Rolland Léa
David Leopold
SICOM

Rapport 4e Livrable

1. Modification du code livrable 3

1.1. Mise au propre

1.1.1. Fonctions gets

Le code étant saturé d'appels de type ->this.pair.cdr->this.pair. etc., nous avons créé une fonction gets : `object get (string cible, object input).`
Elle prend en argument la cible : "car", "cdr", "cadr", ou même "cadddr" et l'objet dont on veut l'élément. Elle retourne ensuite l'objet trouvé.
Elle ne comporte pas de vérification du type "la cible est NULL" si le cdr demandé est vide par exemple. Ces vérifications sont généralement faites dans la fonction qui l'a appelée.
Cela a permis de clarifier le code, dans la gestion de la forme if ou de lambda par exemple.

1.1.2. Fichiers prim

Les fonctions de gestions des primitives prenez plus de place de prévu dans le fichier eval.c et rendaient difficile la navigation entre les différentes fonctions. Nous avons donc créé un fichier prim.c ainsi qu'un prim.h où on a relocaliser toutes les fonctions prim du livrable 3, ainsi que la fonction `void creer_primitives (void)` appelée en initialisation du programme. Cela nous a permis de libérer eval.c de plus de 500 lignes de code.

1.2. Objets de type SFS_PROBLEM

Jusqu'à ce livrable, quand on avait une erreur dans la commande entrée par l'utilisateur, on renvoyait NULL avec un WARNING_MSG . Cela valait pour un problème de type, une variable non définie, un argument manquant ou en trop dans les primitives, etc. Si sfs_print recevait NULL, il n'affichait rien et rendait la main à l'utilisateur. Cela fonctionnait plutôt bien, jusqu'à ce qu'on utilise notre code avec simpleUnitTest.sh, en mode script donc. Si on retournait NULL au main de repl.c alors qu'on était en mode script, le programme s'arrêtait immédiatement et donnait un message d'erreur. Pour résoudre ce problème en affichant toujours des messages et en rendant la main à l'utilisateur, nous avons créé un nouveau type d'objet.
Nous l'avons ajouté à la structure object de type SFS_PROBLEM (=11) dans object.h avec `string warning;` dans l'union.
On crée ce type d'objet avec la fonction `object make_object( uint type )` dans object.c, qui crée un objet de type SFS_PROBLEM et copie dans sa chaîne warning le message qu'on voudra afficher. Tous ces objets sont initialisés dans la fonction `void init_interpreter ( void )` de repl.c. Voici les warning que nous créons (et nous en rajoutons tous les jours) :

object	message	utilisation
arg_plus	“il y a trop d'arguments”	toutes les primitives de prenant qu’un argument de type >, <, conversion ou vérification des types
arg_moins	"il n'y a pas assez d'arguments"	idem
pb_type	"l'argument n'est pas du bon type"	primitives de calcul (on veut un nombre) et de conversion (string->symbol demande un type string)
var_non_def	"la variable n'est pas définie"	les fonctions valeur_symbol, sfs_eval pour un symbol non défini, set!_car et set!_cdr...
cdr_pb	"il n'y a pas de cdr"	dans la primitive cdr, quand il n’y a pas de cdr à renvoyer par exemple
input_vide	"input == NULL"	au début d’eval

Ces objets sont ensuites définis comme variables externes dans object.h. Si on les renvoie à print_atom, celle ci affiche le message qui est stocké dans l’objet et fait un return simple qui rend la main à l’utilisateur.

Nous avons dû modifier tous nos tests qui jusqu’à présent renvoyaient NULL en cas de problème pour les rendre conformes.

1.3. La forme Define

Nous avons dû modifier notre forme define pour qu’elle ne renvoie plus rien, au lieu du symbol qu’on venait de définir (idem pour set!). Nous avons donc enlevé les return object et modifié les fonctions define et set! pour qu’elles aient un type de retour NULL.

Nous avons aussi créé un objet return_define, car comme dit plus haut, juste retourner NULL peut poser des problèmes. C’est un objet de type SFS_PROBLEM qui n’affiche aucun message (ou “”) et rend la main. Le seul problème et qu’il produit un retour à la ligne dont nous n’avons pas trouvé la source, et nous avons donc dû modifier nos test pour enlever les retours de type symbol et rajouter ce retour à la ligne.

1.4. Les environnements en argument

Pour pouvoir étudier les formes lambda et begin, il nous fallait pouvoir évaluer des prédicats dans des environnements en particulier. Il a donc fallu ajouter à la fonction sfs_eval la gestion des environnements : on a donc ajouté dans ses arguments l’environnement où elle devait se placer pour évaluer (le top-level ou un autre environnement chaîné au top-level généralement).

