Rolland Léa
David Leopold
SICOM

Rapport 3e Livrable

1. Initialisation des primitives

On crée d'abord un nouveau type d'objet, qu'on appellera prim, et qui est un pointeur de fonctions :

```
struct primitive_t
{
    struct object_t *(*fonction)(struct object_t *);
} prim;
```

Puis, dans la fonction `void creer_primitives(void)` , on implémente toutes les primitives voulues en les définissant comme des variables de type primitives, dont le nom est celui du symbole appelé (par exemple '+') et la valeur le nom de la fonction pointée ('plus_p'). La forme define nous permet de faire tout cela en une ligne pour chaque primitive :

```
define(make_symbol("+"),make_primitive(plus_p));
define(make_symbol("-"),make_primitive(moins_p));
```

etc.

Cette fonction creer_primitives est appelée au tout début du programme, dans repl.c, en même temps que la création de nil, boolean_true, etc.

2. Utilisation dans eval

2.1. Appel

Dans la fonction sfs_eval, dans le cas où l'objet évalué est une paire, et que son car n'est pas une forme, on va créer un objet p égal à la valeur du car. La fonction `object valeur_symb (string nom)` renvoie la valeur associée à un symbol si celui-ci a été défini dans les environnements (et renvoie NULL sinon). Dans le cas d'une primitive, si le car est '+', valeur_symb va renvoyer l'objet plus_p de type primitive.

Si on a une primitive (c'est à dire p != NULL), on va évaluer les arguments de la primitive, c'est à dire le cadr et le caddr de la pair évaluée. Pour cela, on utilise la fonction `object evaluer_arg (object liste)` qui va renvoyer une liste (paire) contenant les objets évalués à utiliser par la primitive.

Une fois qu'on a la primitive et les arguments, on renvoie `(p->this.prim.fonction)(liste_arg)` , c'est à dire le résultat de la fonction prim appliquée aux arguments.

2.2. Primitives implémentées

2.2.1. Arithmétiques

'Symbol' - Primitive	Commentaire
'+' - plus_p	
'-' - moins_p	
'*' - mult_p	
'quotient' - quotient_p	calcule la division successive de tous les arguments à l'aide d'un double puis renvoie son arrondi (supérieur pour 0,5 et plus, inférieur sinon)
'remainder' - remainder_p	renvoie le reste de la division euclidienne des arguments. Renvoie NULL s'il y a plus de deux arguments
'<' - inf_p	Renvoie #t si arg1 < arg2, #f sinon. Renvoie NULL s'il y a plus de deux arguments
'>' - sup_p	idem
'=' - egal_p	idem

2.2.2. Type

- 'boolean?' - boolean_p
- 'symbol?' - symbol_p
- 'integer?' - integer_p
- 'char? - char_p
- 'string?- string_p
- 'pair?' - pair_p
- 'nil?' - nil_p
- 'null?' - null_p

Ces primitives renvoient #t si le type de l'argument est le bon, #f sinon. Elles renvoient null s'il y a plus d'un argument.

2.2.3. Conversion

'Symbol' - Primitive	Commentaires
'char->integer' - char_integer_p	si on lui donne un caractère tel que #\3, il ne renvoie pas 3 mais sa valeur en décimal avec ascii, soit 51.
'integer->char' - integer_char_p	
'number->string' - number_string_p	
'string->number' - string_number_p	renvoie 0 si ce qui est dans la chaine n'est pas un nombre
'symbol->string' - symbol_string_p	
'string->symbol' - string_symbol_p	considère que les guillemets entourant la chaîne font partie du nouveau symbol (à corriger (?))

Ces primitives créent de nouveaux objets du type voulu avec l'argument donné, et les retourne. Si l'argument n'est pas du bon type, ou s'il y a plus d'un argument, la primitive renvoie NULL.

2.2.4. Manipulation de liste

'Symbol' - Primitive	Commentaire
'cons' - cons_p	Crée une pair contenant les arguments à l'aide d'ajout_queue.
'car' - car_p	Renvoie le car des arguments
'cdr' - cdr_p	Renvoie une pair contenant tous les arguments sauf le car. Erreur à résoudre : (cdr (cons 1 2)) renvoie (2) et pas 2
'list'	renvoie une liste d'argument. Erreur d'espaces quand utilisé avec cons
'set-car!'	change le car d'une variable définie et la renvoie. Renvoie NULL s'il manque un argument ou que la variable n'est pas définie
'set-cdr!'	change le cdr d'une variable définie est la renvoie.

