Rapport de projet

Vivien Gagliano

Table des matières

Approche choisie

Le sujet proposé nous demandait d'implémenter un interpréteur et un débogueur pour le langage de programmation prog2d.

Les programmes de ce langage sont des fichiers que l'on peut assimiler à une grille de caractères, grille de dimensions données en première ligne du programme. L'exécution d'un tel programme se fait en considérant qu'un curseur se déplace sur cette grille, en exécutant à chaque étape l'instruction correspondante au caractère se trouvant à sa position.

Interpréteur

Il fut donc nécessaire de définir des structures, afin de pouvoir manipuler non seulement la grille et le curseur, mais aussi la pile d'entiers que fait intervenir la plupart des instructions.

Afin de rester aussi général que possible, de pouvoir réutiliser mes travaux, mais aussi par soucis de clarté, j'ai d'une part choisi de séparer toutes ces structures, de les définir comme différentes librairies, et de les inclure dans mon programme principal.

D'autre part, toujours dans l'idée de généralisation, j'ai décidé non pas de travailler sur des piles d'entiers, ou des tableaux de caractères, mais plutôt de définir des types *element*, et de coder mes librairies avec. Ainsi, il me suffit de changer cet *element* pour changer le type de mes piles et grilles.

Pile

La pile a été implémentée de manière assez naturelle à l'aide de listes chaînées (implémentation par ailleurs déjà étudiée en cours). On définit un élément maillon, composé d'une valeur entière, et d'un pointeur vers un autre maillon, et notre pile est définie comme pointeur vers un maillon.

```
typedef struct cell cell;
typedef cell* stack;
struct cell {
    stack_elem head;
    stack tail;
};
```

Les fonctions de base, notamment pop et push (dépiler et empiler respectivement), on aussi été définies de manière sensiblement similaire à ce qui avait été vu en cours.

Matrice

Les matrices, utilisées pour représenter notre fichier comme une grille de caractères, ont été définies comme une structure comportant les dimensions de la grille, ainsi qu'un tableau qui est la véritable matrice.

```
struct matrix {
    int width, height;
    matrix_elem** grid;
};
typedef struct matrix matrix;
```

Ce choix d'incorporer la taille au type matrice permet de synthétiser toutes les informations dans une variable, améliorant ainsi la lisibilité, et évite de devoir transmettre les dimensions à chaque appel.

Curseur

Le curseur quant à lui est une structure composée de deux coordonnées, qui servent à repérer le curseur sur la grille, ainsi qu'une direction, *i.e.* une énumération sur les huit déplacements canoniques possibles, qui permet de savoir quelle est sa vitesse instantannée.

```
struct cursor {
int pos_x, pos_y;
direction dir;
};
typedef struct cursor cursor;
```

La librairie curseur comporte aussi une fonction *move* qui permet de déplacer un curseur sur une grille, en fonction de sa position et direction actuelle, en prennant en compte que la grille est torique.

Exécutable

S'appuyant sur ces librairies, l'interpréteur devait dans un premier temps lire et récupérer le programme .p2d donné en argument.

J'ai décidé pour cela d'utiliser un flux libc : en stockant la première ligne dans un tampon à l'aide de la fonction *getline*, j'ai pu récupérer les dimensions de la grille. Il m'a ensuite suffi de créer une matrice de la bonne taille, et de la remplir élément par élément avec la fonction *getc*.

Une fois le curseur créé et initialisé, et la grille remplie, il suffit de parcourir cette dernière en suivant les instructions.

Le programme ne terminant que lorsque le curseur rencontre le caractère @, la boucle

```
while (curr_val != '@')
```

s'imposa naturellement pour le parcours de la grille. On distingue alors chaque cas avec un *switch* sur le caractère pointé par le curseur, en déplaçant le curseur et actualisant la valeur actuelle à l'aide de fonctions définies au préalable à chaque étape.

Débogueur

Le débogueur demandé devait nous permettre d'exécuter un programme .p2d à l'aide de l'interpréteur, en le compilant étape par étape et en utilisant des commandes prédéfinies.

