Projet 3: Interpréteur LISP

Nicolas Bailluet, **Tom Bordin**, **Samuel Bouaziz** et **Rémi Piau**

5 mars 2019

ENS Rennes

Les sous-programmes

- opérateurs sur les entiers : `+`, `-`, `*`
- tests : `null`, `list`, `number`, `string`, `symbol`, `=`
- opérateurs sur les listes : `cons`, `car`, `cdr`
- affichage : `newline`, `display`

Les mots-clés

- conditionnels : `if`, `cond`
- gestion d'environnement : `printenv`, `setq`, `let`
- autres : `lambda`, `begin`, `quote`, `help`, `call/cc`

Les directives du Toplevel

- `define`, `definerec`, `load`

Sous-programmes modifiables

Les sous-programmes sont insérés dans l'environnement à l'initialisation.

```
% (printenv)
=> (printenv)
newline <- <subr>
display <- <subr>
list? <- <subr>
symbol? <- <subr>
number? <- <subr>
string? <- <subr>
null? <- <subr>
cdr <- <subr>
car <- <subr>
cons <- <subr>
= <- <subr>
* <- <subr>
- <- <subr>
+ <- <subr>
```

Ordre d'évaluation

- les éléments les plus profonds en premiers
- les paramètres sont évalués avant la vérification des types

```
% (+ (display "a") (newline))
=> (+ (display "a") (newline))
a
Lisp evaluation error: Cannot apply object_to_number: object a is not a number
```

Contraintes sur les paramètres

- pas assez de paramètres → erreur
- trop de paramètres → évaluation de tous les paramètres

```
% (+ 1 2 (display "a") (newline))
=> (+ 1 2 (display "a") (newline))
a
Got: 3
```

Différentes utilisations

- sans paramètres $\rightarrow$ affiche les informations sur tous les mots-clés
- un mot-clé en paramètre $\rightarrow$ affiche l'aide pour ce mot-clé

Messages d'aide

- propres aux **mots-clés**
- les messages sont pré-définis
- l'utilisateur ne peut **pas** en rajouter

```
% (help)
=> (help)
begin: Used to do multiple operations. The syntax is (begin (expr1) (expr2) ...).
 The result will be the result of the last expression.
call/cc: call with current continuation
cond: Evaluate the first expression of which the associate test is true, e.g. (co
nd (test1 expr1) (test2 expr2) (test3 expr3)).
help: Display this help message.
if: Used for condition. The syntax is (if (then) (else)). The else part is option
al.
lambda: Used to create functions. The syntax is (lambda (var) (expr)).
let: Used to create temporary variables, e.g. (define (a) (let ((b 1) (c 2)) (+ b
 c))).
luc: Watch it !
printenv: Display the current environment (binding list).
quote: By doing (quote (expr)), the expression expr will not be evaluated.
set!: (set! var expr) will set var to expr in the environment.
Got: ()


% (help quote)
=> (help quote)
By doing (quote (expr)), the expression expr will not be evaluated.
Got: ()
```

Utilisation

```
(load "init.lsp")
```

Fonctionnement

- chaque expression interprétée est affichée dans le terminal
- en cas d'erreur l'évaluation s'arrête et l'utilisateur reprend le contrôle

Imbrication de `load`

- possible
- aucune vérification de **dépendance cyclique**
- **attention aux boucles infinies**

```
% (load "init.lsp")
=> (load "init.lsp")
Loading: init.lsp

% => (define a 1)
Define : a = 1

% => (definerec fact (lambda (n) (if (= n 0) 1 (* n (fact (- n 1))))))
Define : fact = [closure | (lambda (n) (if (= n 0) 1 (* n (fact (- n 1)
)))) | <env>]

% => (fact 10)
Got: 3628800
```

Extensions

API de coloration

- une API pour la gestion des couleurs dans le terminal
- à chaque type une couleur associée
- alternance des couleurs des parenthèses en fonction de leur profondeur

```
% (define (f x) (begin (+ x 1) (display "f(x)")))
=> (define (f x) (begin (+ x 1) (display "f(x)")))
Define : f = [closure | (lambda (x) (begin (+ x 1)
(display "f(x)"))) | <env>]
```

Principe

- référence à l'environnement de définition sauvegardée → protège contre les effets de bord du contexte global
- la clôture est un objet LISP compatible avec `car`, `cdr`

Pour les fonctions récursives...

