Charte pour l'écriture de programmes en Scheme

Cours « Programmation récursive »
Cycle L d'informatique – LI101
Université Pierre et Marie Curie

Revision: 1.48

Ce document précise comment présenter les programmes à écrire lors des travaux sur machine encadrés (TME) associés au cours « processus d'évaluation ». Ces programmes sont à remettre électroniquement en fin de chaque séance de TME et participent à la note de contrôle continu.

La méthode d'envoi de ces programmes, sous la forme d'un compte-rendu de TME, est détaillée plus bas.

Les règles de présentation ici décrites sont sous-tendues par des principes plus généraux qui se retrouvent dans tous les autres contextes de programmation de licence d'informatique quoique, bien sûr, chaque fois adaptés au génie des langages de programmation employés. Les règles particulières de ce document ne visent qu'à organiser les TME de licence du 1er semestre de 1ère année de l'UPMC et à optimiser la bonne appréciation de votre travail.

1 Présentation des programmes

Les programmes sont écrits dans un langage de programmation et comme tels se doivent de respecter les usages de ce langage. Le cours utilise Scheme comme langage de programmation et, plus particulièrement, DrScheme[1] comme environnement de développement de ces programmes.

À chaque séance de TME, vous aurez à constituer un fichier (nommé par exemple *tp*i.*scm*) dans lequel vous écrirez tous les programmes demandés à cette séance de TME.

À chaque exercice de TME est associé un programme qui est composé de commentaires, de fonctions et de tests.

Chaque programme sera présenté suivant les normes classiques (voir la carte de référence[2] de Scheme et les exemples ci-après).

- **renfoncement gauche**
  Les programmes sont présentés avec les bons alignements afin de permettre une lecture et relecture aisées. [Il existe une commande dans le menu *Scheme / Reindent all* pour ce faire. De plus, le curseur se positionne exactement là où il faut écrire quand on passe à la ligne (et si ce n'est pas là où vous l'escomptiez c'est qu'il y a une erreur avant), la touche de TABulation permet de renfoncer correctement la ligne courante.]
- **passage à la ligne** pour les paramètres des formes spéciales (`if`, `cond`, `begin`, `let`, `let*`).
- **commentaires**
  Les commentaires sont préfixés par des points-virgules dont le nombre indique la portée du commentaire.
- **blancs et passages à la ligne** Ne pas mettre de blanc entre une parenthèse ouvrante et le premier mot d'une liste. Ne pas passer à la ligne quand on ferme une parenthèse (les parenthèses fermantes ne se lisent pas, on les tape mais on ne les regarde jamais, les renfoncements gauches (ou *indentations* en jargon) suffisent à appréhender la structure des programmes. Ne pas non plus écrire une définition sur une seule ligne même si la définition est très courte !

[1] `http://www.plt-scheme.org/`
[2] `reference.html`

– etc.

Chaque programme sera suivi de ses tests. Ces tests seront exécutables et se présenteront sous la forme d'une expression (`verifier` *`fonction`* `...`) où *fonction* est le nom de la fonction qui est testée. Dans le cas où l'on vous demande plusieurs fonctions en même temps, ce groupe de fonctions porte un nom, visible dans le menu associé au bouton « Tester » et c'est ce nom qu'il faut utiliser après `verifier`.

Les tests généraux de bon emploi seront complétés de tests aux limites (sur la liste vide, sur zéro, etc.) Les tests pourront utiliser (cf. carte de références[3] pour plus de détails) :

– la fonction `erreur` pour signaler plus précisément la raison de l'échec. Cette dernière prend comme premier argument le nom de la fonction erronée suivie d'une chaîne de caractères précisant la nature de l'erreur.
– la fonction `erreur?` qui prend une fonction et ses arguments et renvoie vrai si l'application de cette fonction sur ses arguments conduit à une erreur.
– la forme spéciale `verifier` qui prend le nom d'une fonction suivie d'une suite d'expressions de tests.

