

Machines de Turing

Exercice 1. Ecrire le programme qui additionne 1 à toute configuration binaire en remplaçant la séquence initiale par le résultat.

Correction. *L'idée est de lire la séquence initiale de gauche à droite, puis à partir du dernier bit de poids faible) de revenir vers la gauche en changeant tous les 1 en 0 (propagation de la retenue) jusqu'au premier 0 rencontré qu'on change en 1 avant de s'arrêter. Si on revient ainsi au début de la séquence au bit de poids fort il peut être nécessaire d'écrire un 1 à gauche de la séquence (propagation de la retenue) avant de s'arrêter. Par exemple :*

$$\begin{array}{r} 1011 \\ + \quad 1 \\ \hline 1100 \end{array}$$

On voit que l'obtention du résultat consiste à inverser les 1 et les 0 de droite à gauche jusqu'au troisième bit (premier 0 rencontré en allant de droite à gauche). Le programme est le suivant :

$$\begin{array}{ll} q_01 \rightarrow q_01D & : \text{lecture du nombre} \\ q_00 \rightarrow q_00D & : \text{de gauche à droite} \\ q_0b \rightarrow q_1bG & \\ q_10 \rightarrow q_F1 & : \text{le bit de poids faible vaut 0} \\ q_11 \rightarrow q_10G & : \text{propagation de la retenue} \\ q_1b \rightarrow q_Fb & \end{array}$$

Exercice 2. Ecrire le programme qui effectue le complément à deux d'une configuration binaire en remplaçant la séquence initiale par le résultat.

Correction. *Puisque le complément à deux d'une configuration binaire s'exprime comme l'ajout de 1 au complément à un de cette configuration, il suffit d'appliquer le programme écrit dans l'exercice précédent au complément à un de la configuration d'entrée. Pour calculer le complément à un d'un nombre, on commute les 1 et les 0 de ce nombre ; or cette opération peut être effectuée dès la lecture du nombre de gauche à droite par la tête de lecture-écriture.*

Autrement dit, il suffit de modifier légèrement les deux premières règles du programme de l'exercice précédent pour obtenir le programme recherché :

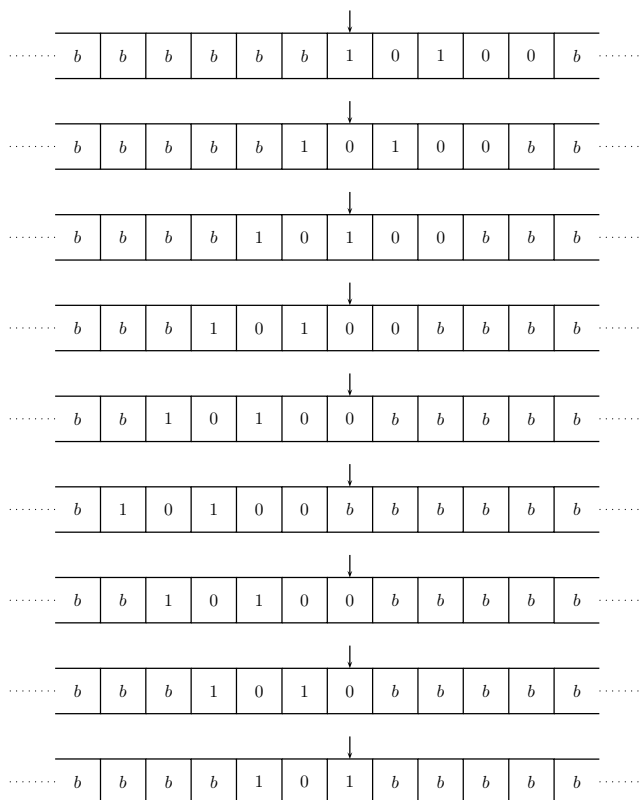
$$\begin{array}{ll} q_01 \rightarrow q_00D & : \text{lecture du nombre et inversion} \\ q_00 \rightarrow q_01D & : \text{de gauche à droite} \\ q_0b \rightarrow q_1bG & \\ q_10 \rightarrow q_F1 & : \text{le bit de poids faible vaut 0} \\ q_11 \rightarrow q_10G & : \text{propagation de la retenue} \\ q_1b \rightarrow q_Fb & \end{array}$$

