CCP 2006 — option informatique

Partie I — Automates et langages

Question I.1

Le langage reconnu par l'automate $\mathcal{E}_1$ est dénoté par l'expression régulière `ab*c`. (On aura observé que l'état D n'est pas co-accessible.)
Le langage reconnu par l'automate $\mathcal{E}_2$ est dénoté par l'expression régulière `(ac)*a` ou encore par `a(ca)*`. (On aura observé que l'état G n'est pas co-accessible.)

Question I.2

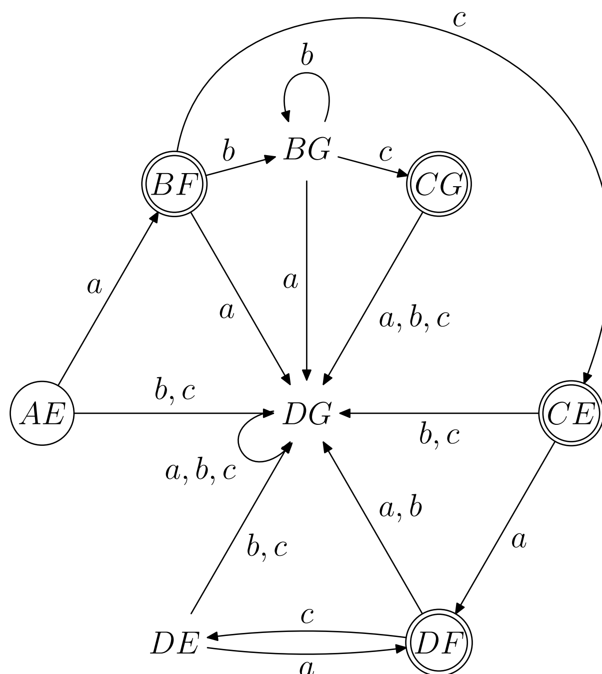

Figure 1 l'automate $\mathcal{E}_1 + \mathcal{E}_2$

Question I.3

On observe que DG est un état-puits. En outre, un calcul de l'automate qui ne passe pas par cet état-puits bifurque en BF ou bien vers BG puis CG ou bien vers CE, après quoi on tombe dans l'alternance DF/DE.
Le langage reconnu par l'automate $\mathcal{E}_1 + \mathcal{E}_2$ est donc dénoté par l'expression régulière `a|ab*c|ac|aca(ca)*` qu'on peut simplifier en `ab*c|a(ca)*`.
Autrement dit le langage reconnu est l'union des langages reconnus par chacun des deux automates précédents.

Question I.4

Il y a bien un seul état initial, et la fonction δ_{1+2} est bien totale puisque δ_1 et δ_2 sont supposées également totales. L'automate $\mathcal{A}_1 + \mathcal{A}_2$ est donc bien déterministe et complet.

Question I.5

Il suffit de procéder par récurrence sur la longueur de m, notée $|m|$.
La propriété est triviale si $|m| = 0$, et la définition de δ_{1+2} fournit la preuve quand $|m| = 1$.
Soit alors e un caractère quelconque et m un mot quelconque sur l'alphabet X.
On écrit $\delta^*_{1+2}((o_1, o_2), e.m) = (d_1, d_2)$ quand $\delta_{1+2}((o_1, o_2), e) = (q_1, q_2)$ et $\delta^*_{1,2}((q_1, q_2), m) = (d_1, d_2)$.

La première propriété signifie que $\begin{cases} \delta_1(o_1, e) = q_1 \\ \delta_2(o_2, e) = q_2 \end{cases}$ et la seconde que $\begin{cases} \delta^*_1(q_1, m) = d_1 \\ \delta^*_2(q_2, m) = d_2 \end{cases}$ d'après l'hypothèse de récurrence.

La définition de δ^*_1 et de δ^*_2 assure alors que $\begin{cases} \delta^*_1(o_1, e.m) = d_1 \\ \delta^*_2(o_2, e.m) = d_2 \end{cases}$ ce qui prouve que la propriété est bien héréditaire, et achève la démonstration.

