

# RAPPORT INCRÉMENT N°3

Vincent MEURISSE Thibault VINCENT, SICOM

23/11/2016

## Implémentation des primitives

Les primitives sont des objets scheme de type pointeur de fonction, fonction de type objet prenant en paramètre un objet (les arguments de la primitive). Nous avons donc ajouté un champ dans l'union de la structure des objets scheme dans object.h:

---

```
1 struct object_t *(*primitive)(struct object_t *);
```

---

Pour les représenter nous avons également défini le type Primitive:

---

```
1 typedef object (*Primitive)(object);
```

---

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

---

```
1 object make_primitive( Primitive p ){
2     object t = make_object(SFS_PRIMITIVE);
3     t->this.primitive = p;
4     return t;}
```

---

Il fallait ensuite modifier la fonction d'évaluation pour que dès qu'elle rencontre une paire, elle regarde d'abord si le car est une forme (cela avait été fait dans l'incrément 2), ou alors si c'est une primitive, sinon elle renvoie une erreur. On a donc créé une fonction eval\_primitive() dans le cas où le car de la paire à évaluer est une primitive.

---

```
1 object eval_primitive(object input){
2     object binding = search_list_env(car(input),LIST_ENV);
3     object o_prim = cadr(binding);
4     if (binding == NULL || o_prim->type != SFS_PRIMITIVE ){
5         WARNING_MSG("Unknown primitive");
6         return NULL;}
7     else{
8         return o_prim->this.primitive(cdr(input));} }
```

---

Cette fonction cherche dans les environnements si elle rencontre un symbole associé à une primitive, et dans ce cas elle appelle la primitive correspondante en lui donnant pour argument cdr(input). Nous avons fait une petite erreur à ce moment de l'incrément, en pensant que

nous ferions l'évaluation des arguments des primitives dans le code de chacune d'entre elles. Nous aurions dû directement évaluer les arguments ici lors de l'appel de la primitive en mettant `sfs_eval(cdr(input))` au lieu de `cdr(input)`. Cela aurait permis de "factoriser" l'évaluation des arguments, qui se fera donc dans chaque primitive dans notre cas. Il était trop tard lorsque nous nous sommes rendu compte de ce léger défaut, mais le fonctionnement du code reste strictement équivalent (pour l'instant).

## Ecriture des primitives

Nous avons écrits les primitives demandées dans le sujet, dans le fichier `primitive.h`. Nous chargeons ces primitives dans un environnement 0 (top level) avec `init_primitive()` qui pour chaque primitive s'occupe de créer le binding de la primitive et de son symbole associé. On ajoute ensuite le binding à l'environnement. On appelle ensuite `init_primitive()` dans `init_interpreter()` de `repl.c` et on ajoute ensuite un environnement vide pour l'utilisateur.

## Prédicats

Le principe des prédicats est toujours le même: il s'agit de tester si tous les objets donnés en argument sont du type donné. On renvoie vrai si c'est le cas, et si au moins un argument n'est pas du type testé on renvoie faux. C'est donc en principe le code d'un ET logique.

## Conversion de type

Les fonctions de conversion de type permettent de renvoyer un objet `scheme` qui est celui donné en paramètre dans un autre type. Pour les conversions `char->int` et `int->char` nous avons utilisé le code ASCII, le principe du code de ces deux fonctions repose en fait sur la conversion de type en C. De plus il fallait créer un objet `scheme` du nouveau type, mettre ce que contenait l'objet en paramètre dedans et le renvoyer.

Nous avons juste eu un petit problème non résolu pour la conversion `string->number` qui vient en fait de la fonction `strtol()` qui ne permet pas de différencier le cas où la chaîne de caractère est "0" ou "000000" ou "0.000" avec le cas où la chaîne ne contient pas de nombre.

## Arithmétique

Nous avons créé les primitives addition (+), soustraction (-) et multiplication (\*) en nous servant de boucles `while`. En indiquant une condition d'arrêt de notre boucle (`cdr` de type `SFS_NIL`), on ajoute le car de notre objet et on passe à l'argument suivant en passant au `cdr` de l'objet.

Il y a eu une difficulté en plus dans la primitive soustraction car le premier élément devait être traité différemment (non soustrait d'entrée). En effet, la syntaxe est de la forme : `(- a b c d)` renvoie `a-b-c-d` et non `-a-b-c-d` comme `(+ a b c d)` renvoie `+a+b+c+d`.

Nous avons créé une primitive division (/) qui permet d'effectuer des divisions avec flottants et de donner des résultats flottants de divisions classiques. Elle traite aussi son premier terme différemment. Elle a la syntaxe suivante : `(/ a b c d etc ?)` divise `a` par `b*c*d*etc..`

Pour ce qui est des primitives quotient, reste, =, < et >, elles prennent toutes uniquement 2 termes en paramètres. Pas besoin donc d'utiliser une boucle while. On utilise les fonctions équivalentes en langage C pour obtenir le résultat souhaité.