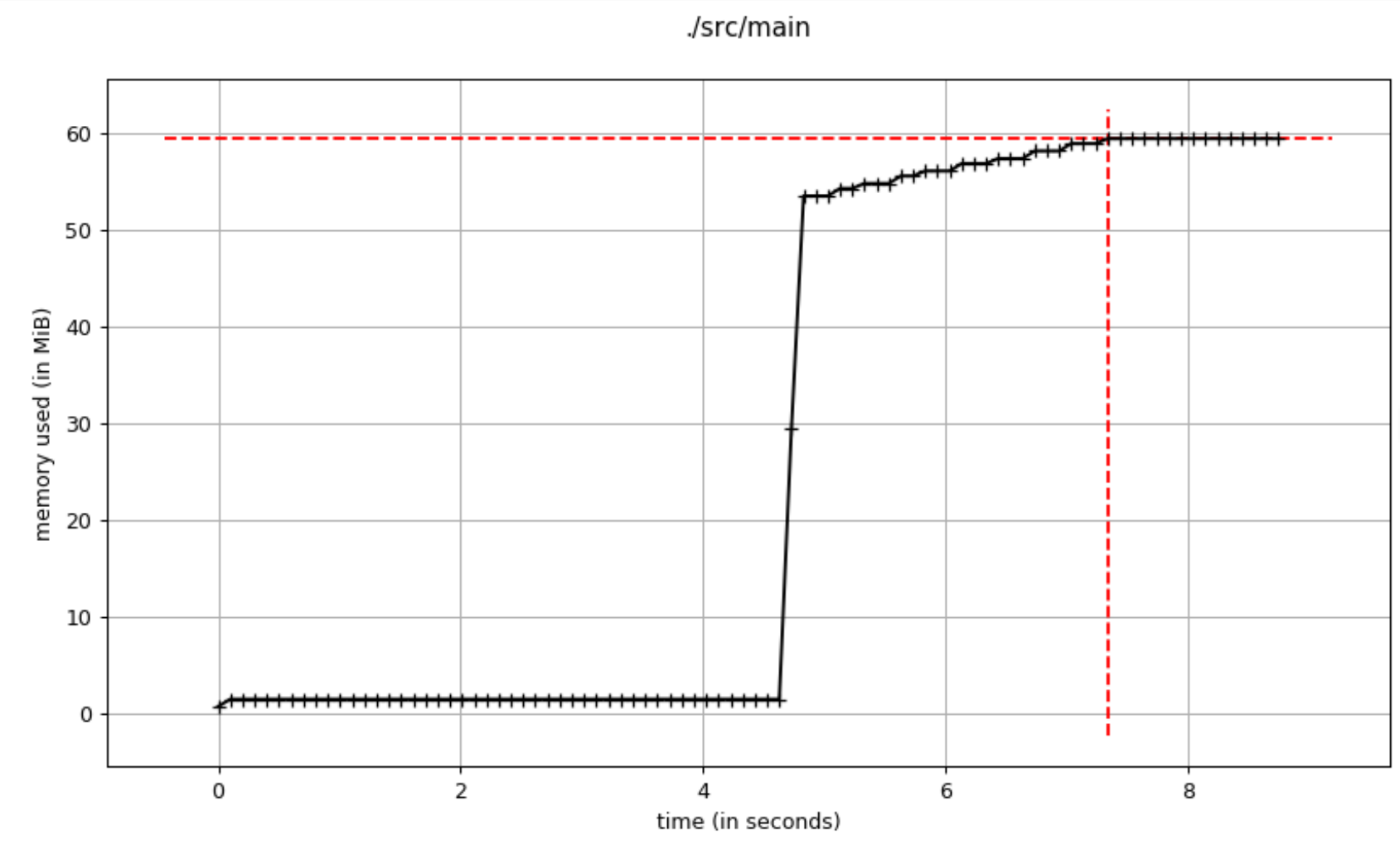
- ajout de la directive `definerec` afin d'éviter l'utilisation fastidieuse des *placeholders*

```
% (definerec fact (lambda (n) (if (= n 0) 1 (* n (fact (- n 1))))))
=> (definerec fact (lambda (n) (if (= n 0) 1 (* n (fact (- n 1))))))
Define : fact = [closure | (lambda (n) (if (= n 0) 1 (* n (fact (- n 1))
))) | <env>]

% (fact 5)
=> (fact 5)
Got: 120
```

Exemple de clôture

```
% (define a 1)
=> (define a 1)
Define : a = 1

% (define (f) a)
=> (define (f) a)
Define : f = [closure | (lambda () a) | <env>]

% (f)
=> (f)
Got: 1


% (define a 2)
=> (define a 2)
Define : a = 2

% (f)
=> (f)
Got: 1
```

```
% => (definerec fib (lambda (n) (if (= n 0) 1 (if (= n 1) 1 (+
(fib (- n 1)) (fib (- n 2)))))))
Define : fib = [closure | (lambda (n) (if (= n 0) 1 (if (= n 1)
 1 (+ (fib (- n 1)) (fib (- n 2))))))) | <env>]

% => (fib 12)
Got: 233
There are 6282 objects
% => (fib 12)
Got: 233
There are 12382 objects
% => (fib 12)
Got: 233
There are 18482 objects
% => (fib 12)
Got: 233
There are 24582 objects
% => (fib 12)
Got: 233
There are 30682 objects
```