Question I.6
Dire qu'un mot m est reconnu par $\mathcal{A}_1 + \mathcal{A}_2$, c'est dire que $\delta^*_{1+2}((o_1, o_2), m) = (d_1, d_2)$ est un état final, c'est-à-dire que $d_1 \in T_1$ ou que $d_2 \in T_2$. Mais on vient de démontrer que $d_1 = \delta^*_1(o_1, m)$ et que $d_2 = \delta^*_2(o_2, m)$, donc finalement m est reconnu par $\mathcal{A}_1 + \mathcal{A}_2$ si et seulement si m est reconnu par $\mathcal{A}_1$ ou par $\mathcal{A}_2$.

Question I.7
Autrement dit : $L(\mathcal{A}_1 + \mathcal{A}_2) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$.

Partie II — Algorithmique et programmation en Caml

Préliminaires

Question II.1

A	B	$A \oplus B$
V	V	F
V	F	V
F	V	V
F	F	F

Table 1 la table de vérité de $A \oplus B$

Question II.2
On peut écrire : $A \oplus B = (A \vee B) \wedge (\neg A \vee \neg B)$ (forme normale conjonctive) et $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$ (forme normale disjonctive).

Question II.3

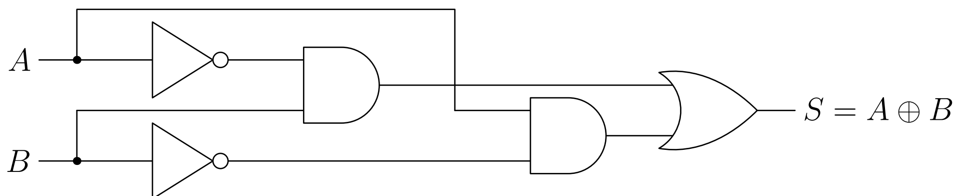

Figure 2 un circuit pour le ou exclusif

Question II.4
On écrit sans difficulté le programme 1.

```
let xor a b =
  if a then not b else b ;;
```

Programme 1 la fonction `xor`

Question II.5

Si $n = 0$, $\ell = 0$, et si $n \geqslant 1$, on dispose de $2^\ell \leqslant n \leqslant \sum_{k=0}^{\ell} 2^k = 2^{\ell+1} - 1$ de sorte que $\ell = \lfloor \lg n \rfloor$.

Question II.6
Le bit de poids faible de n est égal à 0 si et seulement si n est pair. On en déduit le programme 2, page 3, en faisant attention au cas particulier $n = 0$.

Question II.7
La complexité de la fonction `codage` est évidemment $O(\ell) = O(\lg n)$.

Question II.8
On écrit sans difficulté le programme 3, page 3.

```
3   type entier == bool list ;;

4   let codage n =
5     let rec codageRec = function
6       | 0 -> []
7       | n -> (n mod 2 = 1) :: (codageRec (n/2))
8     in
9       if n = 0 then [ false ] else codageRec n ;;
```

Programme 2 la fonction `codage`

```
10  let rec decodage = function
11    | [] -> 0
12    | true :: q -> 1 + 2 * (decodage q)
13    | false :: q -> 2 * (decodage q) ;;
```

Programme 3 la fonction `decodage`

Question II.9

Incrémenter n revient à propager la retenue dans l'addition bit à bit en partant de $r_0 =$ `true`.

On écrit donc pour $0 \leqslant i \leqslant \ell$, $\begin{cases} v_i = b_i \oplus r_i \\ r_{i+1} = b_i \wedge r_i \end{cases}$ avec $r_0 =$ `true`, $m = \ell$ si $r_{\ell+1} =$ `false`, c'est-à-dire si $n \leqslant 2^\ell - 2$, et $m = \ell + 1$ dans le cas contraire. On a alors toujours $v_m =$ `true`, ce qu'impose d'ailleurs la notation choisie.

On peut préférer convenir qu'on a toujours $m = \ell + 1$ à condition de poser simplement $v_m = r_m = r_{\ell+1}$.

Question II.10

On dessine en figure 3, page 4, un incrémenteur 1-bit à l'aide de seulement deux portes logiques.

Question II.11

Notons que comme $r_0 =$ `true`, on a simplement $v_0 = \neg b_0$ et $r_1 = b_0$. On en déduit le schéma de l'incrémenteur 3-bits de la figure 4, page 4.

