Groupe	Nom	Prénom

Devoir sur table - Octobre 2013

LI101

Durée 1h30

Aucun document ni machine électronique n'est permis à l'exception de la carte de référence de Scheme.

Le sujet comporte 13 pages. Ne pas désagrafer les feuilles.

Répondre sur la feuille même, dans les boîtes appropriées. La taille des boîtes suggère le nombre de lignes de la réponse attendue. Le barème apparaissant dans chaque boîte n'est donné qu'à titre indicatif. Le barème total est lui aussi donné à titre indicatif : 51 points.

La clarté des réponses et la présentation des programmes seront appréciées. Les questions peuvent être résolues de façon indépendante. Il est possible, voire utile, pour répondre à une question, d'utiliser les fonctions qui sont l'objet des questions précédentes.

Pour vous faire gagner du temps, il ne vous est pas systématiquement demandé, en plus de la définition, la spécification entière d'une fonction. Bien lire ce qui est demandé : seulement la définition ? la signature et la définition ? la spécification et la définition ? seulement la spécification ?

Lorsque la signature d'une fonction vous est demandée, vous veillerez à bien préciser, avec la signature, les éventuelles hypothèses sur les valeurs des arguments.

Exercice 1

Manindra Agrawal, Neeraj Kayal et Nitin Saxena ont déterminé en 2002 une méthode pour savoir si un nombre entier est premier ou non. Cet exercice propose d'implémenter leur méthode.

Question 1.1

La fonction *factorielle* d'un entier naturel n, que l'on note $n!$, est définie par récurrence (rappel) :

$$n! = \begin{cases} 0 & \text{si } n = 0 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

Donnez la signature et une définition de la fonction `fac` telle que `(fac n)` donne `n!`.

[2/51]

Groupe Nom Prénom

Question 1.2

Le *coefficient binomial* de n et de k, noté $\begin{pmatrix} n \\ k \end{pmatrix}$, qui correspond au nombre de manières de choisir k éléments parmi n, est défini par :

$$\begin{pmatrix} n \\ k \end{pmatrix} = \frac{n!}{k!(n-k)!}$$

Pour que le coefficient binomial soit correctement défini, il faut que $n \geq k$. Notons que $\begin{pmatrix} n \\ n \end{pmatrix} = 1$, et si $k > n$, par convention, $\begin{pmatrix} n \\ k \end{pmatrix} = 0$.

Donnez <u>une définition</u> de la fonction de signature `binom: nat*nat -> nat` telle que `(binom n k)` donne la valeur de $\begin{pmatrix} \mathtt{n} \\ \mathtt{k} \end{pmatrix}$. Exemples :

`(binom 5 0)` → `1`

`(binom 5 3)` → `10`

`(binom 5 4)` → `5`

`(binom 5 5)` → `1`

`(binom 5 6)` → `0`

[2/51]

Question 1.3

Donnez <u>la signature et une définition</u> du prédicat `aks-ok?` tel que `(aks-ok? n k)` vaut `#t` si $\begin{pmatrix} n \\ k \end{pmatrix}$ est divisible par n, c'est-à-dire que le reste de la division entière de $\begin{pmatrix} n \\ k \end{pmatrix}$ par n est égal à 0 ; `(aks-ok? n k)` vaut `#f` sinon. Il faut supposer que $0 < \mathtt{k} \leq \mathtt{n}$.

Groupe | Nom | Prénom

[2/51]

Question 1.4

Donnez la signature et une définition du prédicat `aks-tous-ok?` telle que `(aks-tous-ok? n m)` vaut `#t` lorsque pour tout `k` $\in$ [1..m], $\binom{n}{k}$ est divisible par n, et vaut `#f` sinon.

Intuitivement, `(aks-tous-ok? n m)` donne la même valeur que `(and (aks-ok? n m) (aks-ok? n (- m 1)) ...(aks-ok? n 2))`.

Vous prendrez soin de bien indiquer quelles hypothèses il faut faire sur les arguments `n` et `m`.

[3/51]

Question 1.5

Agrawal, Kayal et Saxena ont basé leur méthode sur le résultat suivant :

> un entier naturel n est premier si et seulement si pour tout $k \in [1 \dots n-1]$, $\binom{n}{k}$ est divisible par n.

Déduisez de ce qui précède la signature et une définition du prédicat `premier-aks?` qui prend en argument un entier naturel et donne la valeur `#t` si ce nombre est premier, et la valeur `#f` sinon.