Exercice 3. Soit $\Gamma = \{0, 1\}, \Sigma = \{0, 1\}, Q = \{q_0, q_1, q_2, q_3, q_Y, q_N\}$. Que fait le programme suivant :

$q_00 \rightarrow q_00D$
 $q_01 \rightarrow q_01D$
 $q_0b \rightarrow q_1bG$
 $q_10 \rightarrow q_2bG$
 $q_11 \rightarrow q_3bG$
 $q_1b \rightarrow q_NbG$
 $q_20 \rightarrow q_YbG$
 $q_21 \rightarrow q_NbG$
 $q_2b \rightarrow q_NbG$
 $q_30 \rightarrow q_NbG$
 $q_31 \rightarrow q_NbG$
 $q_3b \rightarrow q_NbG$

Remarque : 0, 1, représentant les symboles du calcul binaire, q_Y est un état final d'acceptation, q_N , un état final de rejet. Conseil : Tester sur les séquences 10100 et 110.

Correction. Trace sur 10100 :



Ce programme teste la divisibilité par 4 d'une configuration binaire et renvoie comme séquence de sortie le reste de cette division. 10100 est divisible par 4 : la séquence se termine par deux 0, l'exécution du programme s'arrête dans l'état q_Y . En outre, le programme efface les deux derniers zéros de la séquence d'entrée, donnant ainsi le reste de la division. 110 n'est pas divisible par 4 : l'exécution du programme s'arrête dans l'état q_N .

Exercice 4. Ecrire un programme qui duplique une séquence de "1". Donner sa représentation sous forme de graphe d'états finis. Conseil :

1. Commencer par un programme qui duplique un seul "1".
2. Passer au cas général : si nécessaire, penser à la possibilité de se munir d'un caractère supplémentaire sur la bande.

Correction. Le programme qui duplique un seul 1 est élémentaire a priori :

$$\begin{aligned} q_01 &\rightarrow q_01D \\ q_0b &\rightarrow q_1bD \\ q_1b &\rightarrow q_F1 \end{aligned}$$

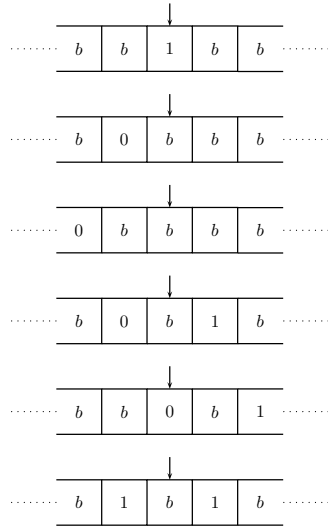
La copie de plusieurs 1 apparaît plus difficile. . . Une première approche, naive, consiste à essayer de compter les 1 de la séquence initiale pour en faire une copie de même longueur. On peut ainsi envisager de copier n fois le caractère 1 avec une machine à n états : chaque état sert à compter un 1 de la séquence : l'état q_0 permet de compter le premier 1, l'état q_1 , le second 1 et ainsi de suite jusqu'au n -ième 1. Mais d'une part, cela revient à écrire un programme par séquence de 1 de longueur différente ce qui est peu élégant, et d'autre part, par hypothèse, le nombre d'états ne doit pas croître indéfiniment. On doit donc trouver une méthode pour copier une suite sans avoir à compter ses éléments.

L'idée est de placer un marqueur qui parcourt la séquence initiale de gauche à droite pour mémoriser l'emplacement de la case en cours de duplication sur le ruban. Le marqueur en question est, par exemple, le caractère * qui va prendre successivement la place de chaque 1 dans la séquence initiale. A chaque pas du marqueur d'une case à la case suivante, les instructions disent à la tête de lecture-écriture d'aller à la dernière case de la séquence, de sauter la case qui contient le caractère "b" de délimitation, de sauter les caractères 1 déjà recopiés, puis de transformer le prochain "b" rencontré en 1. Si la tête de lecture-écriture effectue cette opération autant de fois que le marqueur avance d'une case, il en résulte une nouvelle suite de n 1 écrite immédiatement après la première, et séparée de celle-ci par le caractère blanc ("b").

