Programmation impérative, projet 2019/2020

Ce projet consiste en la création d'un interpréteur et d'un débogueur d'un langage de programmation en deux dimensions appelé *prog2d*, majoritairement constitué de caractères organisés dans une grille.

Structures

Pile

```
typedef struct element element;

struct element {
  int val;
  element* next;
};

typedef struct element* pile;
```

L'interpréteur comme le débogueur utilisent une pile pour stocker des entiers lorsqu'ils interprètent un fichier *prog2d*. J'ai choisi de représenter les piles sous la forme d'une liste chaînée dont les maillons contiennent un entier et l'adresse du maillon suivant.

Les différentes fonctions implémentées sur les piles permettent de :

-créer une pile vide
-empiler un entier sur une pile
-dépiler un entier d'une pile (0 si elle est vide)
-afficher les contenu d'une pile (entier et caractère correspondant) sur une ligne

Coordonnées

```
struct coord {
  int x;
  int y;
};

typedef struct coord coord;
```

La structure coord est simplement un couple d'entier, ce qui permet de gérer plus efficacement la position du curseur dans la grille.

Sauvegarde

```
typedef struct sauvegarde sauvegarde;

struct sauvegarde {
  char** grille;
  coord position;
  int direction;
  pile p;
  int mode_ascii;
  sauvegarde* next;
};
```

La structure sauvegarde contient un ensemble de variables définissant l'état du débogueur à l'étape t. Elle contient donc l'état de la grille, la position du curseur, la direction de celui-ci, l'état de la pile et l'état du mode_ascii (activé ou non). Elle n'est utilisée que dans le cadre de la structure pile de sauvegarde pour la commande *prec* du débogueur.

Pile de sauvegarde

```
typedef struct sauvegarde* pile_sauvegarde;
```

La structure pile_sauvegarde est une liste chaînée de sauvegarde. Elle est utilisée pour stocker l'état du débogueur à chaque étape et pour pouvoir y revenir lorsque l'utilisateur entre la commande *prec*.

Les différentes fonctions implémentées sur les piles permettent de :

-empiler une sauvegarde sur une pile de sauvegarde

-dépiler une sauvegarde d'une pile de sauvegarde

Interpréteur

L’interpréteur ouvre le fichier *prog2d* qui lui est passé en argument, puis lit un caractère à la fois en effectuant l’action correspondant à ce dernier à l’aide d’un *switch* et se déplace dans la grille, ceci jusqu’à atteindre le caractère ‘@’ marquant la fin du programme.

```
if (*mode_ascii == 1) {
  if (grille[position->x][position->y] == '"') {
        *mode_ascii = 0;
  }
  else {
        empile(p,grille[position->x][position->y]);
  }
 }
 else {
  switch (grille[position->x][position->y]) {
  case '+' : addition(p); break;
  case '-' : soustraction(p); break;
  case '*' : multiplication(p); break;
  case ':' : division(p); break;
  case '%' : reste(p); break;
  case '!' : non_logique(p); break;
  case '`' : plus_grand_que(p); break;
  case '>' : *direction = 2; break;
  case '<' : *direction = 6; break;
  case '^' : *direction = 0; break;
  case 'v' : *direction = 4; break;
  case '?' : hasard_direction(direction); break;
  case '\' : entier_direction(direction, p); break;
  case ']' : gauche_45(direction); break;
  case '[' : droite_45(direction); break;
  case '_' : conditionnelle_horizontale(direction, p); break;
  case '|' : conditionnelle_verticale(direction, p); break;
  case '/' : conditionnelle_diagonale_droite(direction, p); break;
  case '\\' : conditionnelle_diagonale_gauche(direction, p); break;
  case '"' : *mode_ascii = 1; break;
  case '=' : dupliquer(p); break;
  case '$' : echanger(p); break;
  case ';' : depile(p); break;
  case '.' : afficher_entier(p); break;
  case ',' : afficher_caractere(p); break;
  case '#' : pont(*direction, largeur, hauteur, position, p); break;
  case 'g' : get(grille, largeur, hauteur, p); break;
  case 'p' : put(grille, largeur, hauteur, p); break;
  case '&' : demander_entier(p); break;
  case '~' : demander_caractere(p); break;
  case '0' :
  case '1' :
  case '2' :
  case '3' :
  case '4' :
  case '5' :
  case '6' :
  case '7' :
  case '8' :
  case '9' : empile(p, grille[position->x][position->y] - 48); break;
  default : break;
  }
}
```

