Rapport du Groupe 3

Calculabilité

Clément BUON, Tifanny NGUYEN, Fanny BACHEY

17 Janvier 2023

1 Création du gestionnaire mémoire

Dans le cadre du projet collectif, le groupe 3 se charge de la gestion et de l'allocation de la mémoire. Nous travaillons en étroite collaboration avec le groupe 2 sans qui nous ne pouvions pas réaliser notre tâche.

1.1 Fonctionnement du gestionnaire de mémoire

Nous avons organisé la mémoire en plusieurs scripts qui seront expliqués ci-dessous. Le script principal de notre module : groupe3_gestionMemoire.py.

Exemple d'input :

Ce module prend en input la sortie du groupe 2 : gestionnaire de nom de variable qui nous fournit un dictionnaire, qui va, à chaque numéro d'instruction, associer les membres gauches et droits d'une affectation. Cela nous permet de savoir quand une variable est créée ou supprimée. Ce qui nous permet de faire le traitement adéquate dans la mémoire.

```
affectations
{'1': {'D': '0', 'G': 'x'},
 '2': {'D': 'x + 1', 'G': 'x'},
 '3': {'D': 'x * 2', 'G': 'y'},
 '8': {'D': 'y', 'G': 'x'}}
suppressions
{'10': ['x', 'y']}
```

La première variable du programme est enregistrée en tant que constante. Ainsi, deux espaces mémoires seront alloués. Un premier pour le 0 qui sera une constante et une autre pour x, qui au début du programme a pour valeur 0 mais qui peut être soumis à des changements plus tard.

L'output se présente de la façon suivante, c'est un dictionnaire qui comprend l'adresse des variables à chacune des instructions mêmes si celles-ci ne sont pas importantes, certaines méthodes permettent de récupérer les adresses en mémoire sans avoir à imprimer toute la bande (voir ci-dessous dans la liste des méthodes disponibles) :

```
Étape n°1:
CONST_0 : adresse 0
x : adresse 32

Étape n°2:
CONST_0 : adresse 0
x : adresse 32
CONST_1 : adresse 64

Étape n°3:
CONST_0 : adresse 0
x : adresse 32
CONST_1 : adresse 64
CONST_2 : adresse 96
y : adresse 128
```

Les différents scripts et leur fonctionnements :

groupe3_gestionMemoire.py

Le script principal prend en entrée un dictionnaire d'affectation (et de suppression) : 'input_affectations' (qui nous est fourni par le groupe 2). On simule l'évolution de la mémoire à chaque instruction, on lit une première fois le dictionnaire pour mettre en mémoire les constantes et on le fait une seconde fois pour enregistrer les variables. Cela nous permet par la suite de renvoyer un dictionnaire avec l'état de la mémoire.
On initialise d'abord la mémoire pour ensuite itérer sur chaque instruction du programme dans la boucle principale et ainsi faire le traitement approprié. On enregistre ensuite l'état de la mémoire dans le dictionnaire avec l'indice correspondant à chaque étape. Et on retourne mémoire qui représente l'état de la mémoire à chaque étape du programme. Sachant que la bande de mémoire est bloquée à 3200, le reste est donc une mémoire vive.

memory_manager.py

On y retrouve les constantes et les variables !

1. Les constructeurs :

Il y a 2 constructeurs : un premier par défaut et un second qui prend en entrée un dictionnaire d'affectations (fournis par le groupe 2), pour initialiser la mémoire, on utilise la méthode 'initialiser_constantes' pour les constantes.

2. Les méthodes pour ajouter des constantes (+ méthode pour initialiser à partir d'affectation) et des variables (+ modifier et supprimer):

- add_constant(self, valeur: int): Ajoute une constante à la mémoire en écrivant sa valeur sur la bande.
- add_variable(self, nom: str, valeur: int): Ajoute une variable à la mémoire en écrivant sa valeur sur la bande.
- update_variable(self, nom: str, valeur: int): Met à jour la valeur d'une variable existante en mémoire.
- delete_variable(self, nom: str): Supprime une variable de la mémoire en écrivant des zéros à son adresse sur la bande.
- initialiser_constantes(self, input_affectations: Dict): Parse les membres droits des affectations pour récupérer des constantes et les charge en mémoire.

3. Les méthodes de lecture des valeurs déjà en mémoire :

- isInMemory(self, nom_variable: str) -> bool: Vérifie si une variable est présente en mémoire.
- readVariable(self, nom_variable: str) -> int: Renvoie la valeur d'une variable à partir de son nom.

- readConstant(self, valeur: int) -> int: Renvoie la valeur d'une constante à partir de sa valeur.

4. Méthode d'affichage et de gestion de la mémoire :
- afficher_memoire(self): Affiche toutes les variables et leurs adresses en mémoire.
- adresse_memoire_vive(self) -> int: Renvoie l'adresse du point de départ de la mémoire vive, après les constantes et les variables.

bande.py

C'est le script qui nous permet de modéliser la bande mémoire pour notre machine de Turing. Comme nous l'avions présenté, elle est composée d'une séquences de "cases" initialement vides. On maintient la positions courante et on propose plusieurs méthodes :