Dans tous les cas précédents, la conclusion de la fonction réside dans un make\_<type>, ce qui peut parfois faire apparaître plusieurs boucles if en fonctions des différents cas de figure rencontrés.

## Manipulation de liste

La principale difficulté rencontrée dans l'écriture des primitives de gestion de liste était de gérer correctement le placement de l'évaluation des arguments. On utilise ensuite les fonctions de gestion de liste définies dans list.c. La fonction list() utilise un code récursif pour pouvoir traiter l'éventuelle infinité de ses paramètres (list est en fait un cons récursif). On aurait également pu faire list() avec un while.

Pour set-car! et set-cdr!, il fallait changer simplement la valeur à l'adresse de l'ancien car de la liste donné en paramètre par celle du nouveau car donné en paramètre.

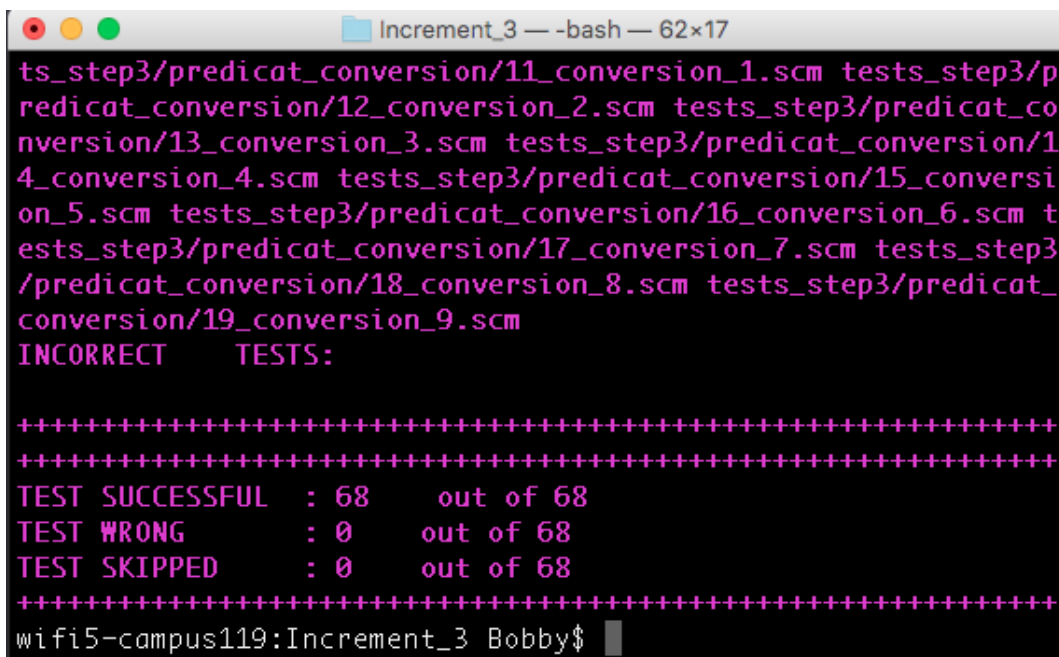
## Tests et vérification du code

Nous avons créé 68 fichiers de tests et résultats correspondants, divisés en 2 dossiers: predi-cats\_conversion et arithmétique\_liste. La commande de test est:

---

```
1 simpleUnitTest.sh -b -e ./scheme tests_step3/*/*.scm
```

---



```
ts_step3/predicat_conversion/11_conversion_1.scm tests_step3/p
redicat_conversion/12_conversion_2.scm tests_step3/predicat_co
nversion/13_conversion_3.scm tests_step3/predicat_conversion/1
4_conversion_4.scm tests_step3/predicat_conversion/15_conversi
on_5.scm tests_step3/predicat_conversion/16_conversion_6.scm t
ests_step3/predicat_conversion/17_conversion_7.scm tests_step3
/predicat_conversion/18_conversion_8.scm tests_step3/predicat_
conversion/19_conversion_9.scm
INCORRECT TESTS:

++++
++++
TEST SUCCESSFUL : 68 out of 68
TEST WRONG : 0 out of 68
TEST SKIPPED : 0 out of 68
++++
wifi5-campus119:Increment_3 Bobby$
```

Au début une bonne dizaine de tests ne passaient pas, ce qui nous a permis de corriger quelques bugs. Il en reste encore sûrement quelques uns, mais globalement le code fonctionne correctement.

## Conclusion

Les objectifs du troisième incrément ont été atteints. Les erreurs et bugs étaient nombreux car il y avait beaucoup de cas particuliers à traiter. Cet incrément posait également la question de normalisation du Scheme, en effet il est possible de coder les primitives de plein de façons différentes, et ce qu'elles doivent faire n'est pas forcément précisé parfois, on a cependant essayé de coller le plus possible au R5RS.