MINISTÈRE
DE L'ÉDUCATION
NATIONALE,
DE L'ENSEIGNEMENT
SUPÉRIEUR
ET DE LA RECHERCHE

Liberté
Égalité
Fraternité

Concours externe BAC + 3 du CAPES

Cafep-Capes

Section Numérique et sciences informatiques

1) Exemple de sujet pour la seconde épreuve d'admissibilité
2) Extrait de l'arrêté du 17 avril 2025

Les épreuves des concours externes du Capes et du Cafep-Capes BAC +3 sont déterminées dans l'arrêté du 17 avril 2025 fixant les modalités d'organisation du concours externe du certificat d'aptitude au professorat de l'enseignement du second degré, publié au Journal Officiel du 19 avril 2025, qui fixe les modalités d'organisation du concours et décrit le schéma des épreuves.

Juin 2025

CAPES BAC +3

Sujet 0 / Seconde épreuve d'admissibilité

1 Exercice I : reconnaissance d'unités lexicales

Dans cet exercice, on s'intéresse à la reconnaissance d'unités lexicales dans un langage donné. Si on s'intéresse à un langage de programmation, les unités lexicales sont les mots du langage (c'est comme cela qu'ils seront d'ailleurs nommés dans la suite). Par exemple, dans un langage comme Python, on peut distinguer les mots-clefs qui ont une syntaxe fixe comme for, if ou encore def des autres unités lexicales qui, elles, sont de structure variable mais doivent respecter néanmoins une certaine syntaxe. On peut prendre l'exemple des identificateurs de variables qui, en python, doivent commencer par le caractère '_', une lettre majuscule ou une lettre minuscule puis être composée d'un nombre quelconque de chiffres, lettres majuscules, lettres minuscules et de caractères '_'.

La reconnaissance des lexèmes peut être effectuée grâce à un automate, comme cela est donné en Annexe A.1. La structure d'un programme Python pour faire ce travail a déjà été définie. Elle est donnée en Annexe A.2.

1.1 Automates et base de données

Question 1. Donner l'automate permettant de reconnaître les identificateurs de variables de python.

Question 2. Donner la liste des instructions SQL permettant d'insérer dans la base de donner les enregistrements nécessaires à la reconnaissance des mots-clefs `if` et `else` ainsi que les unités lexicales de type *identificateur de variable*.

Question 3. Étant donnée la structure de la base de données, en justifiant, répondre à chaque question de la liste suivante :

1. peut-on représenter un automate sans transition ?
2. peut-on décrire plusieurs états avec le même nom dans des automates associés à des mots différents ?
3. peut-on représenter des états qui seraient à la fois des états d'entrée et de sortie ?
4. peut-on décrire deux états de même nom dans le même automate ?
5. peut-on représenter des automates indéterministes ?
6. peut-on représenter des automates avec plusieurs état de départ ?

Question 4. Étant donnée la structure de la base de données, indiquer pour chaque dépendance fonctionnelle suivante si elle est vraie ou fausse en justifiant :

1. `id_etat` $\rightarrow$ `nom_mot, ismotclef`
2. `id_depart, id_arrivee` $\rightarrow$ `etiquette`
3. `id_depart` $\rightarrow$ `etiquette`
4. `id_depart, etiquette` $\rightarrow$ `id_arrivee`
5. `nom_etat` $\rightarrow$ `isentree`

Question 5. Pour chaque phrase suivante, donner une requête SQL permettant de renvoyer les éléments spécifiés :

1. nombre de mots décrits dans la base ;

2. nombre de mots-clef de la base ;
3. nombre d'états dans l'automate du mot "entier" ;
4. nombre d'états dans l'automate de chaque mot de la base ;
5. nombre d'états dans l'automate de chaque mot de la base ayant au moins 5 lettres ;
6. nombre d'états des automates qui ont autant d'états que le nombre de lettres du mot correspondant.

Question 6. Le schéma de la base de données utilisée ici ne permet pas de garantir totalement la cohérence de la représentation des automates associés aux unités lexicales. Donner un exemple d'insertion dans la base mettant en évidence une telle incohérence. Dans un deuxième temps, indiquer quel outil du monde des bases de données pourrait être utilisé pour éviter cette incohérence.