./src/main
memory used (in MiB)
time (in seconds)
0
10
20
30
40
50
60
0
2
4
6
8

Principe du ramasse-miettes

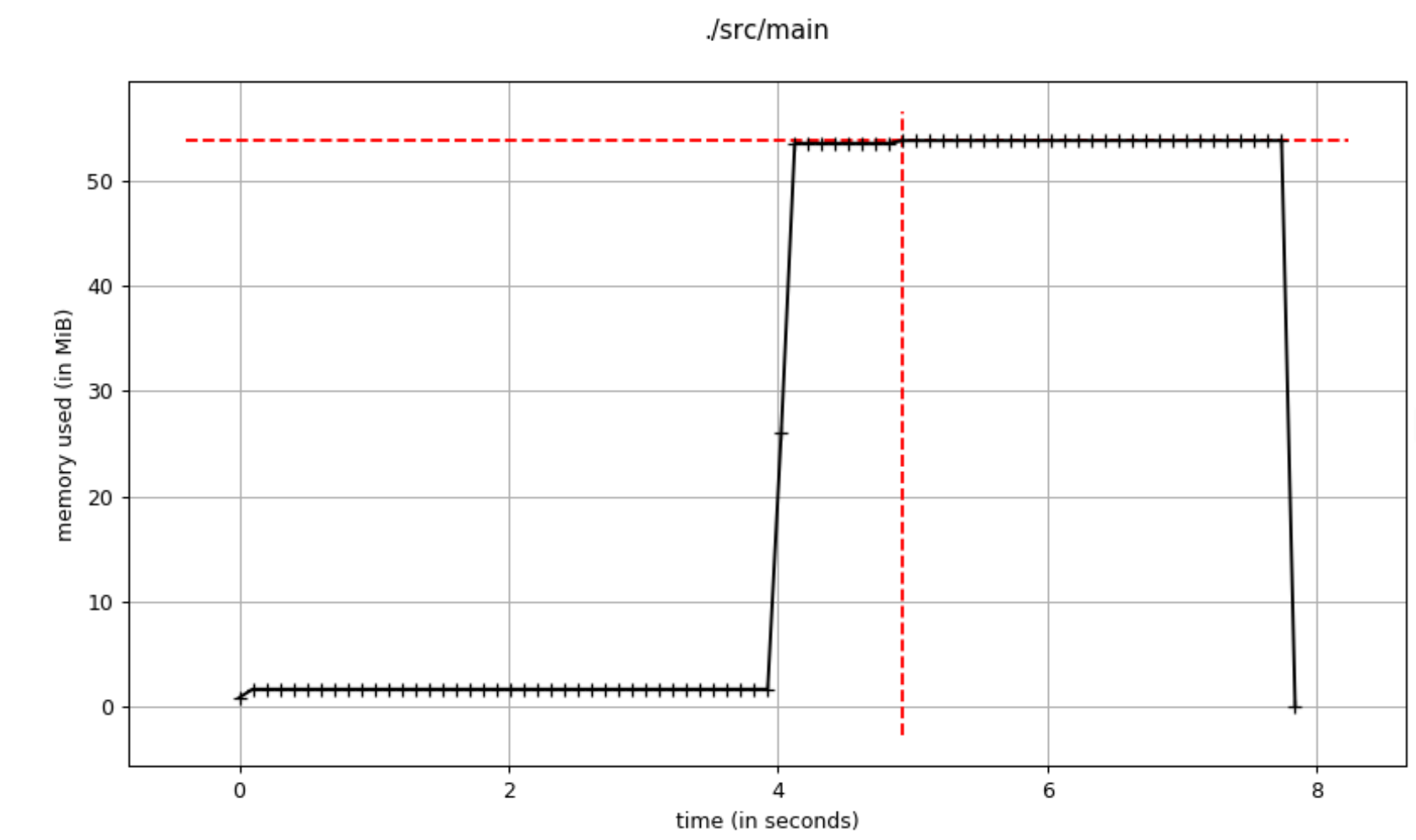
Après retour au **Toplevel** :

1. marquage des cellules de l'environnement (parcours en profondeur)
2. délétion des cellules non marquées

Traçage du nombre de cellules allouées

La mise en place d'un compteur permet de suivre la quantité de cellules et donc de mémoire utilisée.

```
% (fibo 12)
=> (fibo 12)
Got: 233
There were 6276 objects, freed 6145, down to 131
```


./src/main
memory used (in MiB)
time (in seconds)
0
10
20
30
40
50
0
2
4
6
8

Conclusion

Conclusion

Avantages

- Approche modulaire
- Gestion des continuations
- Coloration syntaxique
- Ramasse-miettes

Pour aller plus loin

- Recyclage dynamique (en cours d'évaluation)
- Auto-complétion et coloration en temps réel (ncurses)
- Limite de récursion (éviter les *stack-overflow*)

Annexes

La fonction `luc`