Corrigé TP9

Automates

[a] En lisant la fonction d'initialisation de l'automate `init_par_defaut`, on voit que pour chaque état, on paramètre une transition retournant au même état, quelle que soit l'entrée. Au niveau de l'instruction :

```
A->transitions[i][j]=i;
```

En effet, la variable `A` de type `automate` contient la table de transition qui à chaque couple état/entrée (ligne/colonne) fait correspondre un nouvel état. Dans l'instruction ci dessus, l'indice `i` parcourt tous les numéros d'état et l'indice `j` tous les numéros d'entrée. Et dans la case à la ligne `i` et à la colonne `j`, on place le numéro d'état `i` correspondant donc à l'état d'où l'on part.

[b] En lisant la définition du type `automate` :

```
typedef struct {
  int nb_etats;
  int etat_initial;
  int etats_finals[NB_MAX_ETATS];
  int transitions[NB_MAX_ETATS][NB_MAX_ENTREES];
  char sortie[NB_MAX_ETATS][NB_MAX_ENTREES][LG_MAX_SORTIE];
} automate;
```

On voit que la table de transition contient `NB_MAX_ETATS` lignes et `NB_MAX_ENTREES` colonnes. Ces deux constantes valent 128, ce qui correspond au nombre de caractères de la table ASCII. L'ensemble d'entrée de l'automate pourra donc potentiellement être **l'ensemble des caractères ASCII**.

Mais en observant la fonction `init_mon_automate`, on voit que seuls 2 états sont utilisés ainsi que 3 entrées : les caractères '2', 'r' et 'c'. Toutes les transitions utilisant d'autres caractères mènent à une sortie `"entree_invalide"`. La conclusion est que notre implémentation permet de gérer n'importe quel automate acceptant des caractères ASCII, et que l'automate qui actuellement paramètré a l'ensemble d'entrée : `{2,r,c}`.

Cet ensemble d'entrée correspond évidemment aux actions : *"mettre une pièce de 20 centimes"*, *"appuyer sur le bouton café"* et *"appuyer sur le bouton rendre monnaie"* que l'on pourrait effectuer sur une machine à café.

[c] Lorsque l'on saisi des caractères non prévus, comme expliqué ci-dessus, on reste dans le même état et le message "entree_invalide" s'affiche.

[d] Voici une implémentation possible pour Café1.

On commence bien sûr par rédiger rapidement un Makefile (puisqu'on a l'habitude et qu'on est très à l'aise avec les Makefiles).

```
automate: automate.o main.o
    clang automate.o main.o -o automate

automate.o: automate.c automate.h
        clang -c automate.c

main.o: main.c automate.h
        clang -c main.c
```

On complète ensuite la fonction `simule_automate` comme suit :

```
void simule_automate(automate *A) {
   int etat_courant, etat_suivant;
   int symbole_entree= ' ';

   etat_courant = A->etat_initial;
   while (1) {
      /* lire une entree */
      lire_entree (&symbole_entree);
      if (symbole_entree == 'q') {
         break;
      }

      /* calculer l'état suivant */
      etat_suivant = A->transitions[etat_courant][symbole_entree];

      /* ecrire le message de sortie */
      ecrire_sortie (A->sortie[etat_courant][symbole_entree]);

      /* mettre à jour l'état courant */
      etat_courant = etat_suivant;
   }
}
```

[e] Nous passons maintenant à l'implémentation de Cafe2, qui va lire l'automate depuis un fichier avant de le simuler. On commence par écrire la fonction `lecture_automate` qui interprète le fichier `Mon_automate.auto`.

Le format du fichier est bien décrit dans le sujet du TP. Vous savez manipuler la fonction `fscanf` pour lire tous les entiers et les caractères requis, voilà donc sans plus attendre une implémentation possible :

```
void lecture_automate(automate *A, FILE *f) {
   init_par_defaut(A);

   /* Nombre d'états */
   fscanf (f, "%d", &A->nb_etats);

   /* États finals */
   int nb_etat_finals = 0;
   fscanf (f, "%d", &nb_etat_finals);

   for (int i=0; i<nb_etat_finals; i++) {
      int etat_final = 0;
      fscanf (f, "%d", &etat_final);
      A->etats_finals[etat_final] = 1;
   }

   /* Transitions */
   int nb_transitions = 0;
   fscanf (f, "%d", &nb_transitions);

   for (int i=0; i<nb_transitions; i++) {
      int etat, etat_suivant;
      char entree;
      fscanf (f, "%d %c %d", &etat, &entree, &etat_suivant);
      A->transitions[etat][(int)entree] = etat_suivant;
      A->sortie[etat][(int)entree][0] = '\0';
   }

   /* Sorties */
   int nb_sorties = 0;
   fscanf (f, "%d", &nb_sorties);

   for (int i=0; i<nb_sorties; i++) {
      int etat;
      char entree;
      fscanf (f, "%d %c ", &etat, &entree);

      /* Une fois l'etat et l'entree passés, on lit tous
       * les caractères jusqu'à la fin de la ligne */
      int i = 0;
      char cc;
```

```
        char sortie[LG_MAX_SORTIE];
        fscanf (f, "%c", &cc);
        while (!feof(f) && cc != '\n') {
            sortie[i] = cc;
            fscanf (f, "%c", &cc);
            i++;
        }
        sortie[i] = '\0';
        strcpy (A->sortie[etat][(int)entree], sortie);
    }
}
```

