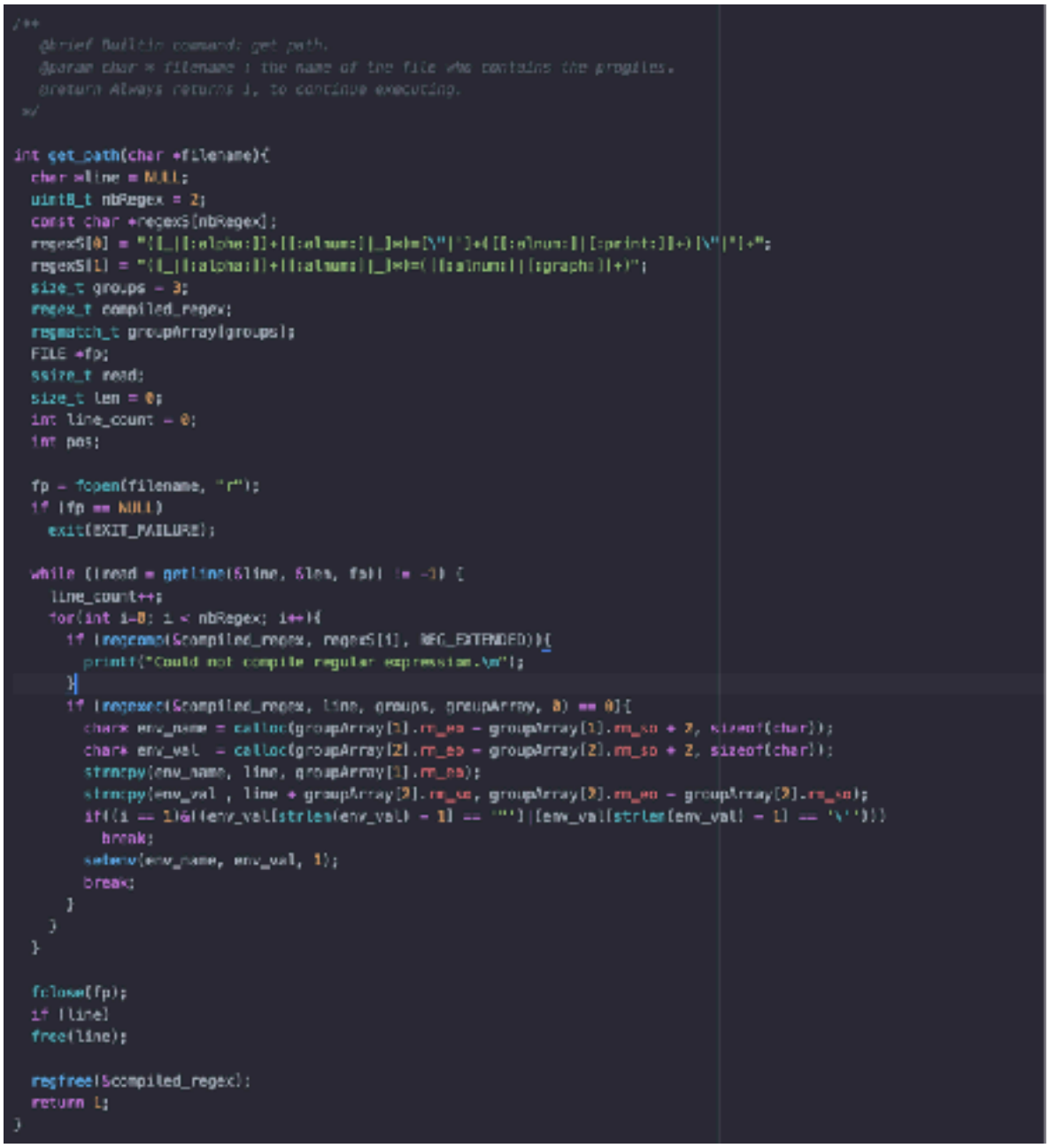
Rapport sur l’interpréteur de commandes

L'objectif de ce projet est d'implémenter un interpréteur de commandes Unix simplifié en C, similaire à bash.

A l’issue de ce projet nous avions comme rôle d’implémenter ce programme avec les demandes suivante :

