# Corrigé TP11

## Développement d'un interpréteur de commandes (1/2)

Ce TP est sans doute le plus difficile de l'année et celui qui fera appel au champ le plus large de vos connaissances. Il est aussi le plus intéressant et le plus complet :

- Il nécessite de lire et de comprendre du code déjà écrit
- Il nécessite d'utiliser des fonctions sans s'attarder sur leur implémentation.
- Il vous demande de vous concentrer durant deux séances sur le même projet. Ce n'est donc pas un projet jetable, "code and forget" comme tous les TPs précédents.

Il n'existe pas de méthode absolue pour découvrir et comprendre, *explorer* une base de code existante. Vous allez ainsi rencontrer des problèmes de la vie réelle :

- Trouver où, mais **OÙ** cette punaise de fonction est déclarée
- Comprendre les noms obscurs des variables et leur signification
- Voir que l'indentation aide **VRAIMENT** à lire du code
- Voir qu'un Makefile est **INDISPENSABLE** lorsqu'on aborde un projet que l'on ne connait pas
- etc.

Tout cela fait partie du boulot d'un informaticien. N'oubliez pas qu'un développeur passe plus de temps à lire et explorer du code existant qu'à en écrire. Et à ce petit jeu, le gagnant et souvent celui qui a les meilleurs outils (et qui sait s'en servir).

**[a]** On jette un oeil aux fichier `lignes.c` et `variables.c` et on découvre avec gravité que toutes les fonctions sont vides. Il va falloir les compléter ...

**[b]** Une implémentation possible, dans le fichier `lignes.c`, de la fonction `lire_ligne_fichier` :

```c
int lire_ligne_fichier(FILE * fichier, char *ligne) {
   char cc = 0;
   int i = 0;
   fscanf (fichier, "%c", &cc);
   while (!feof (fichier) && cc != '\n') {
      ligne[i] = cc;
      i++;
      fscanf (fichier, "%c", &cc);
   }
   ligne[i] = '\0';
   return (cc == '\n');
}
```

**[c]** Après avoir complété `lire_ligne_fichier`, on compile en utilisant `make`, puis on teste notre interpreteur sur le fichier de commande `test_1_lecture_ligne.sh`.

On voit que l'interpreteur s'exécute bien sur chacune des lignes, mais qu'il n'exécute que la commande de base, sans prendre en compte les paramètres.

**[d]** Voici une implémentation possible pour les fonctions de `variables_base.h` :

```c
char * const_empty = "";

void init_variables(variables * ens) {
   ens->nb = 0;
}

int ajouter_variable(variables * ens, char *nom, char *valeur) {
   int index = -1;

   for (int i=0; i<ens->nb; i++) {
      if (strcmp(nom, ens->T[i].nom) == 0) {
         strcpy(ens->T[i].valeur, valeur);
         index = i;
         break;
      }
   }

   if (index == -1) {
      if (ens->nb < (NOMBRE_MAX_VARIABLES - 1)) {
         strcpy(ens->T[ens->nb].nom, nom);
         strcpy(ens->T[ens->nb].valeur, valeur);
         index = ens->nb;
         ens->nb++;
      }
   }

   return index;
}

int nombre_variables(variables * ens) {
   return ens->nb;
}

int trouver_variable(variables * ens, char *nom) {
   int index = -1;

   for (int i=0; i<ens->nb; i++) {
      if (strcmp (ens->T[i].nom, nom) == 0) {
         index = i;
         break;
      }
   }

   return index;
}

char *nom_variable(variables * ens, int i) {
   char *nom = NULL;

   if (i >= 0 && i < ens->nb) {
      nom = ens->T[i].nom;
   }

   return nom;
}

char *valeur_variable(variables * ens, int i) {
   char *valeur = const_empty;

   if (i >= 0 && i < ens->nb) {
      valeur = ens->T[i].valeur;
   }

   return valeur;
}

void modifier_valeur_variable(variables * ens, int i, char *valeur) {
   strcpy (ens->T[i].valeur, valeur);
}
```

**NB :** On note plusieurs choses :

— On initialise juste en plaçant `ens->nb` à 0. Comme ça, le programme saura que ce n'est pas la peine

d'aller consulter le tableau, puisqu'il n'y a pas de variable enregistrée. Pas la peine d'initialiser chaque valeur du tableau à NULL.