Les entiers 0 et 1 ne sont pas des nombres premiers, votre définition doit respecter les cas suivants :

`(aks-premier? 5)` $\rightarrow$ `#t`

Groupe Nom Prénom

`(aks-premier? 4)` → `#f`

`(aks-premier? 3)` → `#t`

`(aks-premier? 2)` → `#t`

`(aks-premier? 1)` → `#f`

`(aks-premier? 0)` → `#f`

[3/51]

Question 1.6

Donnez la signature et une définition de la fonction `liste-premiers` telle que `(liste-premiers n)` donne la liste des nombres premiers inférieurs ou égaux à l'entier naturel `n`.

[4/51]

Optimisation

Rappelons que la notation $\prod_{i=x}^{y} i$ désigne le produit des entiers de x à y (lorsque $x \leq y$) : $\prod_{i=x}^{y} i = x \times \cdots \times y$.

On peut donner une définition du coefficient binomial $\begin{pmatrix} \mathtt{n} \\ \mathtt{k} \end{pmatrix}$ qui demande moins de calculs. En

Groupe	Nom	Prénom

effet, si $k < n$ alors

$$n! = \prod_{i=1}^{n} i = \prod_{i=1}^{n-k} i \times \prod_{i=n-k+1}^{n} i = (n-k)! \times \prod_{i=n-k+1}^{n} i$$

Compte tenu de ces égalités, on pose :

$$\begin{pmatrix} \mathtt{n} \\ \mathtt{k} \end{pmatrix} = \frac{\prod\limits_{i=n-k+1}^{n} i}{k!} = \frac{\prod\limits_{i=n-k+1}^{n} i}{\prod_{i=1}^{k} i}$$

Question 1.7

Donnez <u>une définition</u> de la fonction `produit` telle que `(produit m n)` donne le produit des entiers compris entre `m` et `n` lorsque `n` $>$ `m`. Notons que `(produit m m)` vaut `m` et, si `m` $>$ `n` alors, on pose que `(produit m n)` vaut 1.

Exemples :

`(produit 1 5)` → `120`

`(produit 3 5)` → `60`

`(produit 5 5)` → `5`

`(produit 6 5)` → `1`

[2/51]

Question 1.8

En tenant compte de la définition ci-dessus, sans utiliser la fonction `fac`, mais en utilisant la fonction `produit` : donnez <u>la définition</u> de la fonction `binom-opt` qui prend en arguments deux entiers naturels et calcule leur coefficient binomial. Cette nouvelle fonction doit se comporter comme la fonction `binom` de la question 1.2 :

`(binom-opt 5 0)` → `1`

`(binom-opt 5 3)` → `10`

`(binom-opt 5 4)` → `5`

`(binom-opt 5 5)` → `1`

`(binom-opt 5 6)` → `0`

Groupe Nom Prénom

[2/51]

Exercice 2

Un *bit* (contraction de *binary digit*) est un nombre binaire, c'est-à-dire soit 0 soit 1. Les briques élémentaires constitutives des micro-processeurs que l'on trouve dans les ordinateurs manipulent des chaînes de bits. Dans cet exercice nous manipulons de telles chaînes dans le langage Scheme. Pour simplifier les notations, nous introduisons le type `Bit` représentant les entiers 0 ou 1. Une chaîne de bits sera représentée par le type `LISTE[Bit]`.

Les opérations élémentaires sur les chaînes de bits sont : le *non binaire*, le *et binaire* et le *ou binaire*. Ces opérations élémentaires sont au cœur des calculs informatiques.

Question 2.1

Le principe du *non binaire* est d'inverser la valeur d'un bit : 1 devient 0 et 0 devient 1.

Donner une définition de la fonction `non-binaire` dont la spécification est la suivante :

```
;;; non-binaire: Bit -> Bit
;;; (non-binaire b) inverse le bit b
```

[1/51]

Question 2.2

Donner la signature et une définition de la fonction `non` qui généralise le non binaire aux chaînes de bits. C'est-à-dire que chaque bit 0 de la chaîne est transformé en 1 et vice-versa.

Par exemple :

`(non (list 1 0))` → `(0 1)`

`(non (list 1 1 0 0 1 0 1 0))` → `(0 0 1 1 0 1 0 1)`

`(non (list))` → `()`

Groupe Nom Prénom

[3/51]

Question 2.3

Le principe du *et binaire* est de combiner deux bits pour produire un bit résultat :
- valant 1 si et seulement si les deux bits de départ sont également à 1
- et 0 sinon

Donner la signature et une définition de la fonction `et-binaire` qui réalise cette combinaison.

[1.5/51]

Question 2.4

L'opération de *et binaire* décrite précédemment se généralise aux chaînes de bits avec la spécification suivante :

```
;;; et: LISTE[Bit] * LISTE[Bit] -> LISTE[Bit]
;;; (et B1 B2) combine par un et-logique les bits de B1 et de B2
;;; HYPOTHESE: les listes B1 et B2 sont de même longueur
```

Par exemple :

```
(et  (list 1 0 1 1 0 0 1)
     (list 0 1 1 0 0 1 1))
→ (0 0 1 0 0 0 1)

(et (list) (list)) → ()
```

Donner une définition de la fonction `et`.