1.1 Un exemple commenté

Ce qui suit n'est qu'un exemple ! La version HTML de cet exemple possède des annotations que votre souris peut dévoiler si elle se faufile au-dessus de ces zones sensibles.

```
;;; Exercice 1:-1- la factorielle

;;; fact:  nat -> nat-2-
;;; (fact n) calcule la factorielle de n-3-.

(define (fact n)                                                    -4-
  (if (= n 0)                                                       -5-
      1
      (* n (fact (- n 1))) ) )

;;; Les tests associés à-6- la factorielle.

(verifier fact
  (fact 1) == 1
  (fact 2) == 2
  (fact 4) == 24 )

;;; DIFFICULTÉS RENCONTRÉES
;;; écrire - n et non -n
;;; écrire (- n 1) et non (- 1 n) ni (-1 n)

;;; Exercice 2:-1- l'itérateur map

;;; map: (alpha -> beta) * LISTE[alpha] -> LISTE[beta]
;;; (map f l) rend la liste dont les termes sont les images
;;; par f des termes de la liste l.

(define (map f liste)                                               -4-
  ;; reduce: (alpha * beta -> beta) * beta * LISTE[alpha] -> beta
  ;; reduce(f, fin, l) = f(e1, f(e2, ... f(en, fin) ...)) avec l = (e1 e2 ... en)
  (define (reduce f fin liste)                                      -7-
    (if (pair? liste)                                               -5-
        (f (car liste)                    ; f est binaire
           (reduce f fin (cdr liste)) )
        fin ) )
  ;; composer: alpha * LISTE[beta] -> LISTE[beta]
```

[3] `reference.html`

```
  ;; (composer e r) ajoute l'image de e par f à r
  (define (composer element resultat)                                    -7-
    (cons (f element) resultat) )
  (reduce composer '() liste) )

;;; Tests.

;;; identite: alpha -> alpha
;;; la fonction identité
(define (identite x)                                                     -4-
  x )

(verifier map
  (map identite '(a b c d)) == (append '(a b) '(c d))
  (map car '((a 1)(b 2)(c 3)(d 4))) == (list 'a 'b 'c 'd) )
```

-1- Ceci est un commentaire global (préfixé par trois points-virgules) qui indique le début du programme réponse à une question du TME.

-2- Le type de la fonction `fact`

-3- La sémantique de la fonction `fact` ; elle ne doit pas surtout pas paraphraser l'algorithme employé. La spécification (c'est-à-dire le nom, le type et la sémantique) est nécessaire et doit décrire l'intention de la fonction ainsi que son interface d'emploi.

-4- Notez l'agencement d'une définition de fonction. Le nom et les variables sont sur la première ligne. Le corps de la définition apparaît sur les lignes suivantes.

-5- Notez l'agencement de l'alternative (la forme spéciale `if`) : ses trois paramètres sont alignés verticalement.

-6- Les tests permettant de tester les fonctions précédentes. Tous les tests individuels doivent retourner Vrai (c'est-à-dire `#t`) s'ils sont positifs. On pourra utiliser les fonctions `equal?`, `=`, ou les formes spéciales `and` ou `or` etc.

-7- Les fonctions internes sont précédés d'un commentaire bloc définissant leur spécification. Ces commentaires sont introduits par deux points-virgules et ne portent que sur l'expression qui suit.

-8- Un commentaire local (précédé par un unique point-virgule) ne porte que sur la ligne qu'il clôt. Il permet d'annoter fort localement des petits morceaux de code.

2 Divers

Ce document a vocation à s'enrichir. Dans ce but, il est disponible sur le réseau[4] ainsi qu'en version PDF[5] (ce qui s'imprime mieux).

Vous devez, pour tous les cours, TD ou TME ainsi qu'aux examens, porter sur vous la carte de référence[6] de Scheme.

Si vous avez des questions ou des remarques sur ce document, faites-le nous savoir [7]. Merci !

[4] `http://www.infop6.jussieu.fr/deug/2003/mias/mias-a/public/CharteScm.html`
[5] `http://www.infop6.jussieu.fr/deug/2003/mias/mias-a/public/CharteScm.pdf`
[6] `reference.ps`
[7] `mailto:mias.cederom2004@infop6.jussieu.fr`