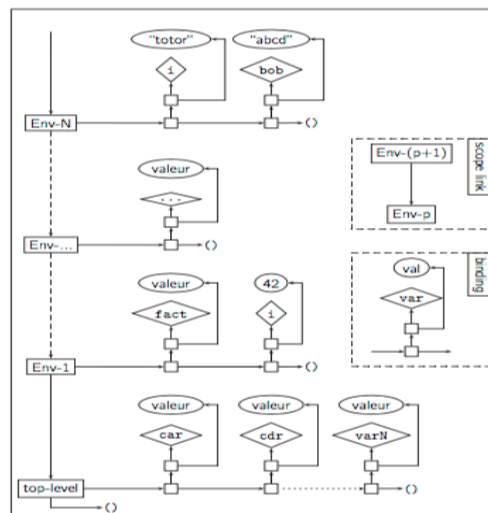


RAPPORT INCRÉMENT N°2

Implémentation des environnements

Les environnements en Scheme sont pour nous la liste des paires (symboles valeur), paires que l'on appellera "binding" par la suite. Nous avons créé une Liste d'environnements sous forme de variable globale qui contient les environnements scheme selon le schéma suivant:



Pour représenter les environnements, nous avons utilisé les listes Scheme. Une liste d'environnements est également une liste scheme qui contient les environnements. Le scope link dans notre cas est que l'environnement le plus récent est à gauche de la liste d'environnement (en tête). Pour plus de détail, le lecteur est renvoyé aux fichiers environnement.h , environnement.c et à init_interpreter() au début de repl.c

Nous avons créé des fonctions de gestion des environnements. Créer un environnement, c'est créer une liste vide: d'où creat_env() retourne nil. Créer un binding c'est créer une paire (symbole valeur). On ajoute un binding à un environnement avec add_env() qui est en fait un simple cons (voir list.c).

Ensuite nous avons créé des fonctions associées à des symboles dans la fonction d'évaluation permettant à l'utilisateur de gérer les environnements. Nous ne savons pas si c'est autorisé mais il faut bien que l'utilisateur puisse gérer les environnements avec son interpréteur Scheme, de toute façon celles ci nous sont utiles pour développer l'interpréteur:

- « show_list » montre la liste d'environnement
- « add_env » permet d'ajouter un environnement dans la liste et d'en faire l'environnement courant. Il s'agit juste d'un cons de create_env() avec la liste d'environnement globale.
- « rm_env » permet de supprimer l'environnement courant et de faire de l'environnement précédent l'environnement courant. on remplace juste la liste d'environnement par son cdr.

La fonction la plus intéressante et utile pour les fonctions d'évaluation est `search_list_env()` (et son petit frere `search_env()`). Elle prend en paramètre un symbole (uniquement, sinon message d'erreur) et retourne le binding correspondant à la premiere occurence de ce symbole dans la liste d'environnement. Elle renvoie NULL si elle ne trouve rien. On code cela avec une simple récursivité sur le `cdr` de l'environnement. Le code est ci dessous:

Listing 1: `search_env()`

```
1
2 object search_env(object symbol, object env){
3     if (env->type != SFS_NIL){
4         if (strcmp(symbol->this.symbol,car(car(env))->this.symbol) == 0)
5             return car(env);
6         else
7             return search_env(symbol,cdr(env));}
8     else
9         return NULL;
10 }
11
12 object search_list_env(object symbol, object list_env){
13     if (list_env->type != SFS_NIL)
14     {
15         if (search_env(symbol, car(list_env)) != NULL)
16             return search_env(symbol, car(list_env));
17         else
18             return search_list_env(symbol,cdr(list_env));}
19     else
20         return NULL;
21 }
```

Le lecteur est invité à tester les environnements depuis l'interpréteur grace aux commandes `show_list` , `add_env` et `rm_env`. Il pourra par exemple créer des variables (avec `define`), ajouter un environnement, définir de nouvelle variables puis changer les valeurs de certaines d'entre elles (avec `set!`) et vérifier à chaque fois l'état de la liste d'environnement. Sinon des fichiers de tests ont été crée spécialement pour cela.

Le fait d'avoir implémenter cela avec des listes `scheme` permet l'affichage à l'écran de la liste d'environnement, ce qui est plutôt pratique.

Fonction d'évaluation

Dans cet incrément, l'objectif était de créer la fonction d'évaluation de l'interpréteur Scheme, qui constitue la deuxième étape de la boucle REPL (read, eval, print). Les étapes read et print sont déjà fonctionnelles. Pour eval nous avons pour l'instant implémenter les formes Scheme de base telles que quote, define, set!, if, and et or, ce qui était l'objectif.

sfs_eval() :

Le coeur de la récursivité de l'évaluation réside dans la fonction sfs_eval. Cette fonction prend en entrée un objet (objet input) et la traite de la manière suivante :

- Si le type de l'objet est une paire, elle renvoie sur eval_pair(), qui nous permet d'évaluer les formes (pour l'instant),
- Si le type est un symbole, elle renvoie sur eval_symbol() (qui nous permet de renvoyer la valeur associée à une variable, d'utiliser des fonctions utiles pour gérer les environnements).
- Dans les autres cas, elle auto-évalue l'objet, c'est-à-dire qu'elle le retourne tel quel.

eval_symbol() :

Cette fonction utilise la fonction search_env() pour renvoyer la valeur associée à la variable trouvée dans l'environnement courant, ou une erreur si elle n'est pas définie.

eval_pair() :

Afin d'évaluer les formes désirées, la fonction compare le car de la paire aux chaînes de caractères quote, define, set!, if, and et or et renvoie sur la fonction correspondante. Dans les autres cas, elle auto-évalue la paire.