1.2 Programmation en Python

Une première structure de code a été réalisée en Python et est présentée en Annexe A.2.

Question 7. Donner la requête à utiliser dans chacune des méthodes de la classe `DAO`.

Question 8. Écrire le code des différentes méthodes de la classe `BaseReconnaissance`.

1.3 Conception et programmation orienée objet

On souhaite reprendre le programme d'analyse de lexèmes avec une conception plus modulaire. La trame de la nouvelle architecture a déjà été réalisée et est donnée sur la figure 2 en Annexe A.3.

Question 9. Compléter le diagramme de classes en faisant figurer les cardinalités

Question 10. Donner le code des classes `Sommet`, `Arc` et `Graphe` permettant de doter la classe `Graphe` de cette nouvelle version des mêmes fonctionnalités que la classe `Graphe` du programme Python donné en annexe 1.2 (en se restreignant au cas où les étiquettes sont de simples caractères).

Question 11. Proposer un nouveau diagramme de classe permettant d'avoir aussi bien des arcs étiquetés par de simples caractères que des arcs étiquetés par des mots-clefs comme "chiffre" et "lettre".

2 Exercice II : interpréteur pour un assembleur

Dans cet exercice, on s'intéresse à l'écriture d'un simulateur pour l'assembleur d'un micro-processeur
Le micro-processeur en question, le BUB25 est décrit en Annexe B.1.

2.1 Architecture et assembleur du processeur BUB25

Question 12. Proposer un schéma représentant l'architecture du processeur BUB25 et des composants auxquels il est connecté.

Question 13. Toutes les instructions possibles doivent pouvoir être représentées sur 64 bits. Donner, en le justifiant, la taille maximale de la mémoire adressable.

Question 14. Préciser, en le justifiant, si le mécanisme mis en œuvre pour les comparaisons entre les valeurs de 2 registres permet d'envisager tous les cas de figure ($=, \neq, <, \leq, >, \geq$).

Question 15. La documentation du micro-processeur BUB25 donnée en Annexe B.1 donne un exemple de programme, mais celui-ci n'est pas documenté. Expliquer ce que fait ce programme.

Question 16. Écrire un programme en assembleur BUB25 réalisant un programme qui transcrit en assembleur le programme Python suivant :

```
s = 0
for i in range(10):
  s += i
  if s % 2 == 0:
    print(s)
print(s)
```

2.2 Interpréteur en java pour l'assembleur

On souhaite écrire en Java un interpréteur pour cet assembleur. Une architecture préliminaire du logiciel en question a été conçue et est présentée en Annexe B.2.

Question 17. Préciser, en le justifiant, comment implanter l'association *instructions* figurant sur la figure 3.

Question 18. Le diagramme de classes proposé ne permet pas de représenter les étiquettes. Compléter ce schéma pour corriger ce manque.

Question 19. Les opérandes sont représentés par une interface. Justifier ce choix, et compléter le diagramme de classe avec les sous-types de l'interface Operande.

Question 20. Aucune variable d'instance ne figure dans la représentation de la classe Machine. Donner la liste des variables d'instances devant y figurer, en précisant le rôle et le type de chacune.

2.3 Interpréteur en Python

Le choix du langage pour l'écriture de l'interpréteur se porte finalement sur Python. On n'utilisera donc plus de type énuméré ni d'interface. Les instructions seront par ailleurs représentées par des classes spécifiques à chaque opérateur.

La classe `Machine` doit notamment être dotée d'une méthode `parser(fichier : list[string])` qui, à partir de la liste des lignes d'un code source en assembleur BUB25, crée la structure de données représentant le programme.

Question 21. Donner le code Python de la méthode `parser` (on suppose que le paramètre `fichier` représente un programme exempt d'erreurs).

Question 22. Préciser les types d'erreurs que la méthode parser peut détecter.

Dans les questions suivantes, on se place sur un sous-ensemble de l'assembleur réduit aux instructions `HLT` et `JSR #addr(reg)`.

