Corrigé TP12

Développement d'un interpréteur de commandes (2/2)

[a] Ce n'est vraiment pas la peine de faire plus de scripts de test que ce qui est déjà fourni.

[b] On est en train de chercher où est appelé la fonction `decode_entree`. On pense immédiatement à la commande `grep` que nous avons vue au début du semestre. On se place donc à la racine de notre projet et on tape :

```
$ grep -Rn decode_entree
listing/commandes.c:81:int decode_entree(char *mot) {
listing/commandes.c:162:    entree = decode_entree(bout_de_ligne);
```

Grep nous renvoie donc, en couleur s'il vous plait, les deux endroit où il a trouver le nom de la fonction. Ligne 81 du fichier commandes.c, c'est la déclaration et ligne 162, c'est le seul appel pour l'instant dans le projet.

Voilà une implémentation possible de cette fonction :

```
int decode_entree(char *mot) {
   int mnemo = INSTRUCTION;

   for (int i=0; i<nombre_mnemoniques; i++) {
      if (strcmp (mot, mnemonique[i]) == 0) {
         mnemo = valeur_mnemonique[i];
         break;
      }
   }

    return mnemo;
}
```

[c] La difficulté de l'implémentation des fonctions manquantes de `commandes.c` est que vous n'êtes guidés qu'avec partimonie sur les endroits à compléter. Il faut donc bien comprendre l'objectif du programme pour savoir où l'on va.

Trois choses à compléter, donc. La fonction `decode_entree` a été vue à la question précédente.

La fonction `init_automate_commandes` consiste à remplir la table de transition de l'automate pour le faire correspondre à l'automate de gestion du if vu en TD. Ensuite, il faut également remplir la table de sortie pour savoir dans quels cas la ligne courante doit être exécutée. Enfin, il faut paramétrer l'état courant de l'automate à l'état initial. Cette façon de faire est un peu différente de celle dont on a l'habitude : l'état courant est à présent stocké directement dans la structure de l'automate.

Enfin, il faut compléter une partie de la fonction `analyse_commande_interne`. C'est à cet endroit que l'on va insérer l'algorithme usuel pour faire marcher l'automate, c'est à dire :

- Récupérer l'entrée
- Calculer l'état suivant en fonction de l'état courant et de l'entrée
- Calculer la sortie de la même manière
- Mettre à jour l'état courant avec l'état suivant