Groupe	Nom	Prénom

[3/51]

Question 2.5

Le principe du *ou binaire* est de combiner deux bits pour produire un bit résultat :
- valant 1 si au moins l'un des deux bits de départ vaut 1
- et 0 sinon

Cette opération se généralise aussi naturellement aux chaînes de bits, avec la spécification suivante :

```
;;; ou: LISTE[Bit] * LISTE[Bit] -> LISTE[Bit]
;;; (ou B1 B2) combine par un ou binaire les bits de B1 et de B2
;;; HYPOTHESE: les listes B1 et B2 sont de même longueur
```

Par exemple :

```
(ou (list) (list)) → ()
(ou (list 0 1 1) (list 1 0 1)) → (1 1 1)

(ou  (list 1 0 1 1 0 0 1)
     (list 0 1 1 0 0 1 1))
→ (1 1 1 1 0 1 1)
```

Donner une définition de la fonction `ou`.

[4/51]

Groupe	Nom	Prénom

Question 2.6

La porte logique *nand* ou `non-et` est un combinateur de chaînes de bits dit universel et présent en exclusivité dans certains micro-processeurs. Il s'agit de combiner deux chaînes de bits avec `et` puis d'inverser le résultat avec `non`.

Donner la signature et une définition de la fonction `non-et` telle que, par exemple :

```
(non-et  (list 1 0 1 1 0 0 1)
         (list 0 1 1 0 0 1 1))
→        (1 1 0 1 1 1 0)
```

[2/51]

Question 2.7

La fonction `non-et` est dite universelle car on peut l'utiliser de manière exclusive pour encoder toutes les fonctions imaginables sur les chaînes de bits. On peut notamment redéfinir les fonctions `non`, `et` et `ou` uniquement par des appels à `non-et`.

Par exemple, une définition alternative de la fonction `non` est la suivante :

```
;;; non-avec-non-et: LISTE[Bit] -> LISTE[Bit]
;;; (non-avec-non-et B) retourne la même valeur que (non B)
(define (non-avec-non-et B)
  (non-et B B))
```

Soient les deux fonctions mystères suivantes :

```
(define (mystere-1 B1 B2)
  (non-et (non-et B1 B2) (non-et B1 B2)))

(define (mystere-2 B1 B2)
  (non-et (non-et B1 B1) (non-et B2 B2)))
```

Indiquer qui de `mystere-1` ou `mystere-2` réalise le même calcul que la fonction `et` et qui réalise le même calcul que la fonction `ou`.

[2/51]

Groupe Nom Prénom

Exercice 3

Dans cet exercice nous nous intéressons à la représentation en binaire des nombres entiers décimaux. En effet les humains préfèrent manipuler les nombres en base 10 (nombre décimaux) alors que les ordinateurs préfèrent de loin la base 2 (nombres binaires). Il existe plusieurs formats de représentation, le plus simple étant le codage *BCD* d'un entier naturel qui consiste à coder chacun de ses chiffres individuellement, l'un après l'autre.

Question 3.1

Pour coder un chiffre décimal n (entre 0 et 9) en chaîne de bits, on utilise la formule suivante :

$n = b_8 b_4 b_2 b_1$ avec $b_k =$ `(remainder (quotient` n k`) 2)`

Par exemple pour le codage de $n = 5$ on a :

$b_8 =$ `(remainder (quotient 5 8) 2)` $= 0$
$b_4 =$ `(remainder (quotient 5 4) 2)` $= 1$
$b_2 =$ `(remainder (quotient 5 2) 2)` $= 0$
$b_1 =$ `(remainder (quotient 5 1) 2)` $= 1$

Donc le codage binaire de 5 est la chaîne de bits `(0 1 0 1)`.

En déduire la signature et une définition de la fonction `bcd-chiffre` qui code un chiffre décimal en une chaîne de bits de longueur 4.

On aura ainsi :

`(bcd-chiffre 0)` → `(0 0 0 0)`	`(bcd-chiffre 5)` → `(0 1 0 1)`
`(bcd-chiffre 1)` → `(0 0 0 1)`	`(bcd-chiffre 6)` → `(0 1 1 0)`
`(bcd-chiffre 2)` → `(0 0 1 0)`	`(bcd-chiffre 7)` → `(0 1 1 1)`
`(bcd-chiffre 3)` → `(0 0 1 1)`	`(bcd-chiffre 8)` → `(1 0 0 0)`
`(bcd-chiffre 4)` → `(0 1 0 0)`	`(bcd-chiffre 9)` → `(1 0 0 1)`

[2/51]

Groupe Nom Prénom

Question 3.2

La conversion d'un entier décimal en binaire BCD est très simple. Chaque chiffre de l'entier est converti indépendamment. Ainsi l'entier 609 est codé par la chaîne de bits `(0 1 1 0 0 0 0 0 1 0 0 1)`, soit le codage de 6 `(0 1 1 0)` suivi du codage de 0 `(0 0 0 0)` suivi du codage de 9 `(1 0 0 1)`.

