

Langage LISP

Référence technique et manuel utilisateur

Implémentation `let.c`

Version	1.0 — cours 5	
Date	Juillet 2026	Document généré à partir de <code>ref.html</code>
Environnement	<code>gcc -Wall -Wextra -std=c99</code>	

Table des matières

1	Présentation	5
1.1	Aperçu	5
1.2	Démarrage rapide	5
1.2.1	Compilation	5
1.2.2	Mode ligne de commande	5
1.2.3	REPL interactif	5
1.3	Conventions typographiques	6
2	Référence du langage	7
2.1	Arithmétique	7
2.1.1	Comparaisons	7
2.1.2	Fonctions mathématiques	7
2.2	Logique	8
2.3	Liaison et mutation	8
2.4	Contrôle	8
2.5	Fonctions	9
2.6	Cellules mutables	9
2.7	Listes	9
2.8	Entrée / sortie	10
2.9	Persistance	10
3	Concepts avancés	11
3.1	Objets par passage de messages	11
3.2	Environnements et closures	12
4	Bibliothèque standard — prolog.lisp	13
4.1	Arithmétique et calcul	13
4.2	Générateurs de listes	13
4.3	Accès et transformation	13
4.4	Higher-order functions	14
4.5	Prédicats et analyse	14
5	Fonctions natives ($C \rightarrow LISP$)	15
6	Récapitulatif syntaxique	17
6.1	Tableau de toutes les primitives	17

Chapitre 1

Présentation

1.1 Aperçu

`let` est un interpréteur LISP minimal écrit en C99, conçu comme support pédagogique pour un cours de compilation, de machine virtuelle et de parseur. Il fournit un REPL interactif ainsi qu'un mode ligne de commande.

Caractéristiques principales

- Évaluateur récursif d'expressions en s-expressions.
- Environnements chaînés (pile de portées) avec portée lexicale.
- Closures avec capture de l'environnement (*snapshot*).
- Cellules mutables (`cell` / `cell-ref`) pour l'état partagé.
- Objets par passage de messages (style Smalltalk).
- Récursion naturelle (pas de `loop` / `recur`).
- Bibliothèque standard `prolog.lisp` (21 fonctions).

1.2 Démarrage rapide

1.2.1 Compilation

```
gcc -Wall -Wextra -std=c99 -lm sexpression.c let.c -o let
```

1.2.2 Mode ligne de commande

```
# valuer une expression
./let '(+ 1 2)'
# Plusieurs expressions
./let '(let x 10)' '(+ x 2)' '(print x)'
```

1.2.3 REPL interactif

```
./let
> (let x 42)
-> 42
> (+ x 1)
-> 43
> exit
```

Pour quitter le REPL, tapez `exit` ou `quit`, ou envoyez EOF (Ctrl+D).

1.3 Conventions typographiques

Notation	Signification
<code>(op a b)</code>	Signature de la primitive
<i>nom</i>	Argument obligatoire
<code>code</code>	Fragment de code LISP ou C
<i>valeur</i>	Résultat attendu
mot-clé	Mot-clé réservé du langage

Chapitre 2

Référence du langage

2.1 Arithmétique

Les opérateurs arithmétiques acceptent des entiers et des flottants. Si l'un des opérandes est un flottant, le résultat est un flottant.

```
(+ a b) — Addition  
(- a b) — Soustraction  
(* a b) — Multiplication  
(/ a b) — Division entière (erreur si b = 0)
```

```
(+ 2 3)      ; -> 5  
(- 10 3)     ; -> 7  
(* 6 7)     ; -> 42  
(/ 15 4)     ; -> 3
```

2.1.1 Comparaisons

```
(> a b) — Supérieur  
(< a b) — Inférieur  
(>= a b) — Supérieur ou égal  
(<= a b) — Inférieur ou égal  
(= a b) — Égalité numérique
```

Les comparaisons retournent 1 (vrai) ou 0 (faux).

```
(> 5 3)      ; -> 1  
(< 2 8)      ; -> 1  
(≥ 4 4)      ; -> 1  
(= 42 42)    ; -> 1
```

2.1.2 Fonctions mathématiques

```
(sqrt x) — Racine carrée (retourne un flottant).  
(sin x) — Sinus trigonométrique (radians, flottant).  
(random) — Nombre flottant pseudo-aléatoire dans [0, 1).
```

```
(sqrt 25)    ; -> 5  
(sin 0)      ; -> 0
```

```
(random) ; -> 0.374540 (exemple)
```

