

# RAPPORT INCRÉMENT N°4

Vincent MEURISSE Thibault VINCENT, SICOM

23/11/2016

## Forme begin

Il s'agit simplement d'une forme permettant d'évaluer tous ses arguments dans l'ordre et de renvoyer l'évaluation du dernier argument. C'est l'équivalent d'un bloc d'instruction en C. On code cette forme très simplement, quelques lignes suffisent:

```
1
2 object sfs_begin(object input, object env){
3     input = cdr(input);
4     object eval_car = NULL;
5     while (input->type != SFS_NIL) {
6         eval_car = sfs_eval(car(input),env);
7         if (eval_car == NULL) /*évite une seg fault si erreur*/
8             return NULL;
9         input = cdr(input);}
10    return eval_car; }
```

## Implémentation du $\lambda$ -calcul

Nous allons ici détailler les étapes qui nous ont permis d'implémenter le  $\lambda$ -calcul dans notre interpréteur.

### Modification de `sfs_eval()` pour prendre un environnement en paramètre

Pour commencer nous avons modifié le code afin que la fonction d'évaluation prenne un environnement en paramètre. Pour cela il a fallu modifier les prototypes de beaucoup de fonctions afin qu'elles prennent elles aussi un environnement en paramètre. On a rajouté ce paramètre environnement à toutes les fonctions C qui faisaient appel à `sfs_eval()`. Il s'agit des fonctions d'évaluation, des fonctions définissant les formes ainsi que toutes les fonctions définissant les primitives. En effet sur ce dernier point, comme on l'a expliqué dans le rapport de l'Incrément 3, nous évaluons dans les arguments des primitives dans les fonctions C qui définissent ces primitives. On a donc dû également modifier le pointeur de fonction permettant de faire appel à ces primitives afin qu'il prenne deux arguments au lieu d'un seul.

On renvoie le lecteur à `eval.c` et `primitive.c` et leur header associé, ainsi qu'à `object.h` et le rapport de l'Incrément 3.

## Structure de compound

Nous avons ensuite ajouté dans la structure object la structure de compound tel que proposé dans le sujet:

---

```
1      struct { struct object_t *parms;  
2          struct object_t *body;  
3          struct object_t *envt; } compound;
```

---

Dans object.c nous avons rajouté la création d'objet de type compound avec make\_compound():

---

```
1 object make_compound( object parms, object body, object envt ){  
2     object t = make_object(SFS_COMPOUND);  
3     t->this.compound.parms = parms;  
4     t->this.compound.body = body;  
5     t->this.compound.envt = envt;  
6     return t;}
```

---

## Forme lambda

Nous avons ensuite codé la forme lambda. Il s'agit simplement de récupérer les arguments de (lambda (parms) (body)) et de renvoyer un make\_compound(parms, body, env). On renvoie le lecteur à sfs\_lambda() dans eval.c.

Nous avons également modifié la fonction sfs\_print\_atom() dans print.c afin de d'afficher #<procedure> dans le cas ou l'évaluation retourne un objet de type compound.

## Evaluation d'une forme lambda

Tout d'abord nous avons créer une fonction C eval\_compound() permettant comme son nom l'indique d'évaluer un compound. Cette fonction sera appelée dans deux cas:

- Si le car d'une paire est une paire: C'est en fait pour le cas où l'on donne par exemple la s-expression ((lambda (x) (\*x 2)) 3). En effet le car de cette paire est une paire. On appelle alors eval\_compound() et on vérifie tout de même au début de cette fonction que l'évaluation du car en question est effectivement bien de type compound. Une paire ayant pour car une paire ne sera donc admis uniquement que si l'évaluation de ce car retourne un objet de type compound, dans les autres cas nous renvoyons une erreur. Ici se pose donc le problème suivant: la s-expression ((string->symbol "+") 1 2) est elle correcte ? Notre réponse est non, tout comme celle de l'interpréteur MIT/GNU Scheme qui nous a servis de référence et renvoie une erreur dans le cas présenté.
- Si le car d'une paire est un symbole associé à un compound: Dans ce cas on appelle également eval\_compound() qui se chargera d'évaluer ce compound avec les valeurs donnés.

Concernant le code de `eval_compound()`, on récupère `parms`, `body` et environnement d'exécution de la structure de `compound`, on ajoute un environnement vide à cet environnement d'exécution (qui est en fait une liste d'environnement, désolé pour l'abus de langage), on ajoute les bindings entre `parms` et valeurs dans ce nouvel environnement et on retourne l'évaluation du `body` dans l'environnement d'exécution. C'est la le coeur du mécanisme implémentant le  $\lambda$ -calcul dans notre interpréteur.

## Modification de `define`

A cette étape notre  $\lambda$ -calcul est opérationnel, et la forme `define` telle qu'elle a été codée dans l'increment 2 n'a en fait pas besoin d'être modifiée. Nous avons testé sans toucher à son code des choses basiques comme `(define mul-by-2 (lambda (x) (* x 2)))` et plus compliqué comme la définition de `count`, et cela marchait très bien. Il y avait juste un soucis pour la factorielle, qui était causé par l'appel à `sfs_eval()` dans la fonction `define`, ce que nous avons modifié.

