f` et son environnement `new_env` et on applique récursivement **apply**.

```
if ((car f) = lisp_lambda)
then
  let body = (caddr f)
  and lpars = (cadr f)
  in
    let new_env = (extend_largs_env lpars lvals env)
    in (eval body new_env)

(** begin MLlisp_static *)
else if ((car f) = lisp_closure)
then
  let new_f = get_body_closure f
  and new_env = get_env_closure f
  in apply new_f lvals new_env
(** end MLlisp_static *)

else error2 f "Cannot_apply_a_list"
```

### 4.4 Directives

Il reste maintenant à définir spécifiquement la directive `setq` dans le fichier `toplevel.ml`. La seule différence est qu'on utilise maintenant la fonction `update_env_modify` au lieu de `update_env_extend` pour la directive `define`.

```
let handle_set obj =
  let defined_obj = (cadr obj)
  and defining_expr = (extract_defining_expr obj) in
  let defined_name = (obj_to_string defined_obj)
```

```

and defining_value = (eval defining_expr !global_env_ref)
in
begin
  print_string ("SET:␣" ^ defined_name ^ "␣=␣");
  print(defining_value);
  print_newline();
  global_env_ref := update_env_modify
    defined_obj defining_value !global_env_ref
end
;;

```

---

Il faut aussi prévoir ce que signifie une définition de fonction compacte comme

```
(define f (x) (+ x 1))
```

Ici, nous choisissons de la transformer en

```
(define f (quote (lambda (x) (+ x 1))))
```

mais c'est un choix assez arbitraire.

```

let extract_defining_expr obj =
  if (nullp (cdddr obj)) (* At most 3 objects *)
  then (caddr obj) (* Last object *)
  else
    begin
      let rest = (cddr obj) (* Rest but the two first objects *)
      in
        (** begin MLlisp_static *)
        (list2 lisp_quote (cons lisp_lambda rest))
        (** end MLlisp_static *)
    end
end
;;

```

---

## 4.5 Finalement...

Il ne reste plus qu'à changer le prompt. À vous de faire, ô futurs chercheurs !



Alors, ça marche ? ☺

## 5 Scheme

Scheme est, avec Common Lisp, le dialecte de Lisp le plus populaire. Il a été conçu au MIT dans les années 1970 par Gerald Jay Sussman et Guy L. Steele (<https://fr.wikipedia.org/wiki/Scheme>). Jay Sussman est l'un des auteurs du livre de référence SICP. Guy Steele a été

impliqué dans la conception de Common Lisp, Scheme, Fortran 90 et Java. Il a aussi aidé Richard Stallman dans sa première implémentation d'Emacs.

L'implémentation de référence de Scheme est celle du MIT, appelée MIT/GNU Scheme ([https://fr.wikipedia.org/wiki/MIT/GNU\\_Scheme](https://fr.wikipedia.org/wiki/MIT/GNU_Scheme)). Elle a été développée à l'origine par Hal Abelson, l'autre auteur du livre SICP.

❖ Le langage Scheme est défini sous la forme de rapports successifs nommés Revised<sup>n</sup> Report on the Algorithmic Language Scheme, où *n* est le numéro de la révision, et abrégés en R<sub>n</sub>RS. La version 7 est en préparation. Cependant, cette définition n'a pas de statut officiel et juridiquement contraignant.

Le point important qui nous concerne ici est que la sémantique du langage est entièrement centrée sur le principe de la liaison lexicale.

❖ Nous utilisons dans ce texte l'implémentation SISC, connue pour sa portabilité.

<http://sisc-scheme.org/>

*SISC is an extensible Java based interpreter of the algorithmic language Scheme. SISC uses modern interpretation techniques, and handily outperforms all existing JVM interpreters (often by more than an order of magnitude).*

*In addition, SISC is a complete implementation of the language. The entire R5RS Scheme standard is supported, no exceptions. This includes a full number tower including complex number support, arbitrary precision integers and floating point numbers, as well as hygienic R5RS macros, proper tail recursion, and first-class continuations (not just the escaping continuations as in many limited Scheme systems). SISC also attempts to implement the standard as correctly as possible, while still providing exceptional performance.*

Une introduction minimale au langage est disponible sur la page Wikipédia <https://fr.wikipedia.org/wiki/Scheme>. Le rapport R5RS est disponible à l'adresse

<http://www.schemers.org/Documents/Standards/R5RS/>

L'environnement Scheme est divisé en *frames*. À l'intérieur d'une frame, la définition d'une variable est une modification physique de la liaison, à la manière d'un setq. Par contre, à chaque nouvelle *frame*, on ajoute une nouvelle liaison à la manière d'un define. Ceci permet de traiter les définitions récursives de manière naturelle.

```
$ sisc
```

```
SISC (1.16.6)
```

```
#;> (define a 1)
```

```
#;> (define f (lambda () a))
```

```
#;> f
```

```
#<procedure f>
```

```
#;> (f)
```

```
1
```

```
#;> (define a 2)

#;> (f)
2

#;> (define fact (lambda (n) (if (= n 0) 1 (* n (fact (- n 1))))))
#;> (fact 10)
3628800
```

---

Les définitions récursives fonctionnent aussi.

```
#;> (define f (lambda (n) (if (= n 0) 0 (g (- n 1)))))
#;> (define g (lambda (n) (if (= n 0) 1 (f (- n 1)))))

#;> (f 1000)
0
#;> (g 1000)
1
```

---

Le principe est qu'une nouvelle *frame* est créée à chaque fois que l'on passe un  $\lambda$ , que ce soit explicitement ou par un **let**.

```
#;> (define g (lambda (a) (f)))

#;> (g 3)
2

#;> (let ((a 4)) (f))
2
```

---

Ceci permet de traiter les continuations de manière naturelle.

```
#;> (define fact (lambda (n k) (if (= n 0) (k 1)
                                   (fact (- n 1) (lambda (r) (k (* n r)))))))

#;> (define k (lambda (n) (begin (display "n = ") (display n) (newline))))

#;> (fact 10 k)
n = 3628800
```

---

Scheme propose donc un équilibre subtil entre la liaison dynamique et la liaison lexicale qui en fait un langage intéressant d'implémentation logicielle. Par exemple, c'est le langage de script du logiciel de traitement d'images *GIMP*.

La grande contribution sémantique de Scheme est l'introduction de la construction *call/cc* ou *call-with-current-continuation* qui permet de capturer *dynamiquement* une continuation. Scheme permet aussi l'évaluation paresseuse grâce à la construction *future*. Nous étudierons ces points dans les prochaines séances.