Voilà une implémentation possible des deux dernières fonctions.

```
void init_automate_commandes(automate * A) {
    int i, j;

    /* La majorite des transitions vont dans l'etat d'erreur et la sortie est
       majoritairement NON
     */
    for (i = 0; i < NB_MAX_ETATS; i++) {
        for (j = 0; j < NB_MAX_ENTREES; j++) {
            A->transitions[i][j] = ERREUR;
            A->sortie[i][j] = NON;
        }
    }

    A->transitions[NORMAL][INSTRUCTION]             = NORMAL;
    A->transitions[NORMAL][IF_VRAI]                 = ATTENTE_THEN_VRAI;
    A->transitions[NORMAL][IF_FAUX]                 = ATTENTE_THEN_FAUX;
    A->sortie     [NORMAL][INSTRUCTION]             = OUI;
    A->sortie     [NORMAL][IF_VRAI]                 = NON;
    A->sortie     [NORMAL][IF_FAUX]                 = NON;

    A->transitions[ATTENTE_THEN_VRAI][THEN]         = DANS_THEN;
    A->sortie     [ATTENTE_THEN_VRAI][THEN]         = OUI;

    A->transitions[DANS_THEN][INSTRUCTION]          = DANS_THEN;
    A->transitions[DANS_THEN][FI]                   = NORMAL;
    A->transitions[DANS_THEN][ELSE]                 = ATTENTE_FI;
    A->sortie     [DANS_THEN][INSTRUCTION]          = OUI;
    A->sortie     [DANS_THEN][FI]                   = NON;
    A->sortie     [DANS_THEN][ELSE]                 = NON;

    A->transitions[ATTENTE_FI][INSTRUCTION]         = ATTENTE_FI;
    A->transitions[ATTENTE_FI][FI]                  = NORMAL;
    A->sortie     [ATTENTE_FI][INSTRUCTION]         = NON;
    A->sortie     [ATTENTE_FI][FI]                  = NON;

    A->transitions[ATTENTE_THEN_FAUX][THEN]         = ATTENTE_ELSE_OU_FI;
    A->sortie     [ATTENTE_THEN_FAUX][THEN]         = NON;

    A->transitions[ATTENTE_ELSE_OU_FI][INSTRUCTION] = NORMAL;
    A->transitions[ATTENTE_ELSE_OU_FI][FI]          = NORMAL;
    A->transitions[ATTENTE_ELSE_OU_FI][ELSE]        = DANS_ELSE;
    A->sortie     [ATTENTE_ELSE_OU_FI][INSTRUCTION] = NON;
    A->sortie     [ATTENTE_ELSE_OU_FI][FI]          = NON;
    A->sortie     [ATTENTE_ELSE_OU_FI][ELSE]        = OUI;

    A->transitions[DANS_ELSE][INSTRUCTION]          = DANS_ELSE;
    A->transitions[DANS_ELSE][FI]                   = NORMAL;
    A->sortie     [DANS_ELSE][INSTRUCTION]          = OUI;
    A->sortie     [DANS_ELSE][FI]                   = NON;

    /**** etat initial ******************************/
    A->etat = NORMAL;
    A->nb_etats = 7;
    A->etat_initial = NORMAL;
    A->ligne_boucle = 0;
}

int analyse_commande_interne(automate * A, char *ligne_courante) {
    char bout_de_ligne[TAILLE_MAX_LIGNE];
    int entree;
    int i;
    /*
       Par defaut, on indiquera a la fonction appelante qu'elle doit
       executer la commande, on passera cela a NON si on rencontre une
       commande interne completement resolue ici
     */
    int code_sortie = OUI;

    debug("J'essaie de reconnaitre une commande interne\n");
    /*
       On extrait d'abord le premier mot de la ligne pour voir si c'est un
       mnemonique de commande interne
     */
    extrait_premier_mot(ligne_courante, bout_de_ligne);
```

```
    entree = decode_entree(bout_de_ligne);

    if (entree != INSTRUCTION) {
        /*
           Si on a reconnu un mnemonique, on l'affiche (debug)
         */
        debug("Commande interne reconnue : %s\n", bout_de_ligne);
        /*
           Pour pouvoir executer ce qui suit le mot cle, la ligne courante
           devient ce qui reste apres le premier mot :
           on ecrase simplement le debut de la chaine.
         */
        int len = strlen(bout_de_ligne);
        for (i=len; ligne_courante[i] != '\0'; i++)
            ligne_courante[i-len] = ligne_courante[i];
        ligne_courante[i-len] = '\0';
    }

    /*
       Dans le cas d'une structure conditionnelle, le code de sortie sera
       NON (puisqu'on execute la fin de ligne pour le test).
    */
    if (entree == IF) {
        entree = selectionne_alternative(ligne_courante, IF_VRAI, IF_FAUX);
        code_sortie = NON;
    }
/*************** A COMPLETER *****************/
    /*
       Si vous arrivez a l'etape du while, il faut commencer
       par un traitement similaire au cas du if : executer le
       reste de la ligne pour recuperer le resultat du test.
       Il faut egalement penser a memoriser la ligne courante
       pour pouvoir y revenir en fin de boucle.
     */
    /*
       Dans tous les cas, calculer code_sortie et changer
       d'etat en fonction de etat_courant et de entree.
     */
/*********************************************/

    int etat_suivant = A->transitions[A->etat][entree];
    code_sortie = A->sortie[A->etat][entree];
    A->etat = etat_suivant;

    /* prise en compte du code de sortie, la fonction doit retourner :
       - 0 si la ligne ne doit pas etre executee (commande interne geree ici)
       - 1 sinon
       Ce sont les valeurs choisies pour NON et OUI
     */
    debug ("Faut-il executer : %d\n", code_sortie);
    return code_sortie;
}
```

[d] On note que seules deux fonctions de `commandes.c` ont leur prototype déclaré dans `commandes.h`. C'est parce que les autres fonctions ne sont utilisées que dans `commandes.c`. Pas la peine de les exposer au monde entier. On a donc deux fonctions dont ont cherche les endroits où elles sont appelées dans le code. Pour ça, comme pour la première question, rien de tel qu'un bon petit grep !

— `analyse_commande_interne` est appelée dans `interpreteur.c`, ligne 54.

— `init_automate_commandes` est appelée dans `interpreteur.c`, ligne 98.

[e] Une fois testé notre interpréteur, à présent avec quelques scripts contenant des if/then/else, on passe à la suite, c'est à dire à l'historique.

Comme nous avons expliqué tout le principe de l'historique en TD et que le schéma récapitulatif dans le sujet de TP est bien fait, voici sans plus attendre une implémentation possible pour `lignes.c`.