— Dans `ajouter_variable`, on n'oublie pas de vérifier qu'on a pas atteint le nombre maximum de variables possible avant d'en insérer une nouvelle.

— Dans `nom_variable`, on vérifie quand même que l'indice donné est cohérent, c'est à dire positif ou nul, et inférieur au nombre maximum de variables possibles.

— Dans `valeur_variable`, voyez comme on initialise la valeur à retourner : `char *valeur = const_empty;` On a déclaré la chaîne vide ("") comme une variable globale, donc à l'extérieur de toute fonction pour être sûr que cette allocation de mémoire (1 octet, avec '\0' dedans) ne soit pas détruite en sortant de la fonction.

**[e]** On attaque maintenant la fonction `trouver_et_appliquer_affectation_variable`. L'algorithme à suivre pour implémenter cette fonction est bien décrit par une superbe animation dans le TD que nous avons fait. Pour rappel, l'objectif est de chercher dans la ligne une déclaration de variables, comme : `mavariable=42` avec potentiellement des espaces avant, mais pas après. Ensuite, une fois l'affectation reconnue, il faut enregistrer la variable dans la table en appelant les fonctions que nous avons déjà codées. Voilà tout de suite une implémentation possible, à la mode des automates avec plein de switch.

```
int trouver_et_appliquer_affectation_variable(variables * ens, char *ligne) {
   enum { ESP_DEB, NOM, EGAL, VAL, ERR } etat = ESP_DEB;
   int affectation_trouvee = 0;
   char *nom = NULL, *val = NULL;

   int i = 0;
   while (ligne[i] != '\0' && etat != ERR) {
      switch (etat) {

      case ESP_DEB:
         switch (ligne[i]) {
         case '=' : etat = ERR     ; break;
         case ' ' : etat = ESP_DEB ; break;
         default  :
            nom = &ligne[i];
            etat = NOM;
            break;
         }
         break;

      case NOM:
         switch (ligne[i]) {
         case '=' :
            ligne[i] = '\0';
            val = &ligne[i+1];
            etat = EGAL;
            break;
         case ' ' : etat = ERR  ; break;
         default  : etat = NOM  ; break;
         }
         break;

      case EGAL:
         switch (ligne[i]) {
         case '=' : etat = ERR  ; break;
         case ' ' : etat = ERR  ; break;
         default  : etat = VAL  ; break;
         }
         break;

      case VAL:
         switch (ligne[i]) {
         case '=' : etat = ERR  ; break;
         case ' ' : etat = ERR  ; break;
         default  : etat = VAL  ; break;
         }
         break;

      default: break;
      }
      i++;
   }
```

```
    if (etat == VAL) {
        ajouter_variable (ens, nom, val);
        affectation_trouvee = 1;
    }

    return affectation_trouvee;
}
```

**NB :** ici aussi, plusieurs choses à noter :

— L'utilisation d'une enum pour noter tous les états possibles au lieu des `#define` que vous connaissez. Une enum est un peu comme une déclaration automatique de constantes numérotées.

```
enum { ESP_DEB, NOM, EGAL, VAL, ERR } etat;
```

`etat` est une variable de type enum, qui peut prendre les valeurs `ESP_DEB`, `NOM`, `EGAL`, `VAL` ou `ERR`. En fait, en pratique, ces constantes sont des entiers, à partir de 0. Donc `ESP_DEB` = 0, `NOM` = 1, etc.. On peut donc utiliser la variable `etat` dans un switch.

— On a découvert une affectation de variable valide si on arrive au bout de la ligne est qu'on est dans l'état `VAL`.

— On arrête le parcours dans le cas où on détecte une syntaxe invalide d'affectation, c'est à dire lorsqu'on entre dans l'état `ERR`.