J’ai tout d’abord compris que l’action *put* correspondant au caractère ‘p’ était sensée modifier le fichier *prog2d*. C’est pourquoi j’ai commencé à développer mon interpréteur de telle sorte à ce qu’il ouvre le fichier *prog2d* puis qu’il lise ses caractères un par un jusqu’à lire ‘@’. Les opérations étaient donc effectuées directement sur le fichier.
Cependant, la commande *put* -comme toutes les autre d’ailleurs- ne doit pas modifier le fichier *p2d*. Après avoir été informé de mon erreur de compréhension, j’ai donc modifié mon programme. Dans sa nouvelle version, l’interpréteur ouvre le fichier *prog2d*, copie sa grille de caractères dans un tableau à deux dimensions et le ferme ensuite.
Cette méthode s’est d’ailleurs avérée beaucoup plus simple pour gérer la position du marqueur, étant donné que les lignes sont rangées dans des tableaux séparés, contrairement à la méthode précédente où le fichier ouvert sur lequel le programme travaillait n’était qu’un long tableau à une dimension.

Débogueur

Le débogueur ouvre le fichier *prog2d* de la même façon que l’interpréteur mais attend ensuite les commandes entrées par l’utilisateur avant de poursuivre.

Affichage

Le débogueur affiche en outre à chaque étape l’état de la grille et la pile et repère la position du curseur.

Problématiques rencontrées

Les chaînes de caractère en C étant des pointeurs vers l'adresse du premier élément d'un tableau de caractères, tester l'égalité de deux d'entre elles pose problème. Heureusement, le module string.h propose une fonction de comparaison de deux chaînes, renvoyant 0 si elles sont identiques. Il est donc facile de détecter les commandes *step*, *run*, *restart*, *quit* et *prec*. Néanmoins, dans le cas des commandes à paramètres *step n*, *breakpoint x y*, *removebp x y* et *prec n*, cette commande ne suffit pas.
Il a donc fallu que je copie le début de l'input utilisateur dans une chaîne de caractère de la bonne longueur (celle de la commande que je voulais tester) pour ensuite comparer celle-ci avec la commande. Par exemple, pour « step n » :

```
char temp[5] = {'0','0','0','0','\0'};
    strncpy(temp,commande,4);
    if (!strcmp(temp,"step")) {
        //on récupère l'entier n après step (sscanf) et on effectue step n
    }
```

La commande *prec* a été la plus dure à programmer à cause de la complexité de la structure pile_sauvegarde. En effet, étant donné que pile_sauvegarde est une pile de structures contenant elles-mêmes des piles, il a fallu faire attention à la manière dont j'empilais et je dépilais des sauvegardes de ma pile.

Le premier problème a été la modification intempestive des piles d'entier contenues dans ma pile de sauvegarde, que j'ai résolu en faisant attention à empiler une copie des éléments de ma pile d'entier d'origine dans celle de la sauvegarde que j'empilais.

```
pile temp = NULL;
int x;
while(p != NULL) {//se termine lorsque l'on atteint le bout de la pile p
  x = p->val;
  p = p->next;
  empile(&temp,x);
}
while (temp != NULL) {//se termine lorsque l'on atteint le bout de la pile temp
  x = depile(&temp);
  empile(&(s->p),x);
}
```

Le deuxième problème a été la fuite de mémoire lorsque je dépilais des sauvegardes. Cette dernière provenait des piles d'entiers de ces sauvegardes que je ne vidais pas après m'en être servi.

```
int tempi;
while (*p != NULL) {//se termine lorsque l'on atteint le bout de la pile p
  depile(p);
}
while(sauv.p != NULL) {//se termine lorsque l'on atteint le bout de la pile sauv.p
  tempi = depile(&(sauv.p));
  empile(p,tempi);
}
```

Idées d'optimisation du code

Avec le recul, j'ai eu quelques idées pour avoir un code plus propre. Pour alléger visuellement le code, il aurait fallu utiliser la structure sauvegarde pour les instructions correspondant au caractère lu et pas seulement dans le cadre de la commande *prec.x*

De plus, il n'est pas nécessaire de stocker autant de données pour la commande *prec*. En effet, il suffirait de stocker l'état des étapes ayant effectué des opérations aléatoires (changement aléatoire de direction) et celles ayant demandé des input de l'utilisateur (demander_entier ou demander_caractère), puis de recommencer l'interprétation du fichier *prog2d* jusqu'à l'étape désignée par la commande *prec* entrée en remplaçant les valeurs aléatoire et celles entrées par l'utilisateur par celles que l'on a stockées ; ceci économiserait de l'espace mémoire.