Produire un véritable débogueur tel que gdb m'étant impossible, je choisis pour réaliser ce débogueur de modifier le code de mon interpréteur, et d'y ajouter un mode débogeur. Dans un premier temps, je sortis la lecture du caractère actuel et son interprétation du `main` de mon interpréteur, la transformant en une procédure

```
void run_one_step(matrix* mat, cursor* curs, stack* s, char* curr_val)
```

Ainsi ma boucle `while` principale devint

```
while (curr_val != '@') {
    run_one_step(&mat, &curs, &s, &curr_val);
}
```

J'ai ensuite ajouté l'action du débogueur, c'est à dire l'affichage de l'état du programme, ainsi que la lecture et réalisation des commandes, dans la boucle `while`, avant l'exécution de `run_one_step`.

Une fois encore, au vue des commandes à implémenter, il me fut nécessaire de définir quelques structures et fonctions, qui facilitèrent l'organisation du mode débog. Le tout est écrit dans une librairie debugging, incluse dans le .c de l'interpréteur.

Liste d'étapes

Afin de naviguer entre les différentes étapes de l'éxecution du programme (notamment pour la commande `prec`), je choisis d'enregistrer toutes les étapes dans un tableau.

Je définis donc la structure d'étape, comportant toutes les informations du programme à une étape donnée, *i.e.* la grille de caractères, le curseur, et la pile.

```
typedef struct {
matrix grid;
cursor curs;
stack stack;
} one_step;
```

A partir de cela, on définit simplement le tableau qui nous sera nécessaire pour enregistrer les différentes étapes avec

```
typedef one_step* step_list;
```

Enfin, on définit un compteur `int step_count` qui compte le numéro de l'étape actuelle.

Mode débogeur

Le mode débog sur l'interpreteur est contrôlé par un boolén `int debug_mode`, initialisé à 0 pour l'éventualité où l'on lancerait l'interpréteur sans vouloir déboguer. Le coeur de l'action du débogueur ne se fait que si ce booléen est reglé à 1. Si c'est le cas, on effectue toutes les vérifications et commandes avant de pouvoir lancer la fonction `run_one_step`

En parallèle, un autre booléen est défini pour contrôler si cette dernière fonction doit s'effectuer ou non. En effet en fonction de la commande tapée, on va devoir ou non avancer d'une étape le programme. Initialisé à 1 (encore une fois, pour les fois où l'on ne lance que l'interpréteur), ce booléen est reglé à 0 à chaque itération où le mode débog est activé.

En plus des lignes de code ajoutées dans la boucle `while`, on initialise dans le `main` (avant `while`) toutes les variables et structures qui nous sont nécessaires pour déboguer.

A chaque itération où le mode débogueur est activé, la commande de l'utilisateur est récupérée et stockée dans un tampon à l'aide de `fgets(input, 256, stdin)`. On récupère ensuite la première chaîne de caractères, ainsi que les deux premiers entiers de cette entrée avec

```
sscanf(input, "%s %i %i", command, &p, &q);
```

On effectue ensuite un filtrage sur la chaîne de caractère pour savoir quelle commande réaliser.

A noter qu'une commande non valide entraîne le débogueur à redemander une nouvelle commande, et ce jusqu'à en recevoir une valide. De plus, chaque commande vérifie que les arguments qu'elle reçoit sont valides (par exemple que n soit plus petit que le numéro d'étape actuel pour `prec n`). Dans le cas contraire, tout se comporte comme si la commande était invalide.

Commandes utilisateur

Chaque commande (ou presque) ayant eu une implémentation différente, une explication rapide et exhaustive de ces implémentations va suivre.

`step n` : cette commande règle `debug_mode` à 1. Pour pouvoir faire cela pendant `n` itérations seulement, on crée un compteur `int skip_debug`, et on ajoute à la condition de lancement des commandes de débog `! skip_debug`. Il suffit ainsi d'ajouter `n` à ce compteur lors de la commande, et de lui retrancher 1 à chaque appel à `run_one_step`.

`run` : retire le mode débog et règle `should_run` à 1, de telle sorte que `run_one_step` soit appelée. De plus, un compteur `int run_mode` est utilisé pour éviter les interférence avec la commande `run n` : on actualise `skip_debug` seulement si le mode run est desactivé.

`restart` : il suffit pour cette commande de remettre la grille, le curseur, la pile et le compteur d'étapes, auquels on accède avec le tableau des étapes, dans leur état initial.