— Notez comme les caractères (qui ne sont que des nombres représentant un caractère selon la table ASCII) se prêtent bien aux switch ...

**NB2 :** Il faut également voir les limitations de l'implémentation proposée.

— Elle ne prend pas en compte les guillemets (ce n'était pas dans la consigne, mais il n'est pas interdit de faire plus que ce qui est demandé !)

— Elle ne prend en compte que les espaces comme séparateurs. Les tabulations (caractère \n) la font donc planter

— Elle est assez longue (60 lignes). C'est peut être le symptôme que l'on essaye de faire trop de choses en même temps.

**[f]** Voilà la seconde fonction compliquée du TP, vue également en TD. `appliquer_expansion_variables` doit remplacer toutes les utilisations de variable dans la chaîne (`$mavariable` séparé avant et après par des espaces) par leur valeur réelle ou une chaîne vide si la variable appelée n'existe pas. Voilà une implémentation toujours à la mode des automates, avec des switch.

```
void appliquer_expansion_variables (variables * ens, char *ligne_originale, char *
    ligne_expansee) {
    enum { COPY, DOLLAR, EXPAND } etat = COPY;
    char nom[TAILLE_MAX_NOM] = "";

    int i_o = 0, i_e = 0, i_n = 0;
    while (ligne_originale[i_o] != '\0' || etat == EXPAND) {
        switch (etat) {

        case COPY:
            switch (ligne_originale[i_o]) {
            case '$':
                etat = DOLLAR;
                break;
            default:
                ligne_expansee[i_e] = ligne_originale[i_o];
                i_e++;
                etat = COPY;
                break;
            }
            break;

        case DOLLAR:
            if (isalnum(ligne_originale[i_o]) ||
                ligne_originale[i_o] == '*'   ||
                ligne_originale[i_o] == '#') {
                nom[0] = ligne_originale[i_o];
                i_n = 1;
                etat = EXPAND;
```

```
            } else {
               ligne_expansee[i_e]   = '$';
               ligne_expansee[i_e+1] = ligne_originale[i_o];
               i_e = i_e + 2;
               etat = COPY;
            }
            break;

         case EXPAND:
            if (isalnum(ligne_originale[i_o])) {
               nom[i_n] = ligne_originale[i_o];
               i_n++;
               etat = EXPAND;
            } else {
               nom[i_n] = '\0';
               int var_id = trouver_variable (ens, nom);
               if (var_id != -1) {
                  char *val = valeur_variable (ens, var_id);
                  int i_v = 0;
                  while (val[i_v] != '\0') {
                     ligne_expansee[i_e] = val[i_v];
                     i_e++;
                     i_v++;
                  }
               }
               if (ligne_originale[i_o] != '\0') {
                  ligne_expansee[i_e] = ligne_originale[i_o];
                  i_e++;
               } else {
                  i_o--;
               }
               etat = COPY;
            }
            break;

         }
         i_o++;
      }
      ligne_expansee[i_e] = '\0';
}
```

**NB :** Encore des notes pour cette fonction.

- `i_o`, `i_e` et `i_n` sont respectivement les indices du caractère courant pour les chaînes `ligne_originale`, `ligne_expansee` et `nom`.
- Cette fois ci, on utilise la fonction `isalnum` qu'on trouve dans le header `ctype.h` pour déterminer si le caractère courant est alpha numérique, ou autre chose.
- On note aussi que l'on peut continuer le parcours de la chaîne de caractère un cran après le caractère '\0' dans le cas où l'on est encore dans l'état expansion lorsqu'on l'aborde. Ce hack horrible a le mérite de nous faire remarquer qu'un break ne termine que l'expression immédiatement englobante, dans ce cas le switch. Le while qui englobe le switch devient par conséquent impossible à interrompre sans sortir préalablement du switch.