En guise de simplification, on considérera que les entiers en base 10 sont représentés par une liste de chiffres entre 0 et 9. Ainsi l'entier 609 sera représenté par `(list 6 0 9)`, l'entier 10 par `(list 1 0)`, etc.

En déduire <u>une définition</u> de la fonction dont la spécification est la suivante :

```
;;; bcd: LISTE[Nat] -> LISTE[Bit]
;;; (bcd D) retourne le codage BCD binaire de l'entier D codé en décimal
```

Par exemple :

`(bcd (list 6 0 9))` → `(0 1 1 0 0 0 0 0 1 0 0 1)`

`(bcd (list 1 0 0))` → `(0 0 0 1 0 0 0 0 0 0 0 0)`

`(bcd (list 0))` → `(0 0 0 0)`

`(bcd (list))` → `()`

[3.5/51]

Le codage BCD n'est pas très efficace car la représentation de chaque chiffre décimal consomme exactement 4 bits. Un codage plus efficace – celui généralement utilisé en pratique – consiste à effectuer une conversion directement de la représentation en base 10 vers la représentation en base 2, comme c'est le cas pour `bcd-chiffre` mais pour des entiers naturels quelconques.

Ce codage convertit un entier n en une chaîne de bits $(b_k \cdots b_2\ b_1\ b_0)$ telle que :

$$n = b_0 \times 2^0 + b_1 \times 2^1 + b_2 \times 2^2 + \ldots + b_k \times 2^k$$

La chaîne de bits peut se construire étape par étape selon le principe suivant :

- b_0 = `(remainder n 2)`
- b_1 = `(remainder (quotient n 2) 2)`
- b_2 = `(remainder (quotient n 4) 2)`
- ...
- b_k = `(remainder (quotient n` 2^k`) 2)`

Par exemple, la conversion de l'entier 609 donnera la chaîne $(b_9\ b_8\ \ldots\ b_2\ b_1\ b_0)$ avec :

Groupe Nom Prénom

- b_0 = (remainder 609 2) = 1
- b_1 = (remainder 304 2) = 0
- b_2 = (remainder 152 2) = 0
- b_3 = (remainder 76 2) = 0
- b_4 = (remainder 38 2) = 0
- b_5 = (remainder 19 2) = 1
- b_6 = (remainder 9 2) = 1
- b_7 = (remainder 4 2) = 0
- b_8 = (remainder 2 2) = 0
- b_9 = (remainder 1 2) = 1

Donc l'entier 609 codé en binaire donne la chaîne `(1 0 0 1 1 0 0 0 0 1)`.

Question 3.3

On remarque que le codage binaire d'un nombre entier construit d'abord le dernier bit de la chaîne résultat, puis l'avant-dernier, etc. jusqu'au premier. Puisqu'en Scheme les listes se construisent dans le sens inverse, on aura intérêt à définir une fonction utilitaire `renverse` dont la spécification est la suivante :

```
;;; renverse: LISTE[alpha] -> LISTE[alpha]
;;; (renverse L) retourne une liste composée des éléments de L
;;; en ordre inverse
```

Par exemple :

`(renverse (list 1 2 3 4 5))` → `(5 4 3 2 1)`

`(renverse (list 1 0 0 0 0 1 1 1 0 0 1))` → `(1 0 0 1 1 1 0 0 0 0 1)`

`(renverse (list))` → `()`

Donner une définition de la fonction `renverse`.

[4/51]

Groupe Nom Prénom

Question 3.4

La fonction de conversion d'un entier naturel décimal en binaire est la suivante :

```
;;; bin: Nat -> LISTE[Bit]
;;; (bin n) retourne le codage binaire de l'entier n
(define (bin n)
  (renverse (bin-inv n)))
```

Cette fonction utilise la fonction auxiliaire `bin-inv` dont la spécification est la suivante :

```
;;; bin-inv: Nat -> LISTE[Bit]
;;; (bin-inv n) retourne le codage binaire en sens inverse de l'entier n
```

Par exemple :

`(bin-inv 609)` → `(1 0 0 0 0 1 1 0 0 1)`

et avec `bin` on obtient la chaîne dans le bon sens :

`(bin 609)` → `(1 0 0 1 1 0 0 0 0 1)`

Donner une définition de la fonction `bin-inv`.

[5/51]