Question II.12

On écrit facilement le programme 4, page 4.

b_i	r_i	v_i	r_{i+1}
V	V	F	V
V	F	V	F
F	V	V	F
F	F	F	F

Table 2 l'addition 1-bit

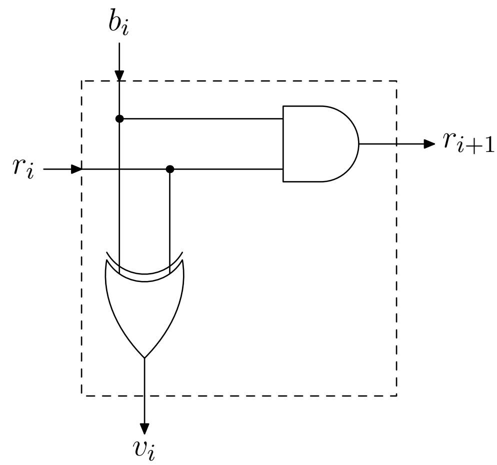

Figure 3 un incrémenteur 1-bit

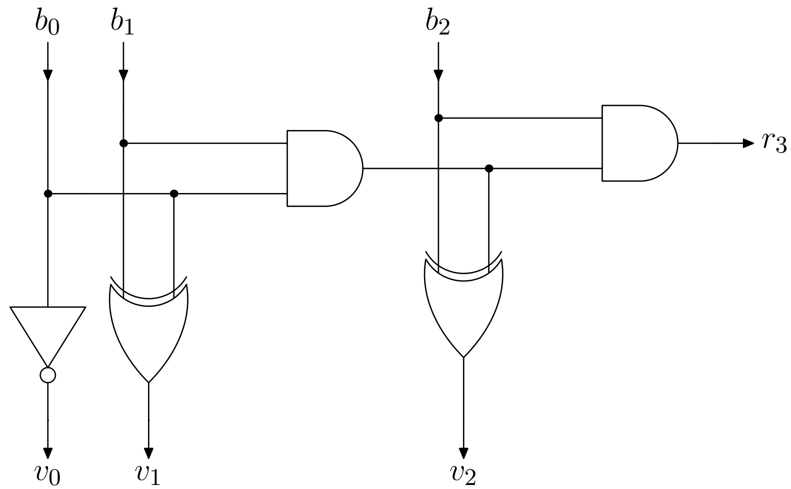

Figure 4 un incrémenteur 3-bits

```
14  let rec incrementation = function
15    | [] -> []
16    | true :: q -> false :: (incrementation q)
17    | false :: q -> true :: q ;;
```

Programme 4 la fonction `incrementation`

Question II.13
La table 3 permet de vérifier qu'on a, dans le cas où $m \geqslant n$ (le cas $m \leqslant n$ est analogue) :

$$\forall i \in \{0, \dots, n\}, \quad \begin{cases} v_i = (a_i \oplus b_i) \oplus r_i \\ r_{i+1} = (a_i \wedge b_i) \vee ((a_i \oplus b_i) \wedge r_i) \end{cases}$$

$$r_0 = \texttt{false}$$

$$\forall i \in \{n+1, \dots, m\}, \quad \begin{cases} v_i = a_i \oplus r_i \\ r_{i+1} = a_i \wedge r_i \end{cases}$$

$$p = m + 1$$

$$v_p = r_p$$

a_i	b_i	r_i	v_i	r_{i+1}
V	V	V	V	V
V	V	F	F	V
V	F	V	F	V
V	F	F	V	F
F	V	V	F	V
F	V	F	V	F
F	F	V	V	F
F	F	F	F	F

Table 3 l'addition de deux bits avec retenue

Question II.14
On propose le circuit de la figure 5, page 5.

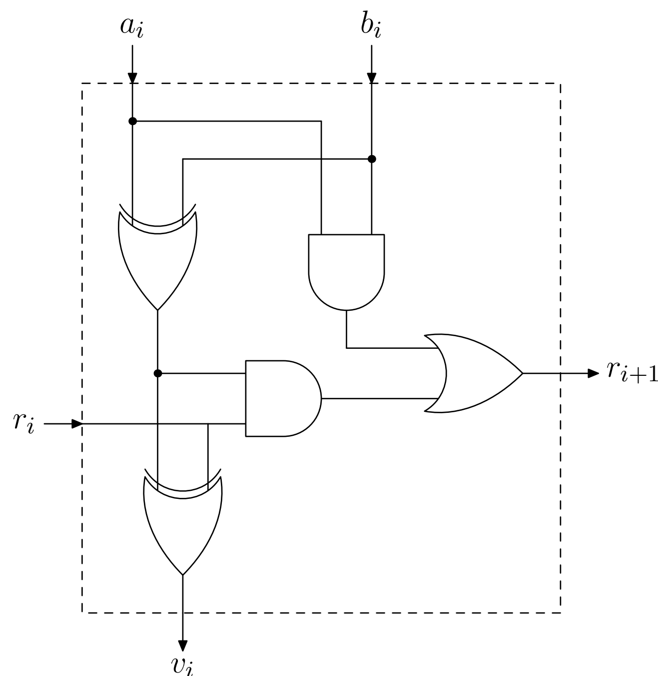

Figure 5 l'additionneur 1-bit

Question II.15
En combinant les circuits précédents, on obtient facilement le circuit de la figure 6, page 6.