Comme de nombreux étudiants, plutôt que de s'adonner aux plaisirs de l'imbrication de switch, ont sûrement choisi d'implémenter cette fonction avec des if en cascade, voilà une implémentation à cette sauce. On remarquera que pour ne pas s'y perdre, on a créer une fonction intermédiaire `expand`.

```
int expand (variables *ens, char *nom, char *ligne_expansee) {
   int var_id = trouver_variable (ens, nom);
   int i = 0;
   if (var_id != -1) {
      char *val = valeur_variable (ens, var_id);
      while (val[i] != '\0') {
         ligne_expansee[i] = val[i];
         i++;
      }
   }
   return i;
}

void appliquer_expansion_variables(variables * ens, char *ligne_originale, char *
    ligne_expansee) {
```

```
    int i_o = 0, i_e = 0, i_n = 0;
    char nom[TAILLE_MAX_NOM] = "";
    int expanding = 0;

    while (ligne_originale[i_o] != '\0') {
        if (ligne_originale[i_o] == '$') {
            if (!expanding) {
                expanding = 1;
            }
        } else if (ligne_originale[i_o] == ' ') {
            if (expanding) {
                if (i_n == 0) {
                    /* dollar isolé */
                    ligne_expansee[i_e] = '$';
                    i_e++;
                } else {
                    nom[i_n] = '\0';
                    i_e += expand (ens, nom, &ligne_expansee[i_e]);
                    i_n = 0;
                }
                ligne_expansee[i_e] = ligne_originale[i_o];
                i_e++;
                expanding = 0;
            } else {
                ligne_expansee[i_e] = ligne_originale[i_o];
                i_e++;
            }
        } else {
            if (expanding) {
                nom[i_n] = ligne_originale[i_o];
                i_n++;
            } else {
                ligne_expansee[i_e] = ligne_originale[i_o];
                i_e++;
            }
        }
        i_o++;
    }

    if (expanding) {
        if (i_n == 0) {
            /* dollar isolé */
            ligne_expansee[i_e] = '$';
            i_e++;
        } else {
            nom[i_n] = '\0';
            i_e += expand (ens, nom, &ligne_expansee[i_e]);
            i_n = 0;
        }
    }
    ligne_expansee[i_e] = '\0';
}
```

Une dernière chose ... Les fonctions de gestion des variables sont véritablement complexes et difficiles à coder. Comme nous ne sommes pas des génies, nous ne pouvons pas y arriver du premier coup. Il est presque obligatoire de procéder par essais/erreurs. Pour cela, le plus facile est de se faire des tests unitaires afin de se concentrer sur LA fonction que l'on est en train de coder et de s'assurer qu'elle rempli bien son rôle dans tous les cas possibles.

Voilà un main permettant de tester les fonctions de `variables.c`.

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

#include "variables.h"
#include "variables_base.h"

#define TAILLE_MAX_LIGNE 1024

#define OK_AFFECTATION_NB 2
#define KO_AFFECTATION_NB 3
#define EXP_NB 4
```

```
variables ens;

void test_aff () {
   init_variables (&ens);
   ajouter_variable (&ens, "TOTO", "5");
   ajouter_variable (&ens, "TITI", "12");
   ajouter_variable (&ens, "TATA", "Chantons gaiement !");
   afficher_ensemble_variables (&ens);
   printf ("\n");
}

void antitest_trouver_et_appliquer_affectation_variable (char *testcase) {
   init_variables (&ens);

   char buffer[TAILLE_MAX_NOM] = "";

   strcpy (buffer, testcase);
   trouver_et_appliquer_affectation_variable(&ens, buffer);
   if (ens.nb != 0) {
      printf ("[FAIL] test case <%s> recognized affectation\n", testcase);
   } else {
      printf ("[ OK ] test case <%s> OK\n", testcase);
   }
}

void test_trouver_et_appliquer_affectation_variable(char *testcase) {
   init_variables (&ens);

   char *nom = "TOTO", *val = "5";
   char buffer[TAILLE_MAX_NOM] = "";

   strcpy (buffer, testcase);
   trouver_et_appliquer_affectation_variable(&ens, buffer);