- **sfs_quote() :**

Cette fonction renvoie tout simplement le cdr de l'input. Nous avons modifié le fichier read.c dans la fonction sfs_read() pour prendre en compte le raccourci de la forme quote '.

- **sfs_define() :**

La fonction renvoie tout d'abord un message d'erreur si la syntaxe n'est pas la suivante : (define variable valeur). On cherche ensuite dans l'environnement courant si la variable existe avec search_env(). On renvoie un message d'erreur si elle existe déjà ou si le nom de la variable est un nom réservé aux formes (grâce à une fonction is_reserved qui liste le nom des formes). Ensuite, la fonction crée la paire et l'ajoute à l'environnement courant.

- **sfs_set() :**

De la même manière qu'avec define, la fonction renvoie tout d'abord un message d'erreur si la syntaxe n'est pas la suivante : (set! variable valeur). Elle cherche ensuite dans la liste d'environnement si la variable existe déjà et s'arrête à la première occurrence de la variable. Si elle n'existe pas, la fonction renvoie un message d'erreur. Ensuite, elle enregistre l'ancienne valeur, elle la modifie ensuite dans l'environnement (avec respect du principe de masquage) puis renvoie l'ancienne valeur.

- `sfs_and()` et `sfs_or()` :

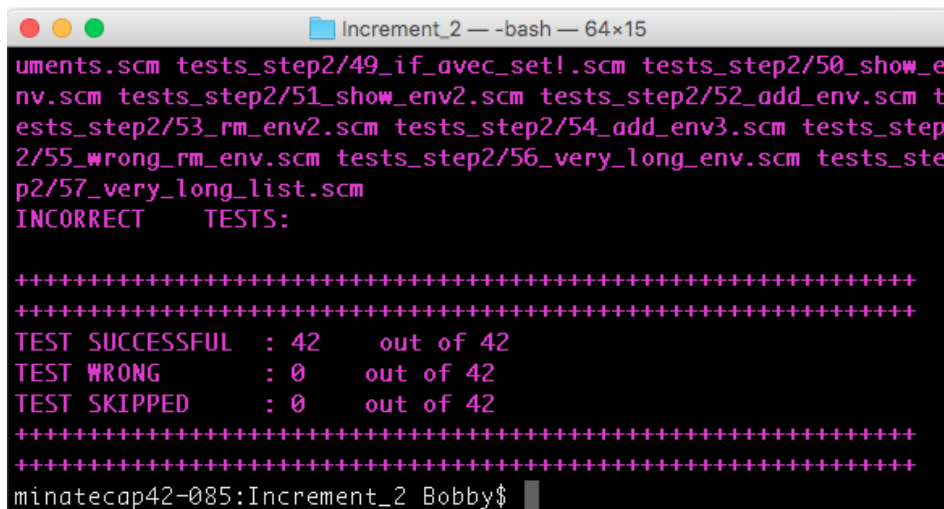
Ces deux fonctions fonctionnent de manière récursive (car `and` et `or` peuvent éventuellement avoir plus de deux arguments). Elles analysent le `car` du `cdr` de l'`input`. Pour `and`, si celui-ci est faux, `sfs_and` retourne `f`, pour `or`, si celui-ci est vrai, `sfs_or` retourne `t`, dans les autres cas, elles continuent l'évaluation avec le terme suivant. La condition d'arrêt est si le `cdr` de l'`input` est un `nil` (fin de la paire).

- `sfs_if()` :

Dans un premier temps, la fonction renvoie un message d'erreur si la syntaxe n'est pas la suivante : (if prédicat conséquence alternative). Ensuite, elle évalue la conséquence (`caddr` de l'`input`) si le prédicat (`cadr` de l'`input`) est vrai ou l'alternative (`caddr` de l'`input`) si le prédicat est faux. Nous n'avons pas implanter le `if` avec `goto` car nous n'avons pas compris le sujet sur ce point, même si on comprend que la multitude d'appel à `if` n'est pas très optimisée. Nous attendons des explications d'un professeur lors de la séance de mercredi.

Tests et vérification du code

Nous avons créé 42 fichiers de tests et résultats correspondants. Nous pensons avoir fait le tour des principaux tests, en rajouter serait redondant, sauf si on a raté quelque chose d'important. Une bonne dizaine de tests ne passaient pas au début, cela nous a permis de corriger quelques erreurs dans le code. Nous avons également eu certains tests où c'était en fait le fichier de résultats qui n'était pas bon... (l'erreur est humaine). En tout cas tous nos tests passent désormais, ce qui ne prouve pas mais nous indique quand même que notre code fonctionne globalement bien.



```

Increment_2 — -bash — 64x15
uments.scm tests_step2/49_if_avec_set!.scm tests_step2/50_show_e
nv.scm tests_step2/51_show_env2.scm tests_step2/52_add_env.scm t
ests_step2/53_rm_env2.scm tests_step2/54_add_env3.scm tests_step
2/55_wrong_rm_env.scm tests_step2/56_very_long_env.scm tests_ste
p2/57_very_long_list.scm
INCORRECT TESTS:

+++++
+++++
TEST SUCCESSFUL : 42 out of 42
TEST WRONG : 0 out of 42
TEST SKIPPED : 0 out of 42
+++++
+++++
minatecap42-085:Increment_2 Bobby$

```

Conclusion

Les objectifs du deuxième incrément ont été atteints. Les erreurs et bugs étaient bien moindres car il y avait beaucoup moins de cas particuliers à traiter que dans l'incrément 1. Nous avons également pris en compte certaines remarques faites sur le code de l'incrément 1 (dans `read.c` surtout, quelques petites modifications).