On note que l'on met une chaîne vide dans toutes les sorties correspondant à une transition lue, puis qu'on la reconfigure lorsqu'on lit la section des sorties. Ce n'est pas une erreur : on n'est en effet pas obligé de définir une sortie pour toutes les transitions que l'on a écrites. Une transition sans sortie doit donc logiquement afficher une chaîne vide. Sans cette assignation, la sortie d'une telle transition serait celle par defaut, c'est à dire **entree_invalide**.

[f] Voir listing ci-dessus pour `lecture_automate`. Voici également la fonction main :

```
#include <stdio.h>
#include <stdlib.h>
#include "automate.h"

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf ("USAGE: %s automate.auto\n", argv[0]);
        exit(1);
    }

    char *filename = argv[1];
    FILE *file = fopen (filename, "r");
    if (file == NULL) {
        printf ("Impossible d'ouvrir le fichier automate\n");
        perror (filename);
        exit(1);
    }

    automate A;
    lecture_automate(&A, file);
    simule_automate(&A);

    fclose (file);
    return 0;
}
```

Et le header modifié :

```
#ifndef __AUTOMATE__
#define __AUTOMATE__

#define NB_MAX_ETATS   128
#define NB_MAX_ENTREES 128
#define LG_MAX_SORTIE  128

typedef struct {
    int nb_etats;
    int etat_initial;
    int etats_finals[NB_MAX_ETATS];
    int transitions[NB_MAX_ETATS][NB_MAX_ENTREES];
    char sortie[NB_MAX_ETATS][NB_MAX_ENTREES][LG_MAX_SORTIE];
} automate;

void lecture_automate(automate *A, FILE *f);
void simule_automate(automate *A);

#endif
```

[g] Pour connaître la taille d'un type, on peut utiliser l'instruction `sizeof`. On passe en paramètre à cette dernière le nom du type que l'on souhaite, comme s'il s'agissait d'une fonction. Le résultat retourné est la taille qu'occuperait une variable du type choisi en octet. `sizeof(int)` par exemple, renvoit 4 sur mon ordinateur.

Attention, cela ne marche que pour les types statiques ! Il ne faut **surtout pas** avoir l'idée saugrenue de s'en servir pour chercher le nombre d'élément d'un tableau ou la longueur d'une chaîne de caractère !

Pour connaître la taille du type automate, on rajoute donc dans le main la ligne :

```
printf ("taille automate : %d\n", sizeof(automate));
```

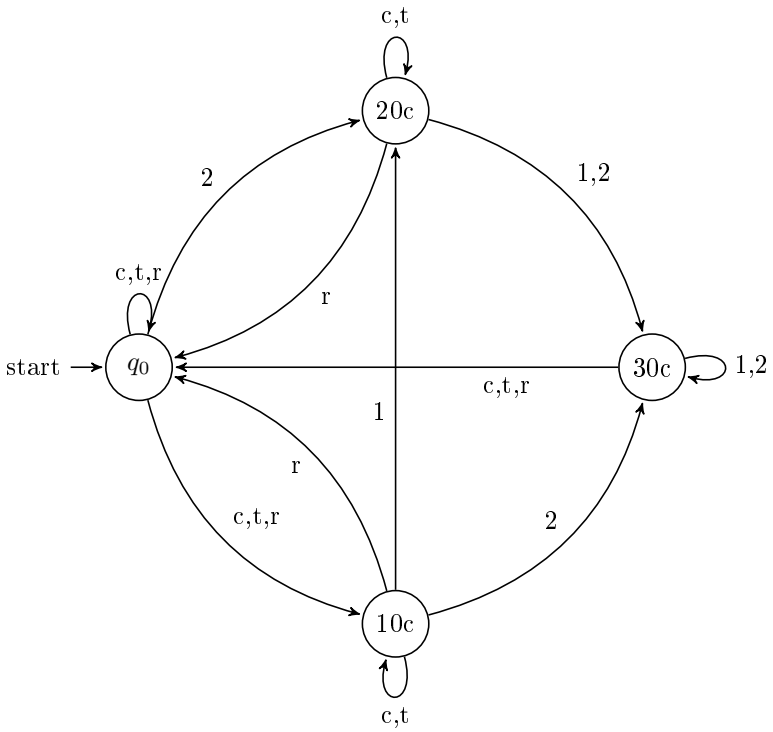
Et l'on voit que la taille de l'automate est de 2163208 octets. Soit environs **2 méga octets** !