1.
Au démarrage, l’interpréteur doit récupérer le répertoire de travail de l'utilisateur ayant exécuté l’interpréteur, lire un fichier appelé profile dans ce répertoire et configurer les variables d'environnement spécifiées dans ce fichier.



```c
/**
   @brief Builtin command: get path.
   @param char * filename : the name of the file who contains the progiles.
   @return Always returns 1, to continue executing.
 */

int get_path(char *filename){
  char *line = NULL;
  uint8_t nbRegex = 2;
  const char *regexS[nbRegex];
  regexS[0] = "([_|[:alpha:]]+[[:alnum:]|_]*=[\"|']+([[:alnum:]|[:print:]]+)[\"|']+";
  regexS[1] = "([_|[:alpha:]]+[[:alnum:]|_]*=([[:alnum:]|[:graph:]]+)";
  size_t groups = 3;
  regex_t compiled_regex;
  regmatch_t groupArray[groups];
  FILE *fp;
  ssize_t read;
  size_t len = 0;
  int line_count = 0;
  int pos;

  fp = fopen(filename, "r");
  if (fp == NULL)
    exit(EXIT_FAILURE);

  while ((read = getline(&line, &len, fp)) != -1) {
    line_count++;
    for(int i=0; i < nbRegex; i++){
      if (regcomp(&compiled_regex, regexS[i], REG_EXTENDED)){
        printf("Could not compile regular expression.\n");
      }
      if (regexec(&compiled_regex, line, groups, groupArray, 0) == 0){
        char* env_name = calloc(groupArray[1].rm_eo - groupArray[1].rm_so + 2, sizeof(char));
        char* env_val  = calloc(groupArray[2].rm_eo - groupArray[2].rm_so + 2, sizeof(char));
        strncpy(env_name, line, groupArray[1].rm_eo);
        strncpy(env_val , line + groupArray[2].rm_so, groupArray[2].rm_eo - groupArray[2].rm_so);
        if((i == 1)&&(env_val[strlen(env_val) - 1] == '"')||(env_val[strlen(env_val) - 1] == '\''))
          break;
        setenv(env_name, env_val, 1);
        break;
      }
    }
  }

  fclose(fp);
  if (line)
  free(line);

  regfree(&compiled_regex);
  return 1;
}
```

Nous avons alors créer une fonction qui permet de récupérer le nom du fichier en argument de la fonction. Nous coupons ensuite les lignes du fichier afin de récupérer les chemins associés aux deux environnement et ensuite les assignons en copiant les environnements dans leurs variables globales associés.
A la fin de la fonction nous fermons le fichier et nous libérons la mémoire alloué pour le compteur de lignes du fichier et du regex.
Nous appelons cette fonction au début du programme, avant même que nous lançons la boucle infinie pour récupérer les commandes.

Nous avons besoin d'appeler cette fonction au début du programme car elle permet aux variables globales d'être assigné, et ainsi à l'appel d'un programme nous pourrions avoir le chemin de parcours pour trouver l'executable.

2.
Votre interpréteur devra implémenter les commande internes cd, pwd, alias qui respectivement change le répertoire de travail de l'interpréteur, affiche le répertoire de travail courant et assigne un alias à une ligne de commande indiquée en paramètre.

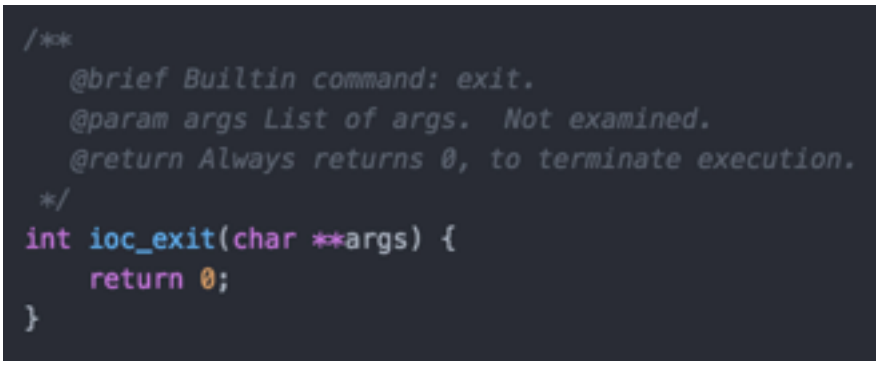
Nous avons commencer par la commande cd qui permet de se déplacer dans un dossier. Nous avons donc utilisé l'appel système « chdir » pour se déplacer entre les différents dossiers.
Nous testons si l'argument est null, s'il ne l'est pas nous executions l'appel système.
Nous testons si il y a eu une perror.

Nous retournons 1 à la fin du programme afin de continuer la boucle infinie de l'interpréteur de commandes et d'attendre un nouvel appel.

Nous avons ensuite réaliser la commande pwd qui permet de nous dire dans quel chemin nous sommes (suites de dossiers).
Nous avons donc utilisé l'appel système getcwd qui permet de récupérer le répertoire de travail actuel puis nous l'affichons.

Ensuite nous avons réaliser une fonction pour créer des alias qui crée simplement un lien entre deux fichiers avec le simple appel système link.

3.
Il devra être possible de quitter votre interpréteur avec un code de sortie donné avec la commande interne exit.



La fonction exit permet d'arrêter le programme en retournant 0 et donc en arrêtant la boucle d'attente de commandes.

4. Nous avons enfin une boucle qui attend les commandes, c’est notre fonction principale qui attend les autres fonctions qui sont listés.

Cette fonction permet alors d’attendre les commandes entrées par l’utilisateur. Elle récupère les arguments et permet de les envoyé dans la fonction d’execution.