Nous avons tout de meme rajouté quelques lignes afin de prendre en compte la forme simplifiée de déclaration de fonction "sans passer par `lambda`" (voir sujet page 19). Il s'agit en fait de transformer cette déclaration sans `lambda` en une déclaration avec `lambda` et de l'évaluer. Nous utiliserons la même méthode pour coder `let`.

## Forme `let`

`Let` permet de déclarer un variable locale. Pour coder `let` nous avons utilisé l'équivalence entre les deux s-expressions suivantes:

---

```
1 (let ((x valeur)) (body))
```

---

```
1 ((lambda (x) (body)) valeur )
```

---

La forme `let` se transforme donc en `lambda`, ce qui est déjà géré par notre interpréteur. On renvoie le lecteur à `eval.c` ligne 309 pour voir le code de `let`.

## Forme `map`

La forme `map` peut s'implémenter en dur en la codant en C, ce qui suppose au moins une trentaine de ligne. On remarque cependant que celle ci s'écrit en une seul instruction `scheme` donnée dans le sujet:

---

```
1 (define (map proc items)
2   (if (null? items)
3       ?()
4       (cons (proc (car items))(map proc (cdr items))))
5   )
6 )
```

---

Il suffit donc de charger cette instruction dès le lancement de notre interpréteur. On crée donc dans `init_interpreteur()` de `repl.c` une string contenant cette instruction, on la passe en argument de `sfs_read` pour créer l'objet `scheme` codant cette instruction et on appelle `sfs_eval` sur cet objet. On code ainsi `map` en 2 lignes.

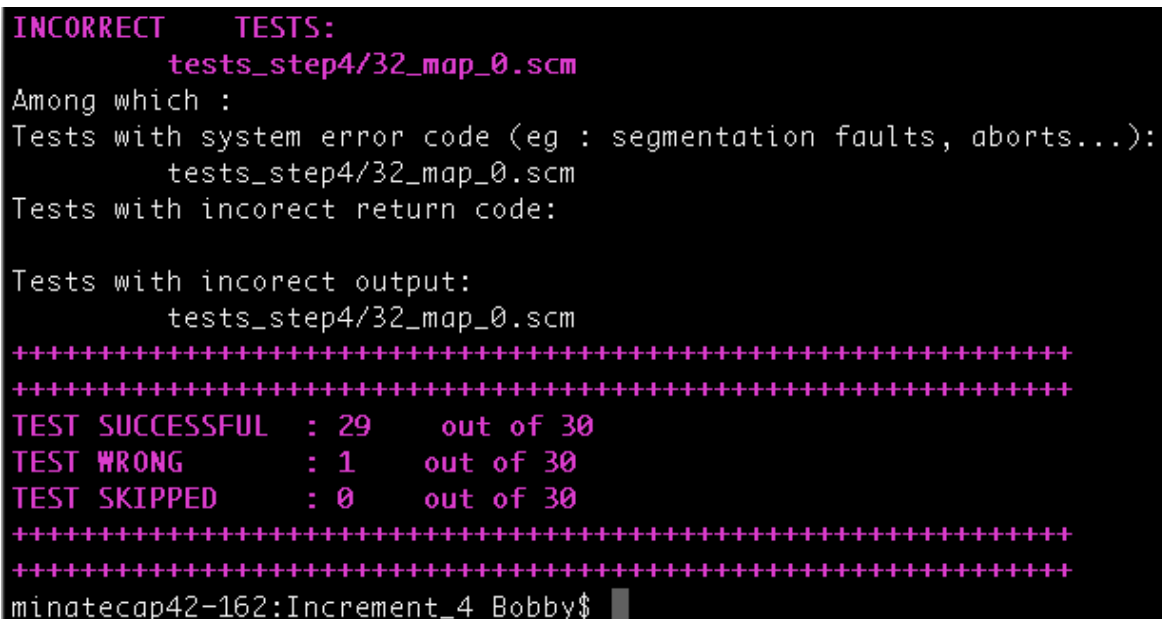
## Tests et vérification du code

Nous avons créé 30 fichiers de tests et résultats correspondants. La commande de test de l'incrément 4 est:

---

```
1 simpleUnitTest.sh -b -e ./scheme tests_step4/*.scm
```

---



```
INCORRECT   TESTS:
      tests_step4/32_map_0.scm
Among which :
Tests with system error code (eg : segmentation faults, aborts...):
      tests_step4/32_map_0.scm
Tests with incorrect return code:

Tests with incorrect output:
      tests_step4/32_map_0.scm
+++++
+++++
TEST SUCCESSFUL : 29 out of 30
TEST WRONG      : 1 out of 30
TEST SKIPPED    : 0 out of 30
+++++
+++++
minatecap42-162:Increment_4 Bobby$
```

Au début une bonne dizaine de tests ne passaient pas, ce qui nous à permis de corriger quelques bugs. Il en reste encore surement quelques uns, notamment de la gestion d'erreur mais globalement le code fonctionne correctement. `Map` ne fonctionne pas (problème avec `readline` ?) .

## **Conclusion**

Les objectifs du quatrième incrément ont été atteints. Cet incrément était vraiment le plus intéressant du projet. Honnêtement nous sommes très satisfaits de ce projet qui nous a permis de développer des compétences en C et de prendre beaucoup d'expérience et de culture générale en programmation. Merci aux professeurs pour leur implication et leur aide, c'était vraiment bien.