/!\ set-car! et set-cdr! ne renvoie pas le nom de sa variable mais sa valeur :

```
> (define x (list 1 2))
==> x
> (set-car! x 2)
==> (2 2)
```

au lieu de x : à corriger.

2.2.5. Égalité polymorphique

Pas encore implémenté.

2.2.6. En plus

Modification de sfs_print_pair pour faire afficher au programme (object . object) quand il reçoit une pair dont le cdr n'est pas une pair, ni un nil/NULL.

3. Tests

On a un nouveau fichier tests_step3 contenant 27 tests pour chaque primitive. On les teste pour un bon fonctionnement, pour une erreur, et si possible avec des formes.

Ils sont tous validés pour les primitives existantes.

4. Problèmes rencontrés

Une fois le principe des primitives compris, il a été assez simple et rapide d'en implémenter la plupart. Mais un problème s'est manifesté provenant du read.c : dans la fonction `object input_to_symbol (char*input, uint*here)` qui convertit la chaîne en symbol avant de renvoyer un objet de type symbol, la gestion en mémoire de la chaîne prévue pour stocker le symbol était mauvaise. Dans un premier temps, on utilisait une simple chaîne de type string ou on copiait *input jusqu'à un certain indice avec strncpy. Lors de la première utilisation, cela marchait bien. Mais si on avait une expression longue telle que (car (cons 1 (cons 2 (cons 3 ()))))), les cons étaient mal lus (souvent "cons?"). J'ai tenté de résoudre le problème en utilisant un pointeur sur chaîne alloué et libéré à la fin de la fonction, mais étant peu à l'aise avec le free ou memset, cela créait plus d'erreurs qu'autres choses. J'ai d'ailleurs remarqué que parfois le problème ne se manifestait pas, surtout quand on mettait des DEBUG_MSG (qui ne sont rien censés changer). Les enlever faisait réapparaître l'erreur (je n'ai toujours pas compris pourquoi d'ailleurs). Cela semblait se produire aussi

systématiquement quand le symbol se finissait par un "?", alors que ça n'est pas forcément le cas pour des symboles de type cons ou car.

J'ai résolu de manière un peu brouillonne le problème en copiant input dans la chaîne caractère par caractère, et une fois la limite sur input atteinte (un espace ou une parenthèse par exemple), on forçait l'ajout d'un caractère '\0'.
J'ai aussi fait un free manuel et peu propre à la fin de la fonction avec une boucle for mettant tous les caractères de la chaîne à '\0'. Sinon, quand elle était créée une deuxième fois lors d'un deuxième appel de la fonction, elle contenait encore les caractères de l'utilisation précédente (ce qui est évidemment la cause initiale du problème). Il est possible qu'il revienne car le code est peu robuste au niveau de la gestion de la mémoire.

5. Améliorations possibles/à faire

Comme dit plus haut, une gestion de la mémoire semble bienvenue, car même si le code marche en apparence, deux livrables plus tard on se retrouve avec des erreurs datants du début du code que l'on n'avait pas détectée. Nous ne sommes donc pas à l'abri de tels problèmes dans le futur.

Le programme en mode debug était saturé de DEBUG_MSG. J'ai dû les enlever dans le fichier read.c, alors qu'ils s'avéraient utiles en fin de compte. J'ai voulu créer un DEBUG_MSG2 dans le make et notify.h mais je n'ai pas pris le temps d'aller jusqu'au bout.

La gestion des réels seraient intéressante à avoir, si nous trouvons le temps.

La distinction entre nil et NULL est devenue floue au sens où on ne sait plus vraiment si on doit s'arrêter de parcourir la liste si son cdr est égal à nil ou à NULL. Cela est dû à une gestion des listes un peu lointaines et mal documentée. Par défaut, nous mettons les deux mais le programme pourrait gagner en clarté si on définissait mieux les deux cas.

Les tests concernant les primitives de manipulation de listes, les plus complexes selon nous, manquent d'exemples et de cas. Avec plus de temps, il faudrait les étoffer.