2.2 Logique

(**and** *e1 e2 ...*) — ET logique (court-circuit).
(**or** *e1 e2 ...*) — OU logique (court-circuit).
(**not** *a*) — Négation logique.
(**eq?** *a b*) — Égalité entre entiers ou symboles.

and et **or** acceptent un nombre variable d'arguments et utilisent l'évaluation à court-circuit : dès qu'un résultat définitif est connu, les arguments restants ne sont pas évalués.

```
(and 1 0) ; -> 0  
(or 0 1) ; -> 1  
(not 0) ; -> 1  
(eq? 'a 'a) ; -> 1  
(eq? 42 42) ; -> 1
```

2.3 Liaison et mutation

(**let** *sym expr*) — Crée une nouvelle portée avec la liaison *sym* = *expr*. Retourne la valeur liée.
(**set!** *sym expr*) — Modifie la liaison existante la plus proche. Si la valeur liée est une cellule, son contenu est modifié sur place. Erreur si *sym* n'existe pas dans aucun cadre.

Portées imbriquées

Chaque appel à **let** crée un *nouveau* cadre de portée. Si *x* existe déjà dans un cadre parent, une nouvelle liaison *x* est créée dans le cadre local sans modifier le cadre parent. Utilisez **set!** pour modifier une liaison existante.

2.4 Contrôle

(**if** *cond then else*) — Conditionnelle.
Si *cond* ≠ 0, évalue et retourne *then*, sinon *else*.
(**begin** *e1 e2 ...*) — Séquence. Évalue les expressions d'ordre et retourne la dernière.
(**quote** *expr*) — Retourne *expr* sans l'évaluer.
Sucre syntaxique : *'expr*.
(**null?** *x*) — Teste si *x* est la liste vide ().

```
(if 1 2 3) ; -> 2  
(if 0 'vrai 'faux) ; -> faux  
(begin (display 1) (display 2) 3) ; -> 3 (affiche 12)  
'(a b c) ; -> (a b c)  
(null? (list)) ; -> 1
```

2.5 Fonctions

`(lambda (params) corps)` — Crée une closure anonyme. Retourne la fonction elle-même.

`(function nom (params) corps)` — Crée une closure nommée, l'enregistre dans l'environnement courant, et la retourne. Supporte la récursion directe.

```
; Lambda (anonyme)
((lambda (x) (* x 2)) 21)    ; -> 42

; Function (nommée)
(function f (x) (* x 2))
(f 21)                      ; -> 42

; Recursion
(function fact (n)
  (if (< n 2) 1 (* n (fact (- n 1)))))
(fact 10)                   ; -> 3628800
```

Closures

`lambda` et `function` créent des *closures* : la fonction capture l'environnement lexical au moment de sa définition. Une closure peut ainsi accéder aux variables du cadre dans lequel elle a été créée, même après que ce cadre n'est plus le cadre courant.

2.6 Cellules mutables

`(cell valeur)` — Crée une cellule mutable contenant `valeur`. Allouée sur le tas avec compteur de références.

`(cell-ref cell)` — Lit la valeur d'une cellule.

Partage de cellules

Quand on copie une cellule via `sexp_copie`, le compteur de références est incrémenté. La mémoire n'est libérée que lorsque plus aucune référence n'existe. Deux closures peuvent ainsi partager la même cellule — c'est le mécanisme de base des objets.

2.7 Listes

Les listes sont construites à partir de paires `cons` (listes à lien unique).

`(list e1 e2 ...)` — Construit une liste avec les éléments évalués.

`(cons a b)` — Ajoute `a` en tête de la liste `b`.

`(car lst)` — Premier élément d'une liste. Erreur si vide.

`(cdr lst)` — Reste de la liste. () si un seul élément.

```
(list 1 2 3)                ; -> (1 2 3)
(car (list 10 20))          ; -> 10
(cdr (list 10 20 30))       ; -> (20 30)
(cons 1 (list 2 3))         ; -> (1 2 3)
```

Programmation fonctionnelle

Avec `function`, `if`, `cons`/`car`/`cdr` et la récursion, vous pouvez implémenter `map`, `filter`, `length`, `append`... comme en vrai LISP. La bibliothèque standard `prolog.lisp` fournit ces primitives prêtes à l'emploi.

2.8 Entrée / sortie

`(display expr)` — Affiche la valeur sans saut de ligne. Retourne la valeur.