Dans `lignes.h`, on rajoute :

```
void init_historique (FILE *f);
int lire_ligne (char *ligne);
void aller_a_la_ligne (int numero);
int obtenir_numero_ligne ();
```

Dans `lignes.c`, on rajoute :

```
char lignes_lues[NOMBRE_MAX_LIGNES][TAILLE_MAX_LIGNE];
int nombre_ligne_lues;
int position_courante;
FILE *fichier;

void init_historique (FILE *f) {
   nombre_ligne_lues = 0;
   position_courante = 0;
   fichier = f;
}

int lire_ligne (char *ligne) {
   int ret = 1;
   char buffer[TAILLE_MAX_LIGNE];
   if (position_courante < nombre_ligne_lues) {
      strcpy (ligne, lignes_lues[position_courante]);
   } else {
      if (nombre_ligne_lues > NOMBRE_MAX_LIGNES - 1) {
         debug ("Taille max historique atteinte\n");
         ligne[0] = '\0';
         return 0;
      }
      ret = lire_ligne_fichier (fichier, buffer);
      strcpy (ligne, buffer);
      strcpy (lignes_lues[nombre_ligne_lues], buffer);
      nombre_ligne_lues++;
   }
   position_courante++;
   return ret;
}

void aller_a_la_ligne (int numero) {
   position_courante = numero;
}

int obtenir_numero_ligne () {
   return position_courante;
}
```

On voit que le mécanisme de l'historique, qui n'est pas si facile à comprendre, se plie en fait en quelques fonctions assez courtes.

Reste à aborder le problème **PRINCIPAL** de cette question. Les fonctions sont maintenant faites. Fastoche. Et en plus, ça compile ! Trop easy ! Mais comment faire pour tester ? En effet, nous venons d'implémenter l'historique alors que nous n'avons pas encore intégrer la gestion du while dans notre interpréteur. Comment donc vérifier que nos fonctions sont justes alors qu'elles ne sont pour l'instant appellées nulle part dans le programme ? Eh bien, pas de miracle, il faut faire des tests unitaires, c'est à dire un main spécial dont le but est uniquement d'appeler nos fonctions dans tous les cas possibles.

On rajoute au `Makefile` les lignes suivantes :

```
test: lignes.o test_historique.o
   clang $^ -o $@

test_historique.o: test_historique.c lignes.h
   clang -Wall -Werror -g -DDEBUG -c $^
```

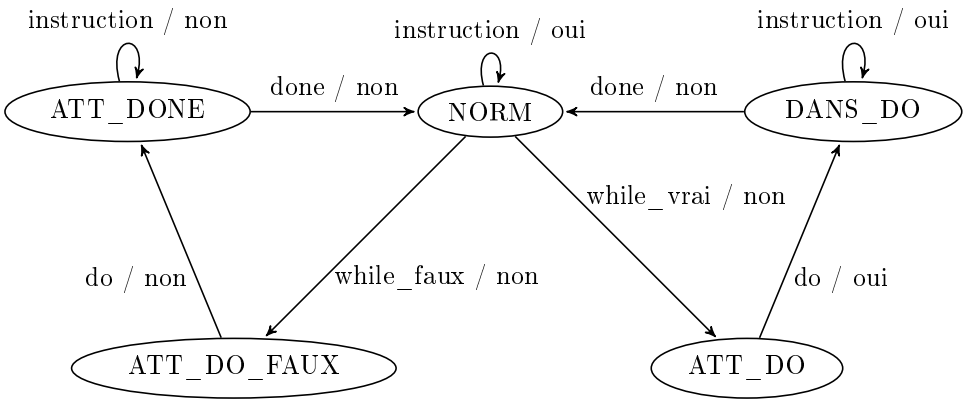
Et on crée le fichier `test_historique.c` avec le contenu suivant :

```c
#include <stdio.h>
#include <string.h>

#include "lignes.h"

Historique histo;

void test_simple () {
   FILE *f = fopen ("utest_histo.txt", "r");
   char devnull[TAILLE_MAX_LIGNE];
   int ret = 1;

   Historique *histo_ptr = &histo;
   init_historique (histo_ptr, f);

   lire_ligne (histo_ptr, devnull);
   lire_ligne (histo_ptr, devnull);
   lire_ligne (histo_ptr, devnull);
   lire_ligne (histo_ptr, devnull);
   lire_ligne (histo_ptr, devnull);
   lire_ligne (histo_ptr, devnull);
   lire_ligne (histo_ptr, devnull);
   ret = lire_ligne (histo_ptr, devnull);

   if (ret == 1) {
      printf ("[FAIL] 7 lignes lues et pas à la fin du fichier !\n");
   } else {
      printf ("[ OK ] après 7 lectures, fin du fichier\n");
   }

   fclose (f);
}

void test_retour () {
   FILE *f = fopen ("utest_histo.txt", "r");
   char buffer1[TAILLE_MAX_LIGNE];
   char buffer2[TAILLE_MAX_LIGNE];

   Historique *histo_ptr = &histo;
   init_historique (histo_ptr, f);

   lire_ligne (histo_ptr, buffer1);
   aller_a_la_ligne (histo_ptr, 0);
   lire_ligne (histo_ptr, buffer2);

   if (strcmp(buffer1, buffer2) != 0) {
      printf ("[FAIL] ligne 0 lue deux fois et non identique : <%s> != <%s>\n", buffer1,
          buffer2);
   } else {
      printf ("[ OK ] Deux lectures de la même ligne donnent résultat identiques\n");
   }

   fclose (f);
}

int main () {

   test_simple ();
   test_retour ();

   return 0;
}
```