- afficher(self) : affiche le contenu de la bande.
- lire_position_courante(self) -> int: Lit la valeur (0 ou 1) à la position courante sur la bande.(qui est permise grâce à get_position_courante(self) juste avant.)
- Lire(self, adresse_variable) -> int: Lit une valeur binaire à partir d'une adresse spécifiée sur la bande. Cette méthode renvoie la valeur d'une variable en décimal : 1. se place sur la dernière case de la série de cases allouées à la variable 2. lit les cases en se déplaçant à gauche 32 fois 3. stocke chaque variable dans un tableau, et convertit le binaire en décimal
- ecrire_position_courante(self, valeur): Écrit la valeur (0 ou 1) à la position courante sur la bande.
- ecrire(self, valeur: int, destination: int): Écrit une valeur binaire (sous forme décimale) à une adresse spécifiée sur la bande. Précisions : Variable : la valeur à encoder en décimal destination = adresse à laquelle on va écrire notre variable 1. convertir en binaire 2. se placer à droite. écrire le 1er zéro. écrire le nombre. écrire des zéros jusqu'à la fin
- se_deplacer1(self, direction: str): Déplace la position courante d'une unité vers la droite ('D') ou la gauche ('G')
- se_deplacer(self, adresse: int): Déplace la position courante à une adresse spécifiée sur la bande.
- inverser_ecriture(self): Inverse le sens d'écriture des variables et constantes sur la bande.

variable.py

Permet de créer un objet variable à partir des informations qui nous sont fournies par le groupe 2. Cette classe a pour attributs adresse, nom et valeur ainsi que 3 méthodes :

1) affecter
2) comparer

3) get_adresse

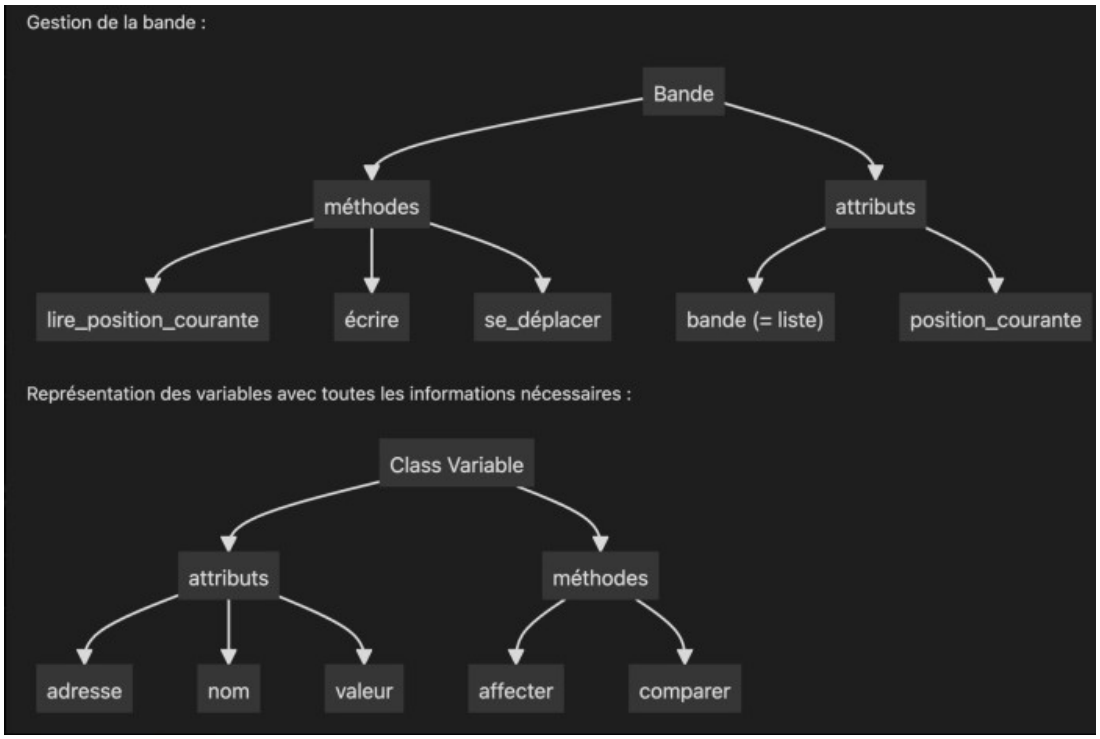
constante.py

C'est une classe qui hérite de la classe variable et qui permet de gérer les constantes.
operateurs.py

Ce script contient une méthode pour chacun des opérateurs si cela est nécessaire pour le groupe 4.

1.2 Conception du gestionnaire de mémoire

Dans un premier temps, nous avons modéliser notre mémoire comme une bande, qui permettait d'allouer des espaces mémoire pour les variables dont la valeur était égale ou inférieur à 29.
Voici l'architecture de notre mémoire pour la première présentation :



Il a été suggéré qu'une partie de la mémoire pouvait être utilisée pour stocker les constantes. Nous avons donc revu notre infrastructure pour les prendre en compte. Après avoir fourni une première version au groupe 4 qui se chargeait de la génération, il nous a été demandé s'il était possible de rajouter une mémoire vive. Ce qui a été fait également.

1.3 Défis rencontrés

Problème : La principale difficultée que nous avons rencontrée est la compréhension du sujet. En effet, il était difficile de faire la passerelle entre la théorie et le code. Et nous avons trouvé qu'une autre séance avant le rendu aurait permis de poser encore des questions et ainsi facilité l'avancée du projet, la séance des présentations n'ayant pas permis de pleinement cerner les attentes. Nous nous sommes sentis désemparés face aux connaissances nécessaires pour faire une mémoire. Nous n'avions en effet, jamais été confronté à ce genre d'exercice . Nos différents parcours n'ont pas été suffisant pour construire le socle théorique de ce projet. Même si la licence de mathématiques de Clément a été un atout non négligeable.

1.4 Répartition du travail

Au niveau de la répartition du travail, Tifanny et Fanny étaient chargées des présentations et du brainstorming sur le travail à faire. Clément s'ocuppe de l'infrastructure du module et de la création de toutes les méthodes nécessaires. C'est notre développeur leader. Fanny s'occupe de la rédaction des différents rapports corrigés par Tifanny.