Question 23. Donner le nouveau diagramme de classes correspondant à l'implantation Python, et complété des méthodes permettant d'exécuter le programme (on supposera que l'exécution effectuée par une méthode `executer()` définie dans la classe `Machine`).

Question 24. Donner le code des méthodes introduites à la question précédente.

2.4 Système multi-tâches

On souhaite maintenant permettre de modéliser le fonctionnement du microprocesseur BUB25 dans un environnement multi-tâches préemptif, comme décrit en Annexe B.3. Dans le modèle envisagé, le processus élu se voit alloué un quantum de temps d'une durée fixe.

Question 25. La durée d'exécution d'une instruction est évaluée en "nombre de cycles" : le nombre d'activations d'unités élémentaires du processeur. Déterminez les paramètres qui influent sur le nombre de cycles nécessaires à l'exécution d'une instruction et donner un exemple d'instruction "brève" et un exemple d'instruction "longue".

Question 26. Préciser ce qui doit être rajouté sur le diagramme de classe pour pouvoir modéliser une machine multi-tâches (structure de processus avec toutes les informations utiles aussi bien au système pour gérer les processus qu'au processus lui-même).

Question 27. Lister les instructions pouvant mener à une suspension anticipée d'un processus (c'est-à-dire avant que le nombre de cycles alloué au processus ne soit atteint).

Question 28. Préciser les mécanismes à mettre en œuvre pour éviter que les actions d'un processus ne puissent interagir sur le fonctionnement d'un autre processus.

Annexe A Documents de l'exercice 1

Annexe A.1 Reconnaissance des mots d'un langage rationnel par un automate

Pour reconnaître les mots d'un langage, on peut utiliser des automates. Un automate $A(E, V, i, o)$ est un graphe étiqueté où :

— E désigne l'ensemble des états (sommets du graphe)de l'automate ;
— $V \in E \times E \times \Sigma$ désigne l'ensemble des transitions (arêtes du graphes). Σ est l'alphabet associé à l'automate ;
— $i \in E$ est l'ensemble des états d'entrée ;
— $o \in E$ est l'ensemble des états de sortie.

L'automate reconnaît un mot $m = w_1 w_2 \dots w_n$ (où $w_i \in \Sigma$) s'il existe un chemin menant d'un état d'entrée à un état de sortie dont les transitions successives sont étiquetées par les lettres w_i successives de m.

Les langages rationnels sont les langages pour lesquels il existe un automate permettant de reconnaître les mots d'un langage.

Par la suite, on ne s'intéressera qu'aux automates déterministes. Un automate est dit déterministe s'il vérifie les 2 conditions suivantes :

— il n'a qu'un seul état d'entrée ;
— pour tout caractère c et état e, il existe au plus une transition issue de e et étiquetée par c.

Les langages de programmation classiques ne sont pas des langages rationnels, mais leur analyse nécessite de commencer par reconnaître les unités lexicales (lexèmes) du langage. Les unités lexicales sont notamment les mot-clefs du langage, les séparateurs (virgule, parenthèses, etc.), les identifiants (noms de variables, de fonctions, etc.) et les constantes (nombres, chaînes de caractères, etc.). La figure 1 donne un exemple d'automate permettant de reconnaître les nombres naturels.

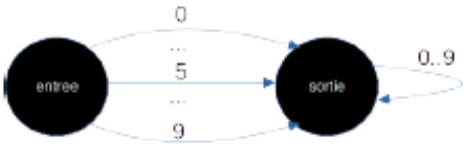

FIGURE 1 – Automate permettant de reconnaître les naturels

NB : dans cet automate, la transition étiquetée 0..9 symbolise en réalité 10 transitions, chacune étant étiquetée avec un chiffre différent de l'intervalle [0,9].

Dans la suite, on suppose que chiffre et lettre sont des étiquettes spéciales permettant de reconnaître respectivement tout chiffre ou toute lettre.

Annexe A.2 Descriptif du programme Python existant

Un interprète générique en Python a été écrit pour pouvoir analyser n'importe quel langage. Cela passe notamment par la reconnaissance des lexèmes du langage cible (par exemple, *extends* est un mot-clef de Java, mais pas de Python. De même *'texte'* désigne une chaîne de caractères en Python, pas en Java).