Question II.16
Le programme 5, page 6, reprend les résultats précédents : on utilise d'abord des cellules d'addition (appels à `ajouteBits`), puis des cellules d'incrémentation (appels à `incrementation`).

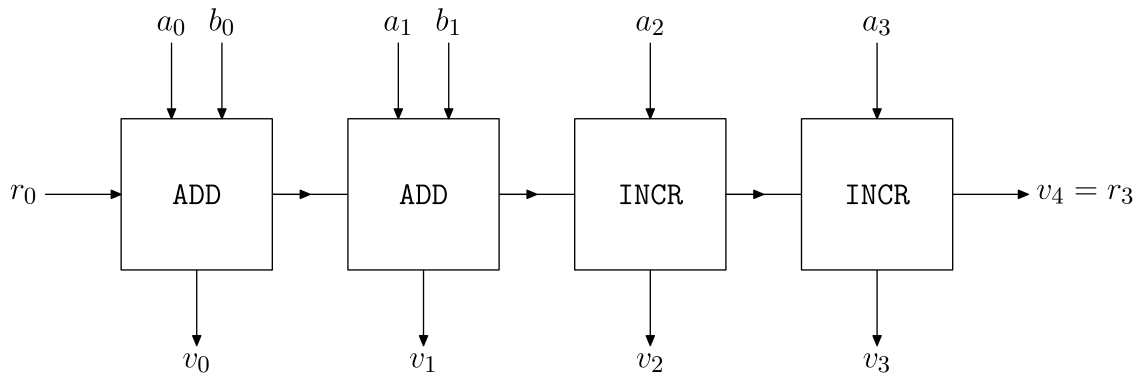

Figure 6 un circuit additionneur

```
18  let ajouteBits a b r =
19    let r' =   (a && b && r)         || (a && b && not r)
20            || (a && not b && r)     || (not a && b && r)
21    and v  =   (a && b && r)         || (not a && not b && r)
22            || (not a && b && not r) || (a && not b && not r)
23    in
24      (r',v) ;;

25  let addition e1 e2 =
26    let rec additionAvecRetenue r e1 e2 = match (e1,e2) with
27      | [],[] -> if r then [ true ] else []
28      | _,[] -> if r then incrementation e1 else e1
29      | [],_ -> if r then incrementation e2 else e2
30      | b1 :: q1, b2 :: q2 ->
31         let (r',v) = ajouteBits b1 b2 r in
32           v :: (additionAvecRetenue r' q1 q2)
33    in
34      additionAvecRetenue false e1 e2 ;;
```

Programme 5 la fonction `addition`

Question II.17
On écrit par exemple :
$$A(a) = (a = \emptyset) \text{ ou } (\mathcal{E}(a) \in \mathbb{N} \text{ et } \forall f \in \mathcal{S}(a), A(f)).$$

Question II.18
On a utilisé ici la fonction `it_list` dont on rappelle la définition.

```
35  type arbre =
36    | Vide
37    | Noeud of int * arbres
38  and arbres == arbre list ;;

39  (* rappel de la définition de it_list : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a *)
40  let rec it_list f a = function
41    | [] -> a
42    | t :: q -> it_list f (f a t) q ;;

43  let rec taille = function
44    | Vide -> 0
45    | Noeud(_,fils) -> it_list (fun x f -> taille f + x) 1 fils ;;
```

Programme 6 la fonction `taille`

Question II.19
On aura
$$\mathcal{H}(a) = \begin{cases} \text{non définie,} & \text{si } a = \emptyset \text{ ;} \\ 0, & \text{si } \mathcal{S}(a) \text{ est une liste vide ;} \\ 1 + \max\limits_{f \in \mathcal{S}(a)} \mathcal{H}(f), & \text{sinon.} \end{cases}$$

Question II.20
Un arbre de taille n de profondeur maximale est un arbre où chaque nœud possède un unique fils. Sa hauteur vaut $n - 1$.
Un arbre de taille n de profondeur minimale est constitué d'un nœud possédant $n - 1$ fils. Sa hauteur vaut 1.