On comprend vite pourquoi : pour être générique, notre implémentation permet de définir 128 états et 128 entrées. Il y a donc 128 * 128 = 16384 transitions possibles. C'est le nombre de case de notre tableau de transition, ce qui en fait déjà un grand tableau ...

Pire ! Notre implémentation permet de stocker un message de sortie de 128 caractères maximum pour **chaque** transition. Le tableau de sortie contient donc 128 * 128 * 128 = 2097152 octets, soit 2Mo à lui tout seul. C'est là que la taille de l'automate explose véritablement.

On comprends donc que des choix qui paraissent raisonables aux premiers abords (128 états possible, 128 entrées possibles et 128 caractères pour une sortie, ces choix pris individuellement semblent tout à fait acceptables) peuvent en fait conduire très rapidement à une forte consommation de mémoire. Prudence, donc ! D'autant que dans notre cas, lorsque l'on paramètre l'automate café1 dans notre programme, seuls 2 états, 3 entrées et 4 sorties sont réellement utilisées. Nos tableaux de transition et de sortie sont donc quasiment vides ! En conclusion, notre implémentation d'automate a le mérite d'être simple à comprendre et à utiliser, mais elle n'est **vraiment pas** optimale !

[h] Voilà un nouvel automate pour notre machine à café 2.0.



On a omis les message de sortie sur le schéma pour ne pas le rendre illisible. On remarque qu'il faut deux états intermédiaires supplémentaires pour gérer les pièces de 10 et 20 centimes. Sinon, l'automate ressemble fortement à la version précédente.

Voilà son fichier auto pour simuler l'automate avec notre programme :

```
4
0
20
0 1 2
0 2 1
0 c 0
0 t 0
0 r 0
1 1 3
1 2 3
1 c 1
1 t 1
1 r 0
2 1 1
2 2 3
2 c 2
2 t 2
2 r 0
3 1 3
3 2 3
3 c 0
3 t 0
3 r 0
19
0 1 credit:10c
0 2 credit:20c
0 c pas-assez-de-pognon-credit:0c
0 t pas-assez-de-pognon-credit:0c
1 1 credit:30c
1 2 CLING!-credit:30c
1 c pas-assez-de-pognon-credit:20c
1 t pas-assez-de-pognon-credit:20c
1 r CLING!
2 1 credit:20c
2 2 credit:30c
2 c pas-assez-de-pognon-credit:10c
2 t pas-assez-de-pognon-credit:10c
2 r CLING!
3 1 CLING!-credit:30c
3 2 CLING!-credit:30c
3 c Boisson_servie
3 t Boisson_servie
3 r CLING!
```

[i] Pas de spoil, on ne donnera pas ici la solution du mystère! Simplement l'état final de l'automate est l'état 2. On peut modifier la contidition du while dans la fonction `simule_automate` pour sortir de la simulation lorsqu'on arrive à un état final. Voilà une façon de faire :

```c
void simule_automate(automate *A) {
   int etat_courant, etat_suivant;
   int symbole_entree= ' ';

   etat_courant = A->etat_initial;
   while (!A->etats_finals[etat_courant]) {
      /* lire une entree */
      lire_entree (&symbole_entree);
      if (symbole_entree == 'q') {
         break;
      }

      /* calculer l'état suivant */
      etat_suivant = A->transitions[etat_courant][symbole_entree];

      /* ecrire le message de sortie */
      ecrire_sortie (A->sortie[etat_courant][symbole_entree]);

      /* mettre à jour l'état courant */
      etat_courant = etat_suivant;
   }
}
```

On se souvient que le tableau `A->etat_finals` contient une case pour chaque état. Si l'état 2 est final, alors la case 2 du tableau contient 1. Sinon, elle contient 0. Comme vous le savez, en C, 0 est évalué comme FAUX et 1 comme VRAI. Donc `A->etat_finals[i]` est VRAI si l'état `i` est final, et FAUX sinon. Dans la fonction, on simule tant que l'état courant n'est pas un état final. Donc tant que `A->etat_finals[etat_courant]` est FAUX.