Projet de programmation impérative : Rapport

Jérémie Spiesser

le 28 décembre 2019

Sommaire

Codes d'erreurs renvoyés par l'interpréteur et le débogueur

L'interpréteur et le débogueur peuvent renvoyer différents codes d'erreurs :

- 1 : Pas assez d'arguments.
- 2 : Erreur lors de l'ouverture du fichier.
- 3 : Dimensions (en entête du fichier) négatives (donc incorrectes).
- 4 : La grille n'est pas carrée (s'il n'y a pas le même nombre de caractères par lignes partout).

Choix d'implémentation

L'interpréteur et le débogueur pour le language prog2d manipulent tous deux 3 structures de données importantes :

- une pile
- une grille
- un curseur

La pile

Il s'agit d'une pile d'entier. Etant donné qu'au début de l'exécution, on ne connaît pas la taille maximale que cette pile doit avoir, pour optimiser la mémoire disponible, j'ai implémenté cette structure comme une liste chainée, ou chaque maillon continent un entier, et un pointeur vers le maillon suivant.

```
struct maillon{
    int head;
    struct maillon * tail;
};

typedef struct maillon maillon;
typedef struct maillon* pile;

pile lapile;
```

Les fonctions de base sont celles permettant d'empiler et de dépiler un élément. Toutes les autres opérations sur cette pile, appelées lors de l'exécution sont rassemblées dans le module pile.

La grille

Il s'agit d'une structure contenant la hauteur, la largeur de la grille, ainsi qu'un char** contenu. Ce dernier champ un tableau de chaines de caractères représentant les différentes lignes du fichier .prog2d. Ainsi, on accède au caractère de coordonnées (x;y) avec `lagrille.contenu[x][y]` .

```
struct grille{
    int hauteur;
    int largeur;
    char** contenu;
};
typedef struct grille grille;

grille lagrille;
```

La variable lagrille est générée au début de l'exécution de l'interpréteur et du débogueur, par lecture du fichier passé en argument grâce à fgets (voir le début du main de interpreter.c ou debugger.c pour plus de détails).

Le curseur

Il s'agit d'une structure contenant les coordonnées x et y du curseur, sa direction (qui est un type énuméré), un champ `n_ignore` et un entier `end_programme`.

```
enum direction {nord=0 , ouest=6 , sud=4 , est=2, nord_ouest=7 ,
nord_est=1 , sud_ouest=5 , sud_est=3 };
typedef enum direction direction;
```

```c
struct curseur {
    int x;
    int y;
    direction direct;
    int n_ignore; /*Nombre de cases suivantes à ignorer, si -1,
    on ignore toutes les cases suivantes (jusqu'a rencontrer un
    " selon l'énoncé) */
    int end_programme;//Si passe à 1, on arrête le programme
};

typedef struct curseur curseur;

curseur lecurseur = init_curseur(); //Initialisation du curseur
```

Le champ `n_ignore` représente le nombre de cases suivantes à ignorer. Il est utilisé dans 2 cas:

- on rencontre un caractère # : n_ignore est initialisé à n (n étant dépilé depuis la pile) et est décrémenté jusqu'à 0. Dans ce cas, on ne fait que mettre à jour les coordonnées du curseur en tenant juste compte de sa direction, tant que n_ignore n'est pas revenu à 0.
- on rencontre un caractère " : n_ignore est initialisé à -1 et on empile tous les caractères rencontrés jusqu'à réobtenir un autre caractère ". A nouveau, la mise à jour des coordonnées du curseur ne se fait que par rapport à sa direction.

NB : la mise à jour des coordonées du curseur en ne tenant compte que de sa position est réalisé par l'appel à la fonction `curseur_update(curseur* c, int h, int l)` du module curseur.

Limites

- On suppose que le fichier passé en argument est correctement formaté.

Ainsi, on suppose que la première ligne est de la forme 'entier1 entier2'.

- On suppose que les dimensions sont inférieures à la taille max d'un int.

Les dimensions sont codées sur un type int, donc le programme ne fonctionnera pas correctement si la grille est de taille supérieure à la taille d'un int (dimensions négatives ou erronnées).

J'ai tenté de palier au problème en utilisant des entiers non signés (entiers positifs donc ayant une plus grande borne supérieure). Cependant, j'obtenais des avertissements à la compilation, lors de comparaison avec les entiers signés de la pile.

En se basant sur ce postulat, l'afficheur de la grille dans le débogueur ne peut afficher sans artefact qu'une grille dont les dimensions font au plus 10 caractères.

Pistes pour les questions bonus

Pour pouvoir revenir en arrière dans le débogueur, deux options sont possibles :

- Créer 3 modules gérant les piles de grilles, les piles de curseurs et les piles de piles.

Il faudrait alors gérer une pile de pile gérant l'historique de l'état de la pile aux différents moments de l'exécution

Cependant, cela implique beaucoup de malloc/free et donc potentiellement des performances non optimales. De plus, on stockerait la grille en un grand nombre d'exemplaires, alors qu'elle n'est modifiée que d'un caractère à la fois (il en va de même pour la pile ou le curseur, ou beaucoup de valeurs seraient dupliquées inutilement)

- Créer des structures `modification_pile` `modification_grille` et `modification_curseur`

```
struct modification_grille{
  int x;
  int y;
  char before;
  char after;
};

typedef struct modification_grille modification_grille;

struct modification_curseur{
  int x_prev;
  int y_prev;
  int direction_prev;
  int n_ingore_prev;
  int end_programme_prev;

  int x_next;
  int y_next;
  int direction_next;
  int n_ingore_next;
  int end_programme_next;
};

typedef struct modification_curseur modification_curseur;
```

```c
struct modification_pile{
  int depile1;
  int depile2;

  int empile1;
  int empile2;
}

typedef struct modification_pile modification_pile;
```

Il faudrait alors développer la panoplie de fonctions permettant de faire revenir en arrière le triplet `(lapile, lagrille, lecurseur` grâce à un triplet `(modification_pile, modification_grille, modification_curseur)`, ainsi que la gestion de piles de modifications pour les 3 types.

Cette approche fonctionne car on ne modifie qu'au plus 1 caractère à la fois (pour la grille) et qu'on ne dépile/empile qu'au plus 2 éléments à la fois pour la pile. Elle est plus complexe à implémenter mais consomme (beaucoup) moins de mémoire et est potentiellement plus rapide que la simple duplication (car n'effectue pas autant d'allocations).