`quit` : on termine simplement le `main`.

`prec n` : de manière similaire à la commande `restart`, on remet l'état du programme à celui de la $n^{ème}$ précédente étape si possible.

`breakpoint x y` : ayant défini au préalable un tableau de `positions` (une structure simplement composée de deux coordonnées, x et y)
`position* breakpoints`, il suffit d'ajouter la position `(x, y)` à ce tableau. On vérifie en début de chaque itération si la position actuelle figure parmi les points de cette liste, auquel cas on force l'activation du débogueur, en activant le mode débog, désactivant le mode `run` et remettant `skip_debug` à 0.

`removebp x y` : on parcours simplement la liste `breakpoints` en retirant tout point de coordonnées `(x, y)`.

Difficultés rencontrées et solutions envisagées

Les difficultés rencontrées furent nombreuses, et une grande partie d'entre elles sont dues à des erreurs d'inattention, de syntaxe, ou de simples coquilles. Ces erreurs là furent résolues de manière relativement simple (mais non nécessairement rapide), après attentive relecture du code.

D'autres problèmes, moins immédiats à résoudre, furent aussi rencontrés. En voici la liste exhaustive et chronologique, ainsi que les solutions envisagées.

Récupérer les dimensions

La première grosse difficulté qui s'imposa à moi fut de récupérer la taille de la grille. Deux problèmes ici : d'une part comment lire la première ligne *de taille inconnue* du programme ? où la stocker sans connaître quelle taille allouer à un potentiel tampon ?

Et ensuite comment récupérer les dimensions à partir de cette ligne, sachant qu'elles y sont stockées en tant que caractères, et non pas qu'entiers ?

J'ai d'abord pensé allouer un tampon pour une ligne de grande taille, mais il est toujours possible d'avoir un fichier plus grand que cette taille, et alors l'interpréteur ne fonctionnerait pas pour tout fichier.

La solution a été d'utiliser la fonction *getline*, une fonction spécialement prévue à cet effet. En effet, si on lui donne en argument un tampon NULL, la fonction alloue automatiquement un tampon de la bonne taille.

Le deuxième problème se résolu de manière similaire : un appel à la fonction *sscanf* de la librairie stdio.h me permi de lire une valeur depuis un flux, ici la première ligne, et donc récupérer les dimensions qui m'intéressaient.

Caractères invisibles

Un autre problème qui, certes pourtant simple, me prit du temps à cerner et résoudre, est que le remplissage de ma grille se faisait avec un décalage d'un caractère par ligne.

La raison pour cela est que, même si ils n'existent pas dans le fichier, des caractères \n, *i.e.* retour à la ligne, sont présents dans le FILE* que l'on utilise pour lire le fichier.

Une fois l'origine du problème identifiée, sa résolution fut simple : il suffit de sauter un caractère à chaque fin de ligne lors du remplissage de la grille, à l'aide de la fonction *fseek*.

Problèmes de coordonnées

La gestion des systèmes de coordonnées fut aussi une source d'erreurs, pour deux raisons. D'une part, à cause de la construction choisie pour la grille, *i.e.* un tableau de pointeurs vers des tableaux de caractères, il m'est arrivé de confondre ligne et colonne de la grille à différentes reprises, et notamment dans la fonction de création de matrice *create_matrix*.

La deuxième difficulté fut de réaliser que les coordonnées du curseur ne correspondaient pas à celles de la grille. En effet `curs.pos_x` correspond aux colonnes, et `curs.pos_y` aux lignes de la grille. La commande pour actualiser la valeur actuelle du curseur était donc

```
*new_char = mat.grid[curs.pos_y][curs.pos_x];
```

ce qui me parut étrange à priori.

Lancer l'interpréteur en mode débog

Une grosse difficulté fut de réussir à lancer l'interpréteur en mode débog, mais sans lui ajouter d'argument supplémentaire (consigne à respecter).

Pour contourner cette difficulté, j'ai utilisé une variable d'environnement `DEBUG_MODE`. Il suffit ainsi de vérifier dans le `main` de l'interpréteur si une telle variable a été définie avec `getenv`, auquel cas on active le mode débogueur. Le débogueur n'a donc qu'à lancer l'interpréteur en initialisant DEBUG_MODE à 1, avec la commande

```
system("DEBUG_MODE=1 ./intepretor argument");
```

Récupérer la commande utilisateur en mode débog

Récupérer et stocker la commande utilisateur fut difficile, car il fallut prévoir l'éventualité où la commande n'a pas un des formats attendus.