1. Un seul caractère 1 en entrée :

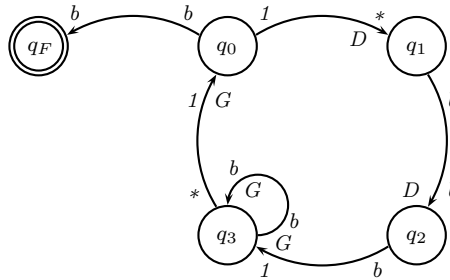
$$\begin{aligned} q_01 &\rightarrow q_1 * D \\ q_1b &\rightarrow q_2bD \\ q_2b &\rightarrow q_31G \\ q_3b &\rightarrow q_3bG \\ q_3* &\rightarrow q_01D \\ q_0b &\rightarrow q_Fb \end{aligned}$$

On obtient la trace suivante :



- *Etape 1* : la tête lit le caractère 1, le remplace par 0 (marqueur), et se déplace vers la droite.
- *Etape 2* : la tête lit le caractère "b" et se déplace vers la droite.
- *Etape 3* : la tête lit le caractère "b", le remplace par 1, et se déplace vers la gauche.
- *Etape 4* : la tête lit le caractère "b" et se déplace vers la droite.
- *Etape 5* : la tête lit le caractère 0 (marqueur), le remplace par 1, et se déplace vers la gauche.
- *Etape 6* : la tête lit le caractère "b" et s'arrête.

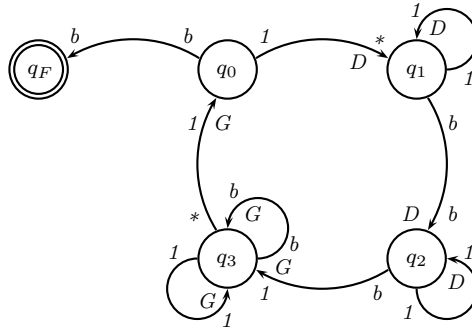
Le graphe d'états finis correspondant au programme est :



2. Cas général :

- $q_0 1 \rightarrow q_1 * D$
- $q_1 1 \rightarrow q_1 1 D$: nouvelle règle
- $q_1 b \rightarrow q_2 b D$
- $q_2 b \rightarrow q_3 1 G$
- $q_2 1 \rightarrow q_2 1 D$: nouvelle règle
- $q_3 1 \rightarrow q_3 1 G$: nouvelle règle
- $q_3 b \rightarrow q_3 b G$
- $q_3 * \rightarrow q_0 1 D$
- $q_0 b \rightarrow q_F b$

Le graphe d'états finis complété est :



Exercice 5. (optionnel) A partir du programme de réplication conçu dans l'exercice , concevoir un programme qui effectue la multiplication d'une séquence de 1 par une autre.

Correction. Soit à multiplier 111 par 11, on va répliquer trois fois la séquence 11 pour obtenir 111111. On lit donc la première séquence et on réplique la seconde en faisant appel au "sous-programme" vu dans l'exercice précédent. La lecture de la première séquence se fait en utilisant un marqueur (par exemple "#"¹) qu'on déplace au fur et à mesure de gauche à droite avec, à chaque déplacement du marqueur, duplication de la seconde séquence. On peut commencer par traiter le cas simple de la multiplication de la séquence 1 par 11. Le programme complet est écrit de la façon suivante : on conserve les états du programme de l'exercice précédent, et on ajoute de nouveaux états q' (par exemple l'état de départ est maintenant q'_0).

$q'_0 1$	\rightarrow	$q'_1 \# D$: initialisation du marqueur
$q'_1 1$	\rightarrow	$q'_1 1 D$: lecture de la 1ère séquence
$q'_1 b$	\rightarrow	$q'_2 b D$	
$q'_2 1$	\rightarrow	$q_0 1$: passage au sous-programme de réplication
		\vdots	
$q_0 b$	\rightarrow	$q_F b G$: retour du sous-programme de réplication
$q_F 1$	\rightarrow	$q_F 1 G$: q_F n'est plus un état final
$q_F \#$	\rightarrow	$q'_0 \#$: déplacement du marqueur
$q'_0 \#$	\rightarrow	$q'_0 1 D$	
$q'_0 b$	\rightarrow	q'_F	: nouvel état final

Machines RAM

Exercice 6. Pour chacune des instructions C++ suivantes, écrire un programme qui la simule sur la machine RAM :

1. $A = 1 ;$
2. $A = A + 10 ;$

¹On peut aussi choisir de réutiliser le marqueur * dans la mesure où il n'y a pas ici de risque de confusion avec son utilisation dans le sous-programme.