`(print expr)` — Affiche la valeur suivie d'un saut de ligne. Retourne la valeur.

```
(display 42)      ; affiche 42 puis retourne 42
(print "hello")   ; affiche hello puis saute de ligne
```

2.9 Persistance

`(load "fichier")` — Charge et exécute un fichier LISP. Chaque forme est lue et évaluée séquentiellement.

`(save "fichier")` — Sauvegarde l'environnement courant dans un fichier au format LISP lisible, reproductible par `load`.

```
(load "prolog.lisp")      ; charge la bibliotheque standard
(save "monprog.lisp")     ; -> 1 (sauvegarde reussie)
```

Chapitre 3

Concepts avancés

3.1 Objets par passage de messages

Un *objet* en `let` est simplement une closure qui capture une cellule mutable et un dialogue basé sur des symboles (messages).

Exemple : compteur

Messages supportés : `'inc`, `'value`, `'reset`

```
(function make-compteur (init)
  (let c (cell init))
  (lambda (msg)
    (if (eq? msg 'inc)
        (begin (set! c (+ (cell-ref c) 1)) (cell-ref c))
        (if (eq? msg 'value) (cell-ref c)
            (if (eq? msg 'reset) (begin (set! c 0) (cell-ref c)) 0))))))

(let c (make-compteur 100))
(c 'inc)      ; -> 101
(c 'inc)      ; -> 102
(c 'value)    ; -> 102
(c 'reset)    ; -> 0
```

Exemple : banque

Messages : `'deposit`, `'withdraw`, `'balance`

```
(function make-bank (initial)
  (let solde (cell initial))
  (lambda (msg mt)
    (if (eq? msg 'deposit)
        (begin (set! solde (+ (cell-ref solde) mt))
                (cell-ref solde))
        (if (eq? msg 'balance) (cell-ref solde)
            0))))))

(let b (make-bank 1000))
(b 'deposit 500)      ; -> 1500
(b 'balance)          ; -> 1500
```

Mécanisme

Le pattern est toujours le même :

1. Créer une cellule mutable pour l'état interne.
2. Retourner une lambda qui décrypte le message (**eq?** sur un symbole).
3. Appliquer la closure pour « envoyer un message » : (**obj 'msg args...**).

3.2 Environnements et closures

L'interpréteur maintient une chaîne de portées (**env**) sous forme de listes d'association liées. Chaque **let** ou **function** crée un nouveau cadre dont le parent est le cadre courant.

Lorsqu'une closure est créée (**lambda** ou **function**), un *snapshot* de la chaîne d'environnements est capturé. Ce snapshot garantit que la closure conserve l'accès aux variables lexicales du moment de sa définition, même si l'environnement courant a changé entre-temps.

```
(let x 10)
(function f () x)      ; f capture x=10
(let x 20)
(f)                    ; -> 10 (pas 20 !)
```

Récursion

Pour supporter la récursion directe, **apply_closure** rattache temporairement l'environnement courant au bas de la chaîne capturée, puis détache avant le nettoyage. Cela permet à une fonction de s'appeler elle-même via son nom lié.

Chapitre 4

Bibliothèque standard — prolog.lisp

Le fichier `prolog.lisp` est chargé automatiquement au démarrage de `let` (silencieux si absent). Il définit 21 fonctions en LISP pur.

4.1 Arithmétique et calcul

`(fact n)` — Factorielle de *n* (retourne un flottant).
`(puissance x n)` — x^n (exponentiation entière, récursive).
`(cos x)` — Cosinus par série de Taylor (10 termes).
`(cos-iter x k n sign)` — Itération interne de `cos`.

```
(fact 5)           ; -> 120
(puissance 2 10)   ; -> 1024
(cos 0)            ; -> 1
```

4.2 Générateurs de listes

`(range start step stop)` — Liste de *start* à *stop* avec le pas *step*.
`(iota n)` — Entiers de 0 à *n* - 1.
`(repeat x n)` — *n* copies de *x*.

```
(range 0 2 8)      ; -> (0 2 4 6 8)
(iota 5)           ; -> (0 1 2 3 4)
(repeat 42 3)      ; -> (42 42 42)
```