Pour ce faire, la configuration de l'interpète est effectuée via une base de données permettant de décrire aussi bien les mots-clefs du langage que les constantes et identifiants valides.

La base de données en question a été créée par la suite d'instructions SQL suivante :

```
create table mot (
        id_mot int primary key,
        nom_mot text NOT NULL,
        ismotclef int CHECK (ismotclef in (0,1)),
        UNIQUE(nom_mot)
);
create table etat (
        id_etat int primary key,
        nom_etat text,
        id_mot NOT NULL REFERENCES mot(id_mot),
        isentree int NOT NULL CHECK (isentree in (0,1)),
        issortie int NOT NULL CHECK (issortie in (0,1)),
```

```
        UNIQUE(nom_etat,id_mot)
);
create table transition (
        id_trans int primary key,
        id_depart NOT NULL REFERENCES etat(id_etat),
        id_arrivee NOT NULL REFERENCES etat(id_etat),
        etiquette text NOT NULL,
        UNIQUE(id_depart,etiquette)
);
```

Ainsi, le mot-clef *class* sera crée avec une simple entrée dans la table mot, tandis que les noms de variables seront définis par une entrée dans la table mot et plusieurs entrées dans les tables etat et transition, ces entrées permettant de définir alors l'automate permettant de reconnaître les noms de variables.

Les étiquettes des transitions peuvent correspondre soit à un caractère unique, soit à un des mots-clefs suivant : `chiffre` (représentant tout chiffre de 0 à 9) et `lettre` (représentant toute lettre, minuscule ou majuscule, éventuellemnt accentuée).