On peut maintenant compiler avec la commande `make test` et exécuter `test` pour vérifier que l'historique marche bien. Et tout ça dans la bonne humeur car on n'oublie pas que :

TESTER N'EST JAMAIS DU TEMPS PERDU

[f] Voilà le schéma de l'automate pour le while, qu'il faudra incorporer dans l'automate principal de notre interpréteur.



[g] L'implémentation du while est donc une sorte de courronnement de votre année. Commençons par prêter un oeil à tout ce qu'il n'y a PAS besoin de faire.

- Les constantes ATTENTE_DO_VRAI, ATTENTE_DO_FAUX, DANS_DO, ATTENTE_DONE sont déjà présentes pour représenter les états de l'automate du while.
- Les constantes WHILE, DO, DONE, WHILE_VRAI, WHILE_FAUX sont déjà présentes pour représenter les entrées de l'automate. On fait attention : comme pour le IF, l'entrée WHILE est temporaire, elle doit être "résolue" en WHILE_VRAI ou WHILE_FAUX selon la condition qui suit.
- Le tableau des mnémoniques contient déjà les correspondances pour le while. La fonction `decode_entree` n'a donc pas besoin d'être modifiée.
- `selectionne_alternative` et `extrait_premier_mot` n'ont pas besoin d'être modifiées, on n'y touchera pas du TP.

Voilà le boulot à effectuer :

- Incorporer les transitions de l'automate du while dans notre automate qui ne gère pour l'instant que le if. Cela consiste donc à modifier la fonction `init_automate`.
- Modifier `analyse_commande_interne` pour effectuer la "résolution" du mnémonique while en WHILE_VRAI ou WHILE_FAUX. Juste en dessous du if qui appelle `selectionne_alternative`.
- Appeler la fonction `obtenir_numero_ligne` losrque l'on détecte un while, qu'il soit vrai ou faux, pour connaître son numéro de ligne. Puis, enregistrer ce numéro de ligne dans l'automate, au registre `A->ligne_boucle`.
- Appeler la fonction `aller_a_la_ligne` lorsque la sortie de l'automate est BOUCLE, afin de revenir à la ligne du while, qu'on ira chercher dans l'automate, puisqu'on l'a enregistrée précédemment.
- Ne pas oublier d'initialiser l'historique avec le descripteur de fichier correspondant au script dans `interpreteur.c`, à l'aide de la fonction `init_historique`.