De plus, après avoir récupéré la commande en tant que chaîne de caractère, il fallut savoir de quelle commande il s'agissait. Or `switch` ne fonctionne pas avec des chaînes de caractères.

La solution au premier problème a été de stocker toute l'entrée dans un tampon de taille maximum prédéfinie à 256 caractères (on a donc une contrainte sur l'entrée) avec `fgets`. Ensuite seulement, on peut récupérer la première chaîne de caractère et les deux premiers entiers avec `sscanf`, qui ignore les caractères ne correspondant pas au format demandé.

J'ai ensuite du créer une fonction `command_filter(char* command)`, pour filtrer "manuellement" les commandes reçues en leur attribuant une valeur entière (de manière similaire à une énumération), afin de faire le `switch` sur ces valeurs.

Tableau de taille inconnue

Lors de la création des tableaux contenant les étapes et les points d'arrêts, on ne peut pas connaître le nombre d'éléments à stocker. Même allouer une très grande quantité de mémoire n'est pas suffisant (on peut en effet potentiellement ajouter des points d'arrêts en nombre aribtrairement grand).

Pour pallier à cela, j'ai initialisé ces tableaux en leur allouant une quantité aribtraire de mémoire. Puis à chaque fois que je veux ajouter un élément à ce tableau, je vérifie que la quantité maximum n'est pas atteinte ; si c'est le cas, je ré-alloue de la mémoire avec la fonction `realloc`.

step ou step n

Ma décision de garder la valeur de p et q entre chaque itération, qui me permet de ré-exécuter facilement les commandes `step n` et `prec n`, fait

aussi que l'on ne peut pas lancer `step` après une de ces deux précédentes commandes (`step n` se lancera à la place).

Pour remédier à cela, j'ai utilisé la valeur de retour de `sscanf` : un entier indiquant le nombre d'instances correctement appariées et attribuées. Ainsi en vérifiant si un seul motif ou non a été lu, on peut savoir si il faut effectuer `step` ou `step n`.

Initialisation de la liste des points d'arrêts

Lors de l'allocation dynamique initiale pour la liste des points d'arrêts, chaque position du tableau est initialisée à `(0, 0)` (utilisation de la fonction `calloc`), or `(0, 0)` est une position atteignable de la grille, ceci interfère donc avec le test pour savoir si la position actuelle est un point d'arrêt.

Il me fallut donc initialiser manuellement les positions du tableau à `(-1, -1)` après création, mais aussi après réallocation de mémoire.

Limites du programme

Dû à des choix de modélisation, ou des solutions envisagées à certains problèmes, mes programmes ont certaines limites. Voici la liste de celles que je remarquai.

Mémoire occupée non utilisée en mode interpréteur

Le fait d'avoir "incorporé" mon débogueur au code de mon interpréteur rend certes le code moins lisible, cependant le réel problème est que la mémoire allouée pour des options de débogage (liste des étapes par exemple) est inutilisée. Cela peut donc ralentir, ou empêcher le programme de tourner sur une machine à faible RAM.

Affichage de l'état du programme limité

Pour afficher l'état actuel du programme, j'ai décidé de créer une matrice de taille légerement supérieure, de manière à inclure les caractères d'affichage (position du curseur, graduations) directement dans la matrice.

Ainsi, les graduations étant de cinq en cinq, si la largeur du fichier excède 99 999, les graduations vont empiêter l'une sur l'autre.

Cependant un fichier de cette largeur étant, même si possible, peu probable, j'ai décidé de laisser ce problème sans solution (le problème n'étant qu'un problème d'affichage, l'exécution du programme n'est pas affectée)

prec n ne lit pas les points d'arrêts

Pour implémenter les points d'arrêts, je vérifie au début de chaque itération de l'interpréteur si la position actuelle est un point d'arrêt. Or la commande `prec n` ne fait pas avancer le programme à rebours, mais met à jour l'étape actuelle. On ne peut donc pas savoir si l'on passe par un de ces points dans les étapes sautées.