Les squelette des classes `Graphe`, `Automate`, `DAO` et `BaseReconnaissance` de l'interprète écrit en Python sont données ci-dessous.

```
# representation d'un graphe par un dictionnaire
# exemple : l'automate permettant de reconnaitre les entiers
# peut etre représenté par le dictionnaire suivant :
# {"entree": {"chiffre":"sortie"}, "sortie": {"chiffre":"sortie"}}
# les clefs sont les noms des état départ d'au moins un arc
# les valeurs sont des dictionnaire contenant des
# couples (nom etat cible, etiquette arc)
# NB : dans cette implantation, si un sommet figure dans le
# dictionnaire alors c'est qu'il existe
# au moins un arc dans le graphe partant de ce sommet.
# autrement dit, le dictionnaire des arcs associés à un
# sommet ne peut pas être vide

class Graphe :
    def __init__(self) :
        '''
        L'ensemble des arcs du graphe est vide au départ
        '''
        self.arcs = {}

    def add_arc(self,s1:any,s2:any,etiquette:str)->None:
        '''
        Ajout d'un arc dans le graphe
        :param s1: sommet de départ de l'arc
        :param s2: sommet d'arrivee de l'arc
        :param etiquette: etiquette de l'arc
        '''

    def get_sommets_source(self)->set[any]:
        '''
        renvoie l'ensemble des états du graphe dont part
        au moins une transition.
        renvoie un ensemble vide si aucun arc n'a été ajouté
        dans le graphe
        :return: liste des états concernés
        '''

    def get_all_sommets(self)->set[any]:
        '''
        renvoie l'ensemble de tous les sommets du graphe
        renvoie un ensemble vide si aucun arc n'a été ajouté au graphe
        :return: liste des sommets du graphe
        '''
```

```
    def get_sommet_arrivee(self, sl:any, car:str, categ:str)->any:
        '''
        renvoie le sommet d'arrivée de l'arc partant de sl et
        étiqueté par car. Si cet arc n'existe pas, on recherche un
        arc partant de sl et étiqueté par categ.
        Sinon on renvoie None
        :param sl: sommet de départ de l'arc
        :param car: étiquette de l'arc
        :param categ: catégorie associée à car
        :return: voir au dessus
        '''

    def is_vide(self):
        '''
        :return: vrai si le graphe est vide (aucun arc)
        '''

    def is_etat_source(self, s:any)->bool:
        '''
        :param s: un sommet du graphe
        :return: vrai si le graphe contient au moins un arc partant de s
        '''

    def is_arc(self, sl:any, etiquette:str)->bool:
        '''
        renvoie vrai s'il existe un arc partant de sl étiqueté par
        etiquette.
        renvoie faux si le graphe est vide, ou si sl n'est pas un
        sommet source du graphe, ou si aucun arc etiquetée etiquette
        ne part de sl.
        :param sl: un sommet
        :param etiquette: étiquette d'arc
        :return: voir au dessus
        '''

    def au_moins_un_arc(self):
        '''
        :return: vrai si le graphe contient au moins un arc
        '''

from Graphe import *

# définition d'un automate à partir d'un graphe existant
class Automate:
    def __init__(self, graphe:Graphe, nentree:any, lsortie:list[any]):
        '''
        Construction d'un automate
        :param graphe: le graphe représentant l'automate
        :param nentree: le nom de l'état d'entrée
        :param lsortie: liste des noms des état de sortie
        '''

        self.graphe = graphe
        ens_etats = graphe.get_all_sommets()
        if not (nentree in ens_etats):
            raise Exception("Etat␣entrant␣n'appartient␣pas␣au␣graphe␣!")
        self.entree = nentree  # état d'entrée
        for etat in lsortie:
            if not (etat in ens_etats):
                raise Exception("Un␣état␣sortant␣n'appartient␣pas␣au␣graphe␣!")
        self.sortie = lsortie  # liste des états de sortie
```

```
    def reconnaitre(self,ch:str)->bool:
        '''
        :param ch: une chaine à reconnaitre
        :return: vrai si ch est reconnu par l'automate
        '''

    def get_categorie(self, car:str)->str:
        '''
        :param car: un caractère
        :return: la catégorie du caractère : chiffre, lettre ou car
        NB : la catégorie car correspond aux caractères qui ne sont ni
        des chiffres ni des lettres
        '''

    def is_etat_final(self, etat:any)->bool:
        '''
        :param etat: un nom d'état
        :return: True si c'est un état final de l'automate
        '''

from GrapheEns import *
from Automate import *
import sqlite3

# classe permettant de gérer les accès à la base de données
class DAO:
    def __init__(self, nom_base:str):
        '''
        permet d'établir une connexion à la base de données sqlite
        :param nom_base: nom complet du fichier sqlite
        '''
        self.connexion = sqlite3.connect(nom_base)
        self.connexion.execute('PRAGMA foreign_keys = ON')

    def get_mots(self)->list[str]:
        '''
        :return: liste des mots de la base
        '''

    def get_motsclef(self)->list[str]:
        '''
        :return: liste des mots-clef de la base
        '''

    def get_etats(self, mot:str)->list[str]:
        '''
        :param mot: un mot de la base
        :return: la liste des noms d'états de l'automate associé à mot
        '''

    def get_entree(self, mot:str)->str:
        '''
        :param mot: un mot de la base
        :return: nom de l'état d'entrée de l'automate associé à mot
        '''

    def get_sortie(self, mot:str)->list[str]:
        '''
        :param mot: un mot de la base
        :return: liste des noms des états de sortie de l'automate associé à mot
        '''
```

```
    def get_transitions(self, mot:str, etat:str)->list[tuple[str, str]]:
        '''
        :param mot: un mot de la base
        :param etat: un nom d'état de l'automate de mot
        :return: la liste des transitions partant de etat dans l'automate de mot.
        Chaque transition est décrite par un tuple (nom etat arrivée, etiquette)
        '''

from DAO import *

class BaseReconnaissance :
    '''
    Classe permettant de stocker tous les mots clefs et les automates de la base
    '''
    def __init__(self, nom_base:str):
        '''
        Crée un objet permettant de gérer la reconnaissance de mots :
        - dao : objet permettant le dialogue avec la base de données
        - liMotsClefs : liste des mots-clefs de la base de données
        - base : dictionnaire des mots avec leur automate
        :param nom_base: nom complet du fichier contenant la base de données sqlite
        '''
        self.dao = DAO(nom_base)
        # dictionnaire des automates
        # clef = mot non mot-clef
        # valeur = automate
        self.li_mots_clef = self.get_mots_clef()
        self.base = self.get_automates()

    def is_mot_clef(self, ch:str)->bool:
        '''
        renvoie vrai si ch est reconnu comme un mot-clef
        :param ch: la chaine à reconnaitre
        :return: true si ch est un mot-clef
        '''

    def reconnaitre(self,ch:str)->str:
        '''
        reconnaissance de ch par un automate de la base
        :param ch: chaine de car à reconnaitre
        :return:
            - None : si ch n'est pas reconnu
            - ch : si ch est reconnue par l'automate associé à mot
        '''

    def get_mots_clef(self)->list[str]:
        '''
        :return: la liste des mots clef de la base
        '''

    def get_automates(self)->dict[str, Automate]:
        '''
        permet de constuire l'automate associé à chaque mot de la base
        :return:
        '''

    def get_automate(self, mot:str)->Automate:
        '''
        Construit à partir de la bdd l'automate associé à un mot
        NB : Le mot est censé exister dans la base
        :param mot: un mot de la base
```

```
        :return: l'automate associé au mot
        '''

    def ajouter_transitions(self, etat:str, li_trans_etat:list[tuple[str, str]],
                            graphe:Graphe)->None:
        '''
        permet d'ajouter dans graphe tous les arcs partant de l'état etat
        et décrits par la liste li_trans_etat
        :param etat: nom de l'état source
        :param li_trans_etat: liste de tuples (nom etat cible, etiquette transition)
        :param graphe: graphe à mettre à jour
        :return: rien
        '''
```

