Rapport Interpréteur Cornelis

Hermier Jules

January 8, 2024

Contents

Le projet consiste à réaliser un interpréteur pour le langage Cornelis, un langage ésotérique visuel permettant de représenter un programme comme un oeuvre d'art. Toutes les opérations se réalisent en parcourant l'image, les différences de couleurs et luminescences sur les pixels permettant de réaliser des actions sur une pile et de manipuler l'entrée standard.

1 Structure de l'interpréteur

1.1 Stack.c

Une structure de pile classique utilisée par les autres modules. Elle est codée à l'aide d'une liste chainée. La seule spécialité est la présence d'un entier représentant le nombre d'éléments afin de ne pas réaliser les opérations si l'on a pas assez d'éléments au préalable.

1.2 Operations.c

Une implémentation de chacune des opérations que Cornellis peut effectuer. Le module dépend donc de stack.c étant donné que le langage utilise une pile.

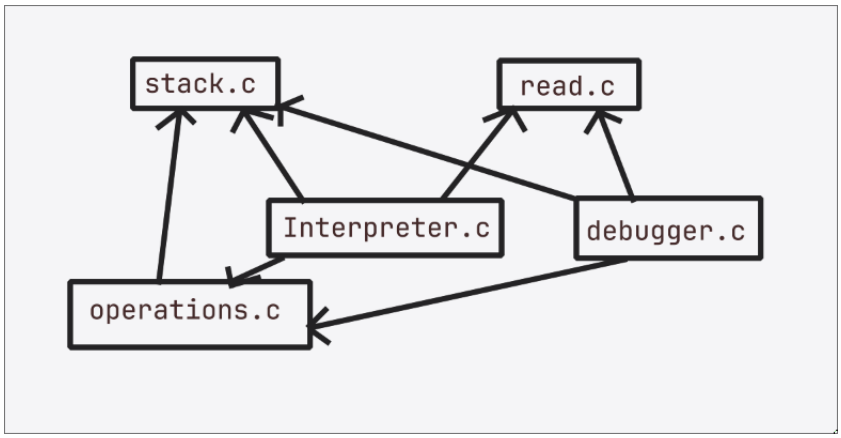

Figure 1: Graphe des dépendances de modules

1.3 read.c

Sert à lire l'image avant d'éxécuter le programme. Afin de pouvoir lire le plus de formats possible j'ai fait usage d'une librairie externe: la librairie de lecture d'image du projet stb. Le programme peut ainsi lire les fichiers png, bmp et encore d'autres en plus des ppm.

La structure choisie pour le stockage de l'image est un struct contenant la largeur, la hauteur et un tableau d'entier contenant le code couleur HTML du pixel. Le choix de stocker le code couleur a été fait car il est plus simple de manipuler une seule valeur pour les comparaisons, égalités, etc... plutôt que de devoir manipuler les 3 valeurs r, g et b (qui sont toujours accessible après conversion).

1.4 Interpreter.c

Le fichier principal gérant l'éxécution du programme. Il consiste d'une fonction "interprete" composé d'une boucle calculant le bloc suivant d'éxécution du programme et faisant l'opération associée si l'on n'est pas passé par un bloc passant.

Pour calculer le nouveau bloc, on suit les étapes données dans la description du langage.

Pour trouver le prochain pixel, avant d'avancer vers le nouveau bloc: j'ai décidé d'utilise un flood-fill auquel je passe en argument 3 fonctions de comparaison:

- Une vérifiant que c'est bien une frontière
- Deux permettant de comparer les coordonées de deux points afin de savoir si un est plus haut/bas/à droite/à gauche.

Ainsi on peut comparer les pixels et prendre celui correspondant le mieux au tableau d'action selon la direction/le bord.

On se sert aussi d'un flood-fill pour déterminer la taille d'un bloc pour l'opération empile.

1.5 Debugger.c

Consiste du même code que Interpreter.c majoritairement mais avec des fonctionnalités supplémentaires:

- Un affichage de l'état de l'interpréteur à chaque étape (direction, bord, pile...)
- Chaque opérations réalisées par l'interpréteurs
- Un affichage graphique du programme, le pixel actuel change de couleur à chaque frame.
- La possibilité d'éxécuter le code étape par étape (en pressant la touche flèche droite sur l'affichage graphique)
- La possibilité d'ajouter des breakpoints mettant l'éxécution en pause quand ils sont atteint.

Afin de gérer l'affichage graphique, j'ai fait usage de la librairie Raylib. Elle est peut-être un peu lourde pour l'échelle du programme mais j'ai choisi de l'utiliser par habitude.

2 Problèmes recontrées

Je n'ai pas rencontré de réel problème lors de l'implémentation. J'ai longuement réfléchi à calculer la taille et bordure de chaque bloc au préalable afin de rendre l'éxécution plus rapide avant de réaliser que la structure typique d'un programme en Coriellis est composé de chemins de pixels unique.

3 Limites du programme

- L'allocation compulsive de tableaux de booléen prend beaucoup de performance. Il serait plus logique de l'allouer au niveau de l'interpréteur et de juste réinitiliser les valeurs (il pourrait même être stocké dans la structure de l'image), Un autre flood fill pourrait être utilisé afin de ne pas parcourir l'entièreté du tableau pour le réinitialiser, ce qui serait plus efficace sur les gros programmes.
- Le stockage de l'entièreté du programme en RAM rend l'interpréteur inutilisable sur des énorme programmes.
- Le debugger ne permet pas de revenir à une étape antérieure.

4 Mode d'emploi

Le Makefile a trois target:

- (default) interpreter: l'interpréteur sans aucune fonctionalité supplémentaire. Ne requiert aucune librairies non communes (librairie standard, lib.h et math.h)
- debugger: l'interpréteur avec le visualiseur et toutes les options de debuggage listée plus haut. Requiert le pacquet Raylib disponnible sur la plupart des distributions unix. Le code source peut-être trouvé ici `https://github.com/raysan5/raylib/releases`.
- clean: supprime les fichiers intermédiaires

utilisation:

```
./interpreter <lien/vers/le/programme.ppm/png/bmp...>
./debugger <lien/vers/le/programme.ppm/png/bmp...>
```

debugger est muni d'une interface de configuration explicite. Lors de la visualisation étape par étape, la touche "flèche droite" permet de passer à l'étape suivant.