Consignes

- Le seul document autorisé est une feuille A4 manuscrite recto-verso.
- Tous les appareils électroniques doivent être éteints et rangés hors de vue.
- Vous devez rendre le sujet complété avec votre copie. Assurez-vous d'y avoir apposé votre nom.
- La clarté et la concision de vos réponses seront appréciées.

1 Analyses statiques

Le compilateur Heptagon emploie quatre analyses statiques : le typage de données, l'analyse d'initialisation, l'analyse de causalité et le calcul d'horloge. L'objectif de cet exercice est de préciser le rôle de chacune.

1. Caractériser en une phrase les analyses *statiques* de programmes, par opposition aux analyses dynamiques.
2. Pour chacun des blocs d'équations ci-dessous, déterminer s'il est bien typé (en termes de données), bien initialisé, causal, et valide du point de vue des horloges. On considèrera les analyses indépendamment. On supposera que toutes les variables utilisées ont été déclarées de type entier. On supposera que la variable `c` désigne une entrée booléenne. Aucune variable n'a reçu d'annotation d'horloge.

   Vous répondrez directement sur le sujet en écrivant *oui* dans chaque case du tableau ci-dessous si le bloc d'équation est valide pour l'analyse correspondante, et *non* sinon. Les colonnes D, I, C et H désignent respectivement le typage de données, l'analyse d'initialisation, l'analyse de causalité et le calcul d'horloge.

Équations	D	I	C	H	
`x = 0 fby y ; y = x + 1`					
`x = 0 -> y ; y = x + 1`					
`x = 0 -> y ; y = pre x`					
`x = 0 -> y ; y = true -> x`					
`x = 42 ; y = 42; z = merge c x y`					
`x = 42 ; y = 42; z = merge c x x`					
`x = 42 ; y = (x + x) when c`					
`x = 42 ; y = x + (x when c)`					
`x = 42 fby y ; y = (x +. x) when c`					
`x = 42 ; y = (x when c) + (x when c)`					

2 Manipulation de tableaux : itération et accès indicé

L'objectif de cet exercice est de manipuler des tableaux via les itérateurs d'Heptagon (**`map`**, **`fold`**, etc.) ou bien via les opérations primitive d'accès indicé.

1. Considérons les équations `y1 = map<<10>> f(x1, x2)` et `y2 = fold<<5>> f(x3, x4)`. Supposons que `f` soit une fonction attendent deux arguments de type **`bool`** et **`float`**, et que son résultat soit de type **`float`**. Donner les types attendus de `x1`, `x2`, `y1`, `x3`, `x4` et `y2`.
2. Définir, en utilisant l'itérateur **`fold`**, un nœud

   `node` `sumproduct<<n : int>>(x : float^n) returns (o : float)`

   dont la sortie, à l'instant k, est la somme des produits de tous les tableaux reçus jusqu'à l'instant k inclus, comme exprimé par la formule ci-dessous.

$$(\texttt{sumproduct<<n>>(x)})_k = \sum_{i \leq k} \prod_{j=0}^{n-1} x_i[j]$$

3. Définir, en utilisant l'itérateur **mapi**, une fonction

   fun swap<<n : **int**>>(x : **int**^n) **returns** (o : **int**^n)

   qui, à chaque instant, renverse le tableau reçu en entrée. Par exemple, `swap<<3>>([1, 2, 3])` est égal à la suite constante `[3, 2, 1]`.
   On rappelle que l'itérateur **mapi** est identique à **map**, à ceci près qu'il fournit à la fonction appliquée aux éléments du tableau l'indice de l'élément courant comme dernier argument. Ainsi, on a par exemple

   $$\textbf{mapi}\texttt{<<n>> f}([x_0, x_1, \ldots, x_{n-1}]) \;=\; [f(x_0, 0), f(x_1, 1), \ldots, f(x_{n-1}, n-1)].$$

4. Définir un nœud

   node serialize<<n : **int**>>(x : **int**^n) **returns** (o : **int**)

   tel que $\mathrm{o}_{in+j} = \mathrm{x}_{in}[j]$ pour tout $0 \leq i$ et $0 \leq j < n$. Voir ci-dessous pour un exemple où $n = 3$.

xs	$[0,1,2]$	$[4,5,2]$	$[5,8,7]$	$[5,7,8]$	$[3,1,3]$	$[8,0,0]$	$[1,2,4]$	$[3,5,3]$	$[6,6,4]$	...
o	0	1	2	5	7	8	1	2	4	...

5. Définir un nœud

   node parallelize<<n : **int**>>(x : **int**) **returns** (c : **bool** ; o : **int**^n :: . **on** c)

   tel que c_k est vrai si et seulement k est égal à $(i+1)n - 1$ où $i \geq 0$, et que $o_{(\ell+1)n-1}[j] = \mathrm{x}[\ell n + j]$ où $\ell \geq 0$ et $0 \leq j < n$. Voir ci-dessous pour un exemple où $n = 3$.

x	0	1	2	3	4	5	6	7	8	...
c	false	false	true	false	false	true	false	false	true	...
o	*abs*	*abs*	$[0,1,2]$	*abs*	*abs*	$[3,4,5]$	*abs*	*abs*	$[6,7,8]$	...

3 C'est l'alarme !

On s'intéresse à la conception d'un détecteur d'intrusion simplifié. On supposera que chaque instant synchrone dure $0,001$ secondes. Le détecteur a accès aux données produites par un capteur de mouvement et par un capteur de température. Le premier indique si du mouvement a été détecté durant l'instant logique écoulé, le second la température maximale dans la pièce. Ces données sont regroupées par le type suivant.

type events = { move : **bool**; heat : **float** }

1. Écrire un nœud

   node time_spent_moving(e : events) **returns** (t : **float**)

   dont la sortie contient la durée du dernier segment de mouvement continu mesuré lorsque du mouvement est détecté, ou 0 sinon.

2. Écrire un nœud

   node average_heat(e : events) **returns** (m : **float**)

   qui renvoie la moyenne des températures maximales mesurées durant la dernière seconde.

3. Écrire un nœud

   node detect(e : events) **returns** (a : **bool**)

   dont la sortie passe de faux à vrai lorsqu'on a détecté du mouvement continu ainsi qu'une moyenne des températures maximales supérieure ou égale à 35 degrés durant la dernière seconde. Une fois l'alarme enclenchée, celle-ci doit rester activée jusqu'à la fin de l'exécution.