Question II.21
Il faut faire très attention au cas de l'arbre vide ou des fils vides.

```
46  let rec hauteur = function
47    | Vide -> -1
48    | Noeud(_,fils) -> it_list (fun x f -> max (hauteur f) x) (-1) fils + 1 ;;
```

Programme 7 la fonction `hauteur`

Question II.22
On écrit par exemple :
$$AT(\emptyset) \text{ et } \left(AT(a) \iff \forall f \in \mathcal{S}(a), \begin{cases} \mathcal{E}(f) > \mathcal{E}(a) \\ AT(f) \end{cases} \right).$$

Question II.23
On en déduit le programme 8, page 8. On a réécrit la fonction standard `for_all : ('a -> bool) -> 'a list -> bool` qui vérifie qu'un prédicat est satisfait par tous les éléments d'une liste, et introduit, pour plus de lisibilité, une fonction `racine`.

```
49  let racine = function
50    | Vide -> failwith "arbre vide"
51    | Noeud(a,_) -> a ;;

52  let rec for_all predicat = function
53    | [] -> true
54    | t :: q -> predicat t && for_all predicat q ;;

55  let rec validerAT = function
56    | Vide -> true
57    | Noeud(a,fils) -> for_all (function f -> racine f > a && validerAT f) fils ;;
```

Programme 8 la fonction `validerAT`

Question II.24

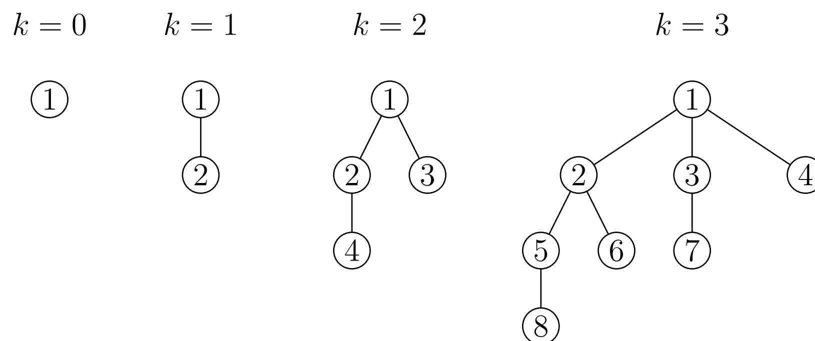

Figure 7 les premiers arbres binomiaux

Question II.25
Les sous-arbres d'un arbre en tas sont eux-mêmes en tas. Réciproquement si B et C sont en tas, et si la racine de C est strictement plus grande que la racine de B, la construction proposée fournira bien un arbre en tas.
Si A est binomial d'ordre $k+1$, le premier fils (à gauche) est C, binomial d'ordre k. Viennent ensuite des fils d'ordres k, puis $k-1$, ..., puis 1, puis 0 : ils constituent ensemble un arbre B qui est lui-même binomial, d'ordre k.
Réciproquement si on accole selon la construction proposée deux arbres binomiaux d'odre k, on obtient bien un arbre binomial d'ordre $k+1$, ce qu'on vérifie en examinant les fils de la racine de A qui sont d'abord C lui-même, puis les fils de B.

Question II.26
Soit n_k la taille d'un arbre binomial d'ordre k. D'après la construction précédente, on a $n_{k+1} = 2n_k$ donc naturellement $n_k = 2^k$.

Question II.27
Soit h_k la hauteur d'un arbre binomial d'ordre k : on aura $h_{k+1} = 1 + h_k$ d'après la construction précédente, donc bien sûr $h_k = k$.

Question II.28
Remarque : la notation des coefficients binomiaux dans l'énoncé est erronée car inversée !

On raisonne par récurrence sur k : les premiers exemples de la figure 7 permettent de vérifier que le résultat est acquis pour $k \leqslant 3$. Supposant la propriété vérifiée au rang k, on considère un arbre binomial A d'ordre $k+1$ qu'on décompose selon la construction du 3.2 en deux arbres binomiaux B et C d'ordre k : il y a alors dans B $\binom{k}{p}$ nœuds à la profondeur p et dans C $\binom{k}{p-1}$ nœuds à la profondeur $p-1$, donc à la profondeur p dans A. Au total on a bien trouvé $\binom{k}{p} + \binom{k}{p-1} = \binom{k+1}{p}$ nœuds à la profondeur p dans A et la récurrence s'enclenche.

