Rapport Incrément №1

Vincent MEURISSE Thibault VINCENT, SICOM 28/09/2016

Représentation des objets Scheme

Pour représenter les objets Scheme, nous avons utilisé la structure de données suivante:

Listing 1: Structure de données

```
typedef struct object_t {
    uint type;
    union {
        num             number;
        int             integer;
        int             boolean_value;
        char            character;
        string          string;
        string          symbol;
        struct pair_t {
            struct object_t *car;
            struct object_t *cdr;
        }               pair;
        struct object_t *special;
    } this
} *object;
```

Pour créer un objet Scheme il suffit ensuite d'appeler les fonctions make associées au type d'objet que l'on veut créer. Ces fonctions utilise la fonction make_object qui alloue la mémoire nécessaire à la création de l'objet et renseigne le type de l'objet scheme. Elles remplissent ensuite le champ correspondant à ce type (string, integer character, symbol...) qui est donné en paramètre de la fonction. Voila à quoi ressemble la fonction make_integer par exemple.

Listing 2: Fonction de création d'un atome entier

```
object make_integer( int val ){
    object t = make_object(SFS_INTEGER);
    t->this.integer = val;
    return t;
}
```

Les fonctions make_boolean et make_nil sont appelées une seule fois au lancement de l'interpréteur et sont déclarées globalement dans tout le programme.

Parsing, fonctions read

Il nous fallait maintenant coder la fonction de lecture et de création en mémoire d'une S-expression. Cette fonction à pour but de construire à partir de l'entrée utilisateur (une chaine de caractère) l'arbre syntaxique qui représente une S-expression, et de vérifier que cette S-expression est bien valide en Scheme (gestion d'erreur). On appelle donc la fonction read() qui fonctionne en récursivité croisée avec une autre fonction read_pair().

Si la fonction lit une parenthèse ouverte, elle rentre dans read_pair (sauf si c'est un NIL). read_pair créée alors un objet. Le car de cet objet est renvoyé dans read. On insère alors une parenthèse après le car (qui sera alors pris en compte dans le cdr) et on envoie à son tour le cdr dans read. L'insertion de parenthèse dans la chaine à la fin de la lecture du car est une astuce qui nous permet de lire facilement le NIL qui forme le cdr du dernier élément d'une paire, et de former correctement les s-expressions en mémoire.

Si la fonction ne lit pas de parenthèse, elle rentre dans read_atom qui retournera un atome, quel que soit son type.

read_atom est quant à elle constituée d'un switch qui relate tout les cas d'atomes différents et retourne l'atome correspondant ou des messages d'erreurs lorsque la syntaxe d'un atome n'est pas valide.

Pour plus de détail, on renvoie le lecteur au fichier read.c.

Affichage de l'arbre syntaxique

Dans le fichier print.c on trouve les fonctions permettant d'afficher(en respectant la syntaxe et les conventions du Scheme) l'arbre syntaxique représentant l'objet donné en paramètre.

La fonction print marche avec le même principe que la fonction read : Elle étudie tout d'abord le type de l'objet qu'elle reçoit en entrée. Si ce type est une paire, elle renvoie print_pair, sinon, elle renvoie print_atome. print_pair marche de la manière suivante :

Si le car est une paire et le cdr est une paire, ou si le car est une paire et le cdr est un NIL : Dans le premier cas on imprime une parenthèse ouverte, on envoie le car sur print, on imprime un espace, et on envoie le cdr sur print. Dans le deuxième cas, on imprime une parenthèse ouverte, on envoie le car sur print puis on imprime une parenthèse fermée.

Sinon (cas où il y a de simples atomes dans le car ou le cdr) : Si le cdr n'est pas de type pair (implicitement : si on se trouve à la fin d'une branche de l'arbre), on envoie le car sur print puis on imprime une parenthèse fermée. Sinon, nous ne sommes pas à la fin de la branche dans laquelle on se trouve, on envoie donc le car sur print, on imprime un espace, puis on envoie le cdr sur print.

Tout cela permet d'afficher les paires avec la convention scheme qui était indiquée dans le sujet. Pour plus de détail, on renvoie le lecteur au fichier print.c.

Tests et vérification du code

Tests fournis

Nous avons d'abord utilisé les test fournis (evolved et simple). Nous avons modifié 2 fichiers de résultats (.res) qui ne comportaient pas de "==>" au début (les tests 92 et 93 de evolved). La quasi-totalité des tests fonctionne correctement. Deux tests échouent.

Le premier test qui échoue concerne deux tests contradictoires. L'un laisse le backslash dans le cas de la présence de \" dans une chaine de caractère, l'autre le supprime (tests 50 de simple et 30 de evolved). Il y en a donc forcément un des deux qui échoue. Face à l'absence de contraintes Scheme à ce sujet, nous avons choisi de laisser le backslash.

Le deuxième test qui échoue concerne la considération des entiers comme infini par scheme lorsqu'ils dépassent une certaine valeur (test 20 de evolved). Nous n'avons pas encore implémenter cette fonctionnalité dans notre code.

```
INCORRECT    TESTS:
         tests_step1/evolved/20_infinite_int.scm tests_step1/simple/50_simpletestglobal.scm
Among which :
Tests with system error code (eg : segmentation faults, aborts...):

Tests with incorect return code:

Tests with incorect output:
         tests_step1/evolved/20_infinite_int.scm tests_step1/simple/50_simpletestglobal.scm
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
TEST SUCCESSFUL  : 37    out of 39
TEST WRONG       : 2    out of 39
TEST SKIPPED     : 0    out of 39
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
thibaults-macbook-pro:Increment_1 Bobby$
```

Tests crées

Nous avons ensuite décidé de faire quelques tests en plus car il restait encore des cas que nous tenions à vérifier (notemment les strings collés à des symboles et des entiers dans les paires). De plus nous gérons les nombre réels (FLOAT) et aucun test n'a été fait pour ce type d'atome. Nous avons bien fait car sur les 5 tests crées, 3 ne passaient pas. On peut trouver ces tests dans tests_step1/very_evolved. Nous avons corrigé le code pour faire fonctionner ces tests.

Conclusion

Notre programme fonctionne comme voulu. Les objectifs du premier incrément ont été atteint. L'automatisation des tests a été d'une grande utilité pour développer ce programme. Peut être qu'il reste encore des bugs mais nous ne pouvons pas tout tester c'est impossible (il faudrait sortir une béta pour obtenir des retours d'utilisateurs).