   if (ens.nb != 1 ||
       strcmp (ens.T[0].nom, nom) != 0 ||
       strcmp (ens.T[0].valeur, val) != 0) {
      printf ("[FAIL] test case <%s> didn't produce TOTO=5\n", testcase);
   } else {
      printf ("[ OK ] test case <%s> OK\n", testcase);
   }
}

void test_appliquer_expansion_variables (char *testcase, char *expected) {
   init_variables (&ens);
   ajouter_variable (&ens, "TOTO", "5");

   char res[TAILLE_MAX_LIGNE];
   char buffer[TAILLE_MAX_LIGNE];

   strcpy (buffer, testcase);
   appliquer_expansion_variables (&ens, buffer, res);
   if (strcmp (expected, res) != 0) {
      printf ("[FAIL] test case      <%s>\n", testcase);
      printf ("       didn't produce <%s>\n", expected);
      printf ("       but            <%s>\n", res);
   } else {
      printf ("[ OK ] test case <%s> OK\n", testcase);
   }
}

void test_affecter_variables_automatiques () {
   init_variables (&ens);

   int argc = 5;
   char *argv[5] = {
      "/bin/monprog",
      "argument1",
      "argument2",
      "argument3",
      "argument4",
   };
```

```
    printf ("Testing with : argc=%d argv=[%s, %s, %s, %s, %s]\n",
            argc,
            argv[0],
            argv[1],
            argv[2],
            argv[3],
            argv[4]);

    affecter_variables_automatiques (&ens, argc, argv);

    int index = -1;

    if ((index = trouver_variable (&ens, "#")) == -1) {
        printf ("[FAIL] $# non définie. Devrait être : 5\n");
    } else {
        printf ("[ OK ] $# = <%s>\n", valeur_variable (&ens, index));
    }

    if ((index = trouver_variable (&ens, "*")) == -1) {
        printf ("[FAIL] $* non définie. Devrait être : </bin/monprog argument1 argument2
            argument3 argument4>\n");
    } else {
        printf ("[ OK ] $* = <%s>\n", valeur_variable (&ens, index));
    }

    if ((index = trouver_variable (&ens, "0")) == -1) {
        printf ("[FAIL] $0 non définie. Devrait être : </bin/monprog>\n");
    } else {
        printf ("[ OK ] $0 = <%s>\n", valeur_variable (&ens, index));
    }

    if ((index = trouver_variable (&ens, "1")) == -1) {
        printf ("[FAIL] $1 non définie. Devrait être : <argument1>\n");
    } else {
        printf ("[ OK ] $1 = <%s>\n", valeur_variable (&ens, index));
    }

    if ((index = trouver_variable (&ens, "2")) == -1) {
        printf ("[FAIL] $2 non définie. Devrait être : <argument2>\n");
    } else {
        printf ("[ OK ] $2 = <%s>\n", valeur_variable (&ens, index));
    }

    if ((index = trouver_variable (&ens, "3")) == -1) {
        printf ("[FAIL] $3 non définie. Devrait être : <argument3>\n");
    } else {
        printf ("[ OK ] $3 = <%s>\n", valeur_variable (&ens, index));
    }

    if ((index = trouver_variable (&ens, "4")) == -1) {
        printf ("[FAIL] $4 non définie. Devrait être : <argument4>\n");
    } else {
        printf ("[ OK ] $4 = <%s>\n", valeur_variable (&ens, index));
    }
}

int main() {
    init_variables (&ens);

    test_aff ();

    printf ("\n");

    test_trouver_et_appliquer_affectation_variable ("TOTO=5");
    test_trouver_et_appliquer_affectation_variable ("     TOTO=5");

    printf ("\n");

    antitest_trouver_et_appliquer_affectation_variable ("     TOTO =5");
    antitest_trouver_et_appliquer_affectation_variable ("     TOTO= 5");
    antitest_trouver_et_appliquer_affectation_variable ("     TOTO=5 ");