Question II.29
On peut écrire que $ABT_0(a) \iff \mathcal{S}(a) = \emptyset$ et

$$ABT_{k+1}(a) = (\mathcal{E}(a) < \mathcal{E}(\text{fst}(\mathcal{S}(a))) \wedge ABT_k(\text{fst}(\mathcal{S}(a))) \wedge ABT_k(b))$$

où b est l'arbre obtenu à partir de a en supprimant le premiers fils de a, et où fst désigne la fonction qui renvoie le premier élément d'une liste.

Question II.30

```
58  let rec validerABT k a = match k with
59    | 0 -> ( match a with
60             | Vide -> false
61             | Noeud(_,fils) -> fils = [] )
62    | _ -> ( match a with
63             | Noeud(x, f :: q) -> racine f > x && validerABT (k-1) f && validerABT (k-1) (Noeud(x,q))
64             | _ -> false ) ;;
```

Programme 9 la fonction `validerABT`

Question II.31

D'après la question II.26, nous avons $T(t) = \sum_{k=0}^{\ell} s_k 2^k$. La signature n'est autre que l'écriture binaire de la taille du tas.

Question II.32
On vient de le dire, non ?

Question II.33
On peut écrire si $t = a_0, a_1, \ldots, a_\ell$:

$$TB(t) = \wedge_{k=0}^{\ell}(a_k = \emptyset \vee ABT_k(a_k)).$$

Question II.34

```
65  let validerTB t =
66    let rec valide k = function
67      | [] -> true
68      | a :: q -> (a = Vide || validerABT k a) && valide (k+1) q
69    in
70      valide 0 t ;;
```

Programme 10 la fonction `validerTB`

Question II.35
Le résultat sur la signature est une conséquence immédiate des questions II-31 et II-32.
La fonction `combine` prend en argument deux arbres binomiaux en tas de même ordre k et renvoie l'arbre binomial en tas d'ordre $k+1$ qui correspond à la construction du 3.2.
On écrit alors la fonction `ajouteAvecRetenue` qu'il faut rapprocher de l'incrémenteur 1-bit, et on obtient le programme 11, page 10.

Question II.36
Le résultat sur la signature est une conséquence immédiate des questions II-31 et II-32.
On écrit cette fois `fusionAvecRetenue` en s'inspirant de ce qu'on avait fait pour l'additionneur, et on obtient le programme 12, page 10.

```
71  let combine a b = match (a,b) with
72    | Noeud(x,filsA), Noeud(y,filsB) ->
73        if x > y then Noeud(y, a :: filsB)
74        else Noeud (x, b :: filsA)
75    | _ -> failwith "erreur" ;;

76  let rec ajouteAvecRetenue r = function
77    | [] -> [ r ]
78    | Vide :: q -> r :: q
79    | a :: q -> let r' = combine a r in Vide :: (ajouteRetenue r' q) ;;

80  let insertion x t =
81    let a = Noeud(x,[]) in ajouteRetenue a t ;;
```

Programme 11 la fonction `insertion`

```
82  let fusion t1 t2 =
83    let rec fusionAvecRetenue r t1 t2 = match (t1,t2) with
84      | [],_ -> ajouteAvecRetenue r t2
85      | _,[] -> ajouteAvecRetenue r t1
86      | Vide :: q1, Vide :: q2 -> r :: (fusion q1 q2)
87      | Vide :: q1, a2 :: q2 ->
88         if r = Vide then a2 :: (fusionAvecRetenue Vide q1 q2)
89         else Vide :: (fusionAvecRetenue (combine r a2) q1 q2)
90      | a1 :: q1, Vide :: q2 ->
91         if r = Vide then a1 :: (fusionAvecRetenue Vide q1 q2)
92         else Vide :: (fusionAvecRetenue (combine r a1) q1 q2)
93      | a1 :: q1, a2 :: q2 -> r :: (fusionAvecRetenue (combine a1 a2) q1 q2)
94    in
95      fusionAvecRetenue Vide t1 t2 ;;
```

Programme 12 la fonction `fusion`