Annexe A.3 Base du diagramme de classes de la nouvelle version

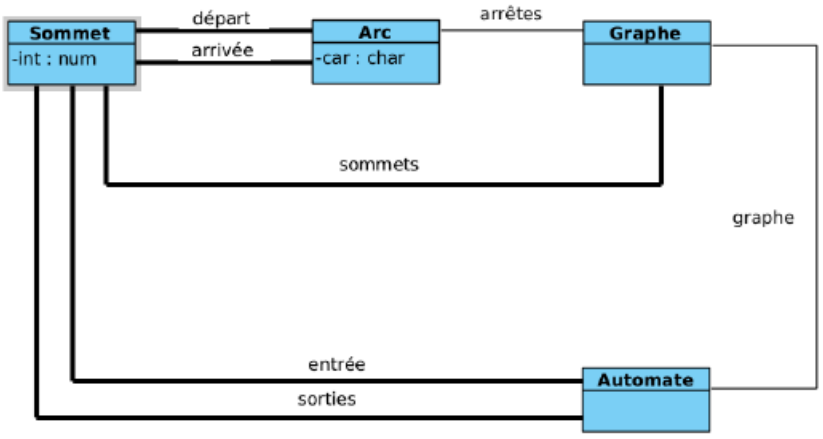

FIGURE 2 – Diagramme de classes de l'analyseur lexical

Annexe B Interpréteur pour l'assembleur BUB25

Annexe B.1 Description du micro-processeur BUB25

Architecture

Le micro-processeur BUB25 est un micro-processeur 64 bits proposant la liste de registres suivante :

- `R0`... `R7` : des registres multi-fonction ;
- `SP` : le registre indiquant le sommet de pile ;
- `PC` : compteur ordinal (adresse de la prochaine instruction à exécuter) ;
- `LR` : adresse de retour (voir plus loin) ;
- `OB` : registre dédié aux entrées-sorties : toute écriture dans ce registre se traduit par l'affichage de la valeur décimale de ce registre sur le périphérique de sortie connecté au micro-processeur, toute lecture de la valeur de ce registre se traduit par l'attente d'une valeur décimale sur le périphérique d'entrée connecté au micro-processeur ;
- `IOC` : registre dédié aux entrées-sorties : fonctionne comme IOB, mais en mode caractère : c'est le caractère dont on donne le code ASCII qui est écrit, et c'est le code ASCII du caractère tapé qui est lu.