    printf ("\n");
```

```
    test_appliquer_expansion_variables ("$TOTO", "5");
    test_appliquer_expansion_variables ("aeofubaoeu $TOTO aeufbaeub", "aeofubaoeu 5
        aeufbaeub");
    test_appliquer_expansion_variables ("aeofubaoeu $TOTO", "aeofubaoeu 5");
    test_appliquer_expansion_variables ("aeofubaoeu $ aefae", "aeofubaoeu $ aefae");

    printf ("\n");

    test_affecter_variables_automatiques ();
    return 0;
}
```

Ces tests dépassent le cadre de ce TP, mais vous pouvez observer, dans le `main` combien il est facile de rajouter des cas de test pour vérifier que nos fonctions produisent bien le résultat escompté même dans les situations les plus tordues ... Ce programme produit la sortie suivante lorsque les fonctions de `variables.c` sont implémentées correctement :

```
corentin@gazelle:src $ ./main_unit_test
TOTO=5
TITI=12
TATA=Chantons gaiement !


[ OK ] test case <TOTO=5> OK
[ OK ] test case <     TOTO=5> OK

[ OK ] test case <     TOTO =5> OK
[ OK ] test case <     TOTO= 5> OK
[ OK ] test case <     TOTO=5 > OK

[ OK ] test case <$TOTO> OK
[ OK ] test case <aeofubaoeu $TOTO aeufbaeub> OK
[ OK ] test case <aeofubaoeu $TOTO> OK
[ OK ] test case <aeofubaoeu $ aefae> OK

Testing with : argc=5 argv=[/bin/monprog, argument1, argument2, argument3, argument4]
[ OK ] $# = <4>
[ OK ] $* = </bin/monprog argument1 argument2 argument3 argument4>
[ OK ] $0 = </bin/monprog>
[ OK ] $1 = <argument1>
[ OK ] $2 = <argument2>
[ OK ] $3 = <argument3>
[ OK ] $4 = <argument4>
$
```

**[g]** Après les fonctions sur les variables, la fonction `decouper_ligne` paraît être une promenade de santée. Elle a été également largement décrite en TD, place à une implémentation possible.

```
void decouper_ligne (char *ligne, char *ligne_decoupee[]) {
    int car_prec_espace = 1, i = 0, index = 0;

    while (ligne[i] != '\0') {
        if (ligne[i] == ' ' && !car_prec_espace) {
            ligne[i] = '\0';
        }

        if (ligne[i] != ' ' && car_prec_espace) {
            ligne_decoupee[index] = &ligne[i];
            index++;
        }

        car_prec_espace = (ligne[i] == ' ' || ligne[i] == '\0');
        i++;
    }
    ligne_decoupee[index] = NULL;
}
```

**[h]** Nous passons aux variables automatiques. Toute cette partie du TP est “complémentaire”, c'est à dire facultative. Elle donne un exemple de l'utilisation de la fonction `strcat`.

On rappelle que toutes les variables en Shell sont des chaînes de caractères. Même les nombres sont donc stockés sous cette forme et ne sont interprétés comme nombre que lorsqu'on les donne à des fonctions comme `test`, qui s'occupe alors de faire la conversion.

`$#` représente le nombre de paramètres passé à la commande. Dans notre cas, cela correspondra donc à `argc - 1`, vu que le Shell ne compte pas le nom du programme comme un paramètre.

Voilà donc une implémentation de la fonction `affecter_variables_automatiques`.

```
void affecter_variables_automatiques(variables *ens, int argc, char *argv[]) {
   char buffer[TAILLE_MAX_NOM];
   char buffer_cat[TAILLE_MAX_NOM] = "";
   int i = 0;

   sprintf (buffer, "%d", argc - 1);
   ajouter_variable (ens, "#", buffer);

   while (i < argc) {
      sprintf (buffer, "%d", i);
      ajouter_variable (ens, buffer, argv[i]);
      strcat (buffer_cat, argv[i]);
      if (i < argc - 1) {
         strcat (buffer_cat, " ");
      }
      i++;
   }

   ajouter_variable (ens, "*", buffer_cat);
}
```