4.3 Accès et transformation

`(length xs)` — Longueur d'une liste.
`(take n xs)` — Les *n* premiers éléments.
`(drop n xs)` — Supprime les *n* premiers éléments.
`(append xs ys)` — Concatène deux listes.
`(reverse xs)` — Inverse l'ordre d'une liste.

```
(length (iota 10)) ; -> 10
(take 3 (iota 10)) ; -> (0 1 2)
```

```
(drop 7 (iota 10))           ; -> (7 8 9)
(append '(1 2) '(3 4))       ; -> (1 2 3 4)
(reverse (iota 4))           ; -> (3 2 1 0)
```

4.4 Higher-order functions

(map *f* *vs*) — Applique *f* à chaque élément.
(filter *pred* *xs*) — Garde les éléments satisfaisant *pred*.
(zip *vs* *ws*) — Combine deux listes en paires.
(foldl *f* *init* *xs*) — Réduction gauche.
(foldr *f* *init* *xs*) — Réduction droite.
(compose *f* *g*) — Composition : retourne $\lambda x. f(g(x))$.

```
(map (lambda (x) (* x 2)) '(1 2 3))
; -> (2 4 6)

(filter (lambda (x) (> x 2)) '(1 2 3 4))
; -> (3 4)

(foldl (lambda (a x) (+ a x)) 0 (iota 100))
; -> 4950

((compose car cdr) '(1 2 3))
; -> 2
```

4.5 Prédicats et analyse

(any *pred* *xs*) — Au moins un élément satisfait *pred*?
(all *pred* *xs*) — Tous les éléments satisfont *pred*?
(deriv *f* *h*) — Dérivée numérique : retourne une approximation de *f*'.

```
(any (lambda (x) (> x 5)) '(3 7 2))           ; -> 1
(all (lambda (x) (> x 0)) '(1 2 3))           ; -> 1
((deriv (lambda (x) (* x x)) 0.001) 3)       ; -> 6.000
```

Chapitre 5

Fonctions natives ($C \rightarrow \text{LISP}$)

Ces fonctions sont écrites en C et enregistrées dans l'environnement global au démarrage de `let`. Elles s'utilisent exactement comme des fonctions LISP classiques — le langage ne fait pas la différence entre une closure et une native.

`(random)` — Nombre flottant pseudo-aléatoire dans $[0, 1)$.
`(sqrt x)` — Racine carrée. Accepte entier ou flottant.
`(sin x)` — Sinus trigonométrique (radians).

```
(random)      ; -> 0.374540 (exemple)
(sqrt 25)     ; -> 5
(sin 0)       ; -> 0
```

Architecture interne

Une native est une fonction C de type `NativeFn`. Le dispatch se fait via `is_native()` dans `evaluer_env`. Elle est stockée dans l'environnement comme n'importe quelle autre valeur, sous le type `SEXPR_NATIVE`.

Chapitre 6

Récapitulatif syntaxique

6.1 Tableau de toutes les primitives

Forme	Description
<code>(+ a b)</code>	Addition
<code>(- a b)</code>	Soustraction
<code>(* a b)</code>	Multiplication
<code>(/ a b)</code>	Division entière
<code>(> a b)</code>	Supérieur
<code>(< a b)</code>	Inférieur
<code>(>= a b)</code>	Supérieur ou égal
<code>(<= a b)</code>	Inférieur ou égal
<code>(= a b)</code>	Égalité numérique
<code>(sqrt x)</code>	Racine carrée
<code>(sin x)</code>	Sinus
<code>(random)</code>	Aléatoire $[0, 1)$
<code>(and e ...)</code>	ET logique
<code>(or e ...)</code>	OU logique
<code>(not a)</code>	Négation
<code>(eq? a b)</code>	Égalité
<code>(let sym expr)</code>	Liaison locale
<code>(set! sym expr)</code>	Mutation
<code>(if c t e)</code>	Conditionnelle
<code>(begin e ...)</code>	Séquence
<code>(quote e)</code>	Non-évaluation
<code>(null? x)</code>	Test liste vide
<code>(lambda (p) b)</code>	Closure anonyme
<code>(function n (p) b)</code>	Closure nommée
<code>(cell v)</code>	Cellule mutable
<code>(cell-ref c)</code>	Lecture de cellule
<code>(list e ...)</code>	Construction de liste
<code>(cons a b)</code>	Ajout en tête
<code>(car l)</code>	Premier élément
<code>(cdr l)</code>	Reste
<code>(display e)</code>	Affichage
<code>(print e)</code>	Affichage + NL
<code>(load f)</code>	Charger un fichier
<code>(save f)</code>	Sauvegarder l'env