Ce micro-processeur est doté d'une unité arithmétique et logique pour tous les calculs. Il dispose aussi des drapeaux (bits) suivant :

- `z` : si la dernière comparaison a constaté une égalité. Contient aussi le bit "sorti" en cas de décalage ;
- `n` : si lors de la dernière comparaison, le premier opérande était plus petit que le deuxième.

Jeu d'instructions

Le jeu d'instructions du processeur, toutes codables sur 64 bits, est le suivant :

- `STO #val, reg` : stocke la valeur `val` dans le registre `reg`
- `STO reg, #addr` : stocke la valeur contenue dans le registre `reg` à l'adresse `addr`
- `STO reg1, #addr(reg2)` : stocke la valeur contenue dans le registre `reg1` à l'adresse `addr + reg2`
- `LOD #addr, reg` : charge dans le registre `reg` le contenu situé à l'adresse `addr`
- `LOD #addr(reg1), reg2` : charge dans le registre `reg2` le contenu situé à l'adresse `addr + reg`
- `CPY reg1, reg2` : copie la valeur du registre `reg1` dans le registre `reg2`
- `ADD reg1, reg2` : ajoute au registre `reg2` la valeur contenue dans le registre `reg1`
- `SUB reg1, reg2` : soustrait au registre `reg2` la valeur contenue dans le registre `reg1`
- `MUL reg1, reg2` : multiplie le contenu de `reg2` par la valeur contenue dans le registre `reg1`
- `DIV reg1, reg2` : divise le contenu de `reg2` par la valeur contenue dans le registre `reg1`
- `SHL reg` : décale d'un bit vers la gauche le contenu du registre `reg`. La valeur du bit le plus à gauche avant décalage (bit "sorti") est stockée dans le drapeau `z`.
- `SHR reg` : décale d'un bit vers la droite le contenu du registre `reg`. La valeur du bit le plus à droite avant décalage (bit "sorti") est stockée dans le drapeau `z`.
- `JMP #addr` : indique un saut dans le programme à l'adresse `addr`
- `JSR #addr` : indique un saut dans le programme à l'adresse `addr`, en mémorisant l'adresse de l'instruction suivante comme adresse de retour dans le registre `LR`.
- `RTS` : revient à l'adresse mémorisée comme adresse de retour dans le registre `LR`.
- `CMP #val, reg` : compare la valeur `val` et la valeur contenu dans `reg` et met à jour les drapeaux en fonction du résultat de la comparaison
- `CMP reg, #val` : compare la valeur `val` et la valeur contenu dans `reg` et met à jour les drapeaux en fonction du résultat de la comparaison
- `CMP reg1, reg2` : compare les valeur de `reg1` et `reg2` et met à jour les drapeaux en fonction du résultat de la comparaison
- `BEQ #addr` : branchement à `addr` si la dernière comparaison a mené à une égalité
- `BGT #addr` : branchement à `addr` si lors de la dernière comparaison, le premier opérande était supérieur au deuxième
- `BGE #addr` : branchement à `addr` si lors de la dernière comparaison, le premier opérande était supérieur ou égale au deuxième.
- `PSH reg` : empile la valeur contenue dans le registre `reg` (et augmente donc la valeur du registre `SP`).
- `POP reg` : dépile le sommet de la pile et le stocke dans le registre `reg` (et diminue donc la valeur du registre `SP`).

— `HLT` : arrêt du programme
— `DAT #val` : il ne s'agit pas d'un instruction. Cela consiste juste à préciser qu'à l'adresse courante du programme, la mémoire contient l'octet `val` (on peut aussi mettre directement un caractère entre apostrophes pour représenter son code ASCII).

Implantation

Le processeur BUB25 est un processeur embarquée sur une carte avec une mémoire de 2000 octets. Lorsqu'un programme est chargé, il l'est à partir de l'adresse 0. Par ailleurs, à l'issue du chargement du programme, le sommet de pile est initialisé à 1000 et tous les autres registres, y compris le compteur ordinal, sont initialisés à 0.