Une fois cela fait, il a fallu aussi l’implémenter à toutes les fonctions qui considéraient par défaut que l’environnement où se placer était le top-level.

```
object sfs_eval( object input, object envt);
void set(object variable, object valeur, object envt);
void define (object variable, object valeur, object envt);
```

```
uint chercher_symb (string nom, object envt);
object valeur_symb (string nom, object envt);
object ajout_env (object env_present);
uint ajout_binding (object variable, object valeur, object envt);
object evaluer_arg (object liste, object envt);
```

La fonction ajout_env permet d'ajouter en 3 lignes un nouvel environnement en tête de celui qu'on lui donne en argument.

2. Formes lambda et begin

2.1. Compound

Dans un premier temps on inclut l'objet de type SFS_COMPOUND (=10) dans la structure object de object.h :

```
struct compound_t
{
    struct object_t* parms ;
    struct object_t* body ;
    struct object_t* envt ;
}compound;
```

Nous lui ajoutons une fonction `object make_compound (void)` dans object.c pour initialiser un agrégat (avec les paramètres, le body et envt à NULL);

2.2. Formes

2.2.1. Lambda seul

Nous avons identifié deux façons d'avoir un lambda : en reconnaissant la forme, avec la fonction is_form dans sfs_eval, et en évaluant un paramètre qui est en fait le nom associé à une fonction (quand on a fait au préalable (define f (lambda…))).
Nous les gérons à deux endroits différents, alors que nous pensons qu'il aurait été possible d'alléger le code en traitant les deux cas de la même manière (peut être avec un appel de fonction faisant l'évaluation du compound, ou bien un go to quand on rencontre un cas pour se placer au niveau de l'évaluation de l'autre cas ?). Par manque de temps nous l'avons laissé ainsi.

Quand on rencontre une forme lambda, on va créer un agrégat où nous stockerons les données : à l'aide de la fonction `object evaluer_parms (object liste);` qui est semblable à la fonction evaluer_arg pour les primitives, on remplit le .body et le .parms du compound.
evaluer_arg ne fait que créer une liste stockant les différents atomes, elle n'évalue pas.
On crée un environnement chaîné au top-level qu'on met dans le .env, et ça sera l'environnement dans lequel on effectuera les évaluations.

Si le lambda est seul (s'il n'y a pas de valeur d'argument donné après), on retourne l'agrégat, et print.c retournera #<procedure>.

Sinon, dans l'environnement du prédicat, on utilise define pour donner aux arguments les valeurs écrites après le lambda.
Une fois tous les arguments attribués, on évalue le body contenant les arguments modifiés, toujours dans l'argument du prédicat. On retourne le "dernier return" (implémenté pour retourner total dans l'exemple count).

2.2.2. Prédicat de type compound

Dans sfs_eval, quand on trouve un symbol qui a été défini comme un compound, on va faire globalement le même traitement que plus haut en se plaçant dans l'environnement de son env. On définit les arguments pointés par son arg comme ceux donnés après lui dans l'expression (sauf s'il n'y en a pas), et on évalue le body.

2.2.3. Begin

On remplit une liste d'arguments avec evaluer_arg (qui évalue les paramètres) et on retourne le résultat du tout dernier.

2.2.4. Procedure?

On a rajouté une primitive pour évaluer si oui ou non le prédicat et de type compound ou non.

2.2.5. Let

Implémentation de la forme let dans sfs_eval : On crée une nouvelle pair dans le car de laquelle on met le symbol lambda. Dans son cdr on met une pair contenant les symboles qui décrivent les arguments, et on stocke dans un autre objet les valeurs que vont prendre ces objets. Puis dans son cddr on met le body.
On crée une nouvelle pair dont le car sera celle qu'on vient de créer comportant le lambda, et son cdr les valeurs des arguments :
(let ((x 2) (y 3)) (* x y)) devient ((lambda (x y) (* x y)) 2 3).

3. Tests

Dans le dossier tests_step4, on a un fichier pour tester lambda seul, un autre pour tester des fonctions définies avec des lambdas, un autre pour tester begin, et un dernier testant l'exemple count du sujet.
On en rajoute un testant rapidement let.

4. A faire en plus

Nous n'avons pas eu le temps de faire la forme eval du livrable précédent. Elle était la prochaine étape du code, mais nous avons manqué de temps (une ébauche dois d'ailleurs trainer dans le programme).
Des tests séparés en simple et en evolved seraient utilent, pour l'instant on n'a que des tests très simples testant si les formes/primitives marchent. Nous n'avons pas eu le

temps de construire un programme plus élaboré avec un algorithme fait en scheme par exemple.