3. $A = A + B$;

Remarque : les variables A et B sont supposées rangées respectivement aux adresses 0 et 1 de la mémoire. Le contenu de la variable B n'est pas supposé être détruit après exécution de l'instruction $A = A + B$;

Correction. *Les programmes qui simulent les instructions C++ données sont respectivement :*

1. Pour $A = 1$; :

```
LOAD# 1 ;
STORE@ 0 ;
```

2. Pour $A = A + 10$; : il s'agit d'incrémenter dix fois l'accumulateur avant de ranger enfin la valeur obtenue à l'adresse 0 de la mémoire ; il est donc nécessaire de se munir d'un compteur, qu'on range (par exemple) à l'adresse 2 de la mémoire, et qui sera décrémenté d'autant de fois que l'accumulateur sera incrémenté dans une boucle pour obtenir la valeur finale de A.

```
LOAD# 10 ;
STORE@ 2 ;
ADD : LOAD@ 0 ;
      INCR ;
      STORE@ 0 ;
      LOAD@ 2 ;
      DECR ;
      JZ FIN ;
      STORE@ 2 ;
      JUMP ADD ;
FIN :  HALT ;
```

En remarquant qu'il est possible de factoriser l'instruction STORE@ 2 ; présente simultanément avant l'entrée et en fin de boucle, on obtient un programme légèrement plus court (mais pas plus rapide en exécution) :

```
LOAD# 10 ;
ADD : STORE@ 2 ;
      LOAD# 1 ;
      INCR ;
      STORE@ 0 ;
      LOAD@ 2 ;
      DECR ;
      JZ FIN ;
      JUMP ADD ;
FIN :  HALT ;
```

3. Pour $A = A + B$; : On procède comme pour le programme précédent, à ceci près qu'on incrémente à partir de A, et que c'est la valeur de B qui sert de valeur initiale à un compteur indépendant (nécessaire pour ne pas détruire la donnée B), rangé à l'adresse 2 de la mémoire. Il est en outre nécessaire de tester si B contient la valeur 0 avant d'entrer dans la boucle.

```

        LOAD@ 1 ;
        JZ FIN ;
ADD :   STORE@ 2 ;
        LOAD@ 0 ;
        INCR ;
        STORE@ 0 ;
        LOAD@ 2 ;
        DECR ;
        JZ FIN ;
        JUMP ADD ;
FIN :   HALT ;

```

Exercice 7. Ecrire un programme qui étant donné un entier rangé dans la mémoire à l'adresse 0, calcule sa valeur absolue et la range à la même adresse.

Correction. *La question à poser au début du programme est de savoir si l'entier rangé à l'adresse 0 est supérieur ou égal à zéro : si c'est le cas, le programme s'arrêtera immédiatement ; sinon, et puisqu'il s'agit d'un entier inférieur à zéro, on se munit d'un compteur initialisé à zéro et incrémenté simultanément au contenu de l'adresse 0 jusqu'à ce que celui-ci atteigne la valeur zéro. Le compteur contient alors la valeur absolue de l'entier initial et il suffit de recopier sa valeur à l'adresse 0. Il y a toutefois une difficulté technique supplémentaire, due au fait que le jeu d'instructions dont on dispose ne permet pas de tester si un entier est supérieur ou égal à zéro (ou, dualement, strictement inférieur à zéro), mais seulement s'il est inférieur ou égal à zéro. Pour cette raison, il est nécessaire d'incrémenter le compteur au delà de zéro (c'est à dire jusqu'à un) et il faudra donc le décrémenter une fois avant de ranger sa valeur à l'adresse 0.*

```

        LOAD# 0 ;
        STORE@ 1 ;
        LOAD@ 0 ;
        JZ INFEQZ ;
FIN :   HALT ;
INFEQZ : LOAD@ 1 ;
        INCR ;
        STORE@ 1 ;
        LOAD@ 0 ;
        INCR ;
        STORE@ 0 ;
        JZ INFEQZ ;
        LOAD@ 1 ;
        DECR ;
        STORE@ 0 ;
        JUMP FIN ;

```