En un mot comme en cent, voilà les nouvelles implémentations pour `init_automate_commandes`.

```
void init_automate_commandes(automate * A) {
    int i, j;

    /* La majorite des transitions vont dans l'etat d'erreur et la sortie est
       majoritairement NON
     */
    for (i = 0; i < NB_MAX_ETATS; i++) {
        for (j = 0; j < NB_MAX_ENTREES; j++) {
            A->transitions[i][j] = ERREUR;
            A->sortie[i][j] = NON;
        }
    }

    A->transitions[NORMAL][INSTRUCTION]             = NORMAL;
    A->transitions[NORMAL][IF_VRAI]                 = ATTENTE_THEN_VRAI;
    A->transitions[NORMAL][IF_FAUX]                 = ATTENTE_THEN_FAUX;
    A->sortie     [NORMAL][INSTRUCTION]             = OUI;
```

```
    A->transitions[ATTENTE_THEN_VRAI][THEN]          = DANS_THEN;
    A->sortie      [ATTENTE_THEN_VRAI][THEN]          = OUI;

    A->transitions[DANS_THEN][INSTRUCTION]           = DANS_THEN;
    A->transitions[DANS_THEN][FI]                    = NORMAL;
    A->transitions[DANS_THEN][ELSE]                  = ATTENTE_FI;
    A->sortie      [DANS_THEN][INSTRUCTION]           = OUI;

    A->transitions[ATTENTE_FI][INSTRUCTION]          = ATTENTE_FI;
    A->transitions[ATTENTE_FI][FI]                   = NORMAL;

    A->transitions[ATTENTE_THEN_FAUX][THEN]          = ATTENTE_ELSE_OU_FI;

    A->transitions[ATTENTE_ELSE_OU_FI][INSTRUCTION] = NORMAL;
    A->transitions[ATTENTE_ELSE_OU_FI][FI]           = NORMAL;
    A->transitions[ATTENTE_ELSE_OU_FI][ELSE]         = DANS_ELSE;
    A->sortie      [ATTENTE_ELSE_OU_FI][ELSE]         = OUI;

    A->transitions[DANS_ELSE][INSTRUCTION]           = DANS_ELSE;
    A->transitions[DANS_ELSE][FI]                    = NORMAL;
    A->sortie      [DANS_ELSE][INSTRUCTION]           = OUI;

    /***************** Rajout pour le while **********************/
    A->transitions [NORMAL][WHILE_VRAI]   = ATTENTE_DO_VRAI;
    A->transitions [ATTENTE_DO_VRAI][DO]  = DANS_DO;
    A->transitions [DANS_DO][INSTRUCTION] = DANS_DO;
    A->transitions [DANS_DO][DONE]        = NORMAL;
    A->transitions [NORMAL][WHILE_FAUX]   = ATTENTE_DO_FAUX;
    A->transitions [ATTENTE_DO_FAUX][DO]  = ATTENTE_DONE;
    A->transitions [ATTENTE_DONE][DONE]   = NORMAL;

    A->sortie       [ATTENTE_DO_VRAI][DO]  = OUI;
    A->sortie       [DANS_DO][INSTRUCTION] = OUI;
    A->sortie       [DANS_DO][DONE]        = BOUCLE;
    /*************************************************************/

    A->etat = NORMAL;
    A->nb_etats = 7;
    A->etat_initial = NORMAL;
    A->ligne_boucle = 0;
}
```

Et la fonction analyse_commande_interne.

```
int analyse_commande_interne(automate * A, char *ligne_courante) {
    char bout_de_ligne[TAILLE_MAX_LIGNE];
    int entree;
    int i;
    /*
       Par defaut, on indiquera a la fonction appelante qu'elle doit
       executer la commande, on passera cela a NON si on rencontre une
       commande interne completement resolue ici
     */
    int code_sortie = OUI;

    debug("J'essaie de reconnaitre une commande interne\n");
    /*
       On extrait d'abord le premier mot de la ligne pour voir si c'est un
       mnemonique de commande interne
     */
    extrait_premier_mot(ligne_courante, bout_de_ligne);
    entree = decode_entree(bout_de_ligne);

    if (entree != INSTRUCTION) {
        /*
           Si on a reconnu un mnemonique, on l'affiche (debug)
         */
        debug("Commande interne reconnue : %s\n", bout_de_ligne);
        /*
           Pour pouvoir exécuter ce qui suit le mot clé, la ligne courante
           devient ce qui reste apres le premier mot :
           on ecrase simplement le debut de la chaine.
         */
        int len = strlen(bout_de_ligne);
        for (i=len; ligne_courante[i] != '\0'; i++)
```

```
            ligne_courante[i-len] = ligne_courante[i];
        ligne_courante[i-len] = '\0';
    }

    /*
       Dans le cas d'une structure conditionnelle, le code de sortie sera
       NON (puisqu'on execute la fin de ligne pour le test).
    */
    if (entree == IF) {
        entree = selectionne_alternative(ligne_courante, IF_VRAI, IF_FAUX);
        code_sortie = NON;
    }

    /***************** Rajout pour le while *********************/
    if (entree == WHILE) {
        entree = selectionne_alternative(ligne_courante, WHILE_VRAI, WHILE_FAUX);
        code_sortie = NON;
        A->ligne_boucle = obtenir_numero_ligne ();
    }
    /************************************************************/

    int etat_suivant = A->transitions[A->etat][entree];
    code_sortie = A->sortie[A->etat][entree];
    A->etat = etat_suivant;

    /***************** Rajout pour le while *********************/
    if (code_sortie == BOUCLE) {
       aller_a_la_ligne (A->ligne_boucle);
    }
    /************************************************************/

    debug ("Faut-il executer : %d\n", code_sortie);
    return code_sortie;
}
```