Assembleur

L'assembleur permettant d'écrire des programmes pour le micro-processeur BUB25 permet d'utiliser une étiquette devant une instruction inst1 (l'étiquette peut être sur la même ligne ou sur la ligne précédente). Cette étiquette peut être utilisée à la place d'un paramètre `#addr` d'une autre instruction inst2 pour désigner l'adresse de l'instruction inst1. Lorsqu'on utilise une étiquette pour étiqueter une adresse, celle-ci doit forcément figurer sur la première colonne d'une ligne. De plus, son nom doit forcément être suivi du caractère " :". Les instructions, quant à elles, ne peuvent commencer sur la première colonne.

Voici un exemple de programme écrit dans l'assembleur du BUB25 :

```
debut:  STO #1, R0
        JSR addition
        CPY R0, IOB
        HLT
addition:
        LOD donnees, R1
        ADD R1, R0
        RTS
donnees:
        DAT #2
```

Ce programme charge la valeur `1` dans `R0`, puis la valeur `2` dans `R1`. Il ajoute ensuite le contenu de de `R1` (`2`) au contenu de `R0` (`1`), qui contient alors la valeur `3`, puis affiche cette valeur sur le périphérique de sortie.

Exemple de programme en assembleur BUB25

```
debut :
        JSR traitement1
        JSR traitement2
        HLT
traitement1:
        LOD 1, R2
        LOD donnees(R0), R1
        CMP R1, #0
        BEQ fin1
        PSH R1
        ADD R2, R0
        ADD R2, R3
        JMP traitement1
fin1:
        RTS
traitement2:
        CMP R3, 0
        BEQ fin2
        POP IOC
        SUB 1, R3
        JMP traitement2
```

```
fin2:
        RTS
donnees:
        DAT 'I'
        DAT 'L'
        DAT 'O'
        DAT 'J'
        DAT #0
```

Annexe B.2 Structure préliminaire de l'interpréteur

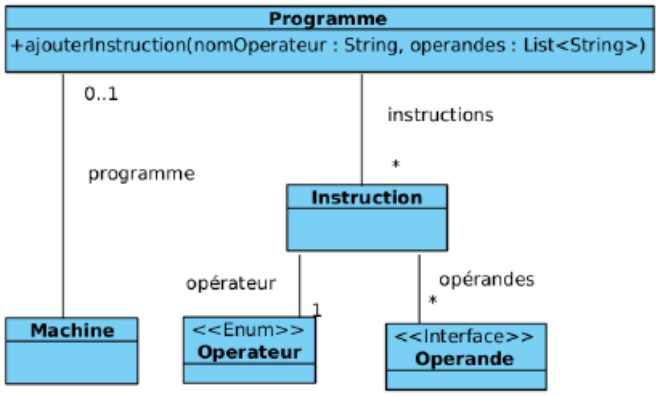

FIGURE 3 – Diagramme de classes de l'interpéteur

Annexe B.3 Multi-tâche préemptif

On rappelle que dans un système multi-tâche préemptif, le système partage le temps du processeur entre les différents processus en cours d'exécution. Le système change de processus au bout d'un certain temps, ou bien lorsque le processus est en attente sur une ressource. À chaque changement du processus courant, un changement de contexte global (notamment les valeurs des registres du micro-processeur) doit avoir lieu.

Réglementation de la seconde épreuve d’admissibilité

Extrait de l’annexe de l’arrêté du 17 avril 2025 fixant les modalités d'organisation du concours externe du certificat d'aptitude au professorat de l'enseignement du second degré, publié au Journal Officiel du 19 avril 2025

A. – Epreuves d’admissibilité

2° Seconde épreuve d'admissibilité.

L'épreuve s'appuie sur un ou plusieurs documents présentant une problématique posée par une situation concrète et des éléments de mise en œuvre d'une ou plusieurs solutions.
Le candidat est invité à analyser, le cas échéant, à compléter et à faire évoluer la ou les solutions à mettre en œuvre pour résoudre la problématique posée.
L'épreuve vise à apprécier la capacité du candidat à comprendre et analyser les documents proposés, à construire et exposer un raisonnement.

Durée : quatre heures.

Coefficient 2.

L'épreuve est notée sur 20. Une note globale égale ou inférieure à 5 est éliminatoire.