UPSimulator

Maxence KLEIN Véronique REYNAUD Guillaume DESJOUIS

2020

Table des matières

1 Présentation générale du projet

1.1.0 Objectifs

Le projet UPSIMULATOR a pour objectif de développer un simulateur de processeur à visée pédagogique. Celui-ci doit permettre d'appréhender la chaine conduisant d'un programme écrit dans un langage de haut niveau au détail de l'exécution à l'échelle du processeur. Pour cela, le projet doit permettre :

- la production d'un code source dans un langage jouet ;
- la compilation du code source et la production d'une version assembleur et binaire de celui-ci. Le simulateur doit permettre l'usage de différents modèles (taille des mots binaires, nombre de registre, ...) ;
- le suivi de l'exécution (registres, mémoire, pointeur, appels à l'UAL,...) ;

Les choix techniques retenus pour chaque fonctionnalités sont développés ci-après.

1.2.0 Structure

diag UML ou équivalent

2 Choix techniques

2.1.0 Langage jouet

Le langage jouet doit permettre à l'utilisateur de produire un exemple de code simple reprenant les principales structures (boucles, branchements conditionnels,...)

```
s = 0
i = 0
m = input()
while i < m:
    if i % 2 == 0:
        s = s + i
        if s < 10:
            y = s * 2
            print(y)
    i = i + 3
print(s)
```

Listing 1 – Exemple de code dans le langage jouet

Expressions admissibles

Les expression admissibles sont présentées dans la table 1 ci-dessous.

TABLE 1 – Expressions admissibles

Variable		`x`
Entier		`n`
Opérations arithmétiques	Somme	`e1 + e2`
	Différence	`e1 - e2`
	Produit	`e1 * e2`
	Division entière	`e1 / e2`
	Reste	`e1`
	Opposé	`-e1`

Opérations logiques		
Binaires	Egalité	`e1 == e2`
	Différence	`e1 != e2`
	Inégalités	`e1 < e2`
		`e1 > e2`
		`e1 <= e2`
		`e1 >= e2`
	Et	`e1 and e2`
		`e1 \& e2`
	Ou	`e1 or e2`
		`e1 | e2`
Unaire	inverse bit à bit	`~e1`
	négation logique	`not e1`

Liste de commandes admissibles

Affectation

```
x=e
```

avec `e` une expression logique ou arithmétique.

Branchement conditionnel

```
if e :
        c1
elif e2:
        c2
else:
        c3
```

avec `e1` et `e2` des expressions et `c1`, `c2` et `c3` des commandes.

Les branchement `else` et `elif` sont optionnels.

Boucle

```
while e :
        c1
```

avec `e` une expression et `c` une commande.

Lecture clavier

```
input()
```

Ecriture sur la sortie courante

```
print(v)
```

avec `e` une expression

Indentations

Le code est indenté comme en python afin de détecter les blocs :

- L'indentation n'augmente qu'après un `:` lié à une structure `if` ou `while`
- L'indentation ne peut diminuer que atteindre un niveau précédemment atteint.

Commentaires

Les commentaires sont repérés par le caractère `##` .

```
aaaaaaaaaa:
  aaaaaaaaaa
  aaaaaaaaaa
    aaaaaaaaaa # pas valable il manque : avant
    aaaaaaaaaa
  aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa:
      aaaaaaaaaa
  aaaaaaaaaa # pas valable. Ce niveau a été atteint avant
  ↪  mais pas dans le même bloc
aaaaaaaaaa
```

Listing 2 – Langage jouet - Commentaires et indentations

2.2.0 Parsing

Une étape d'analyse du code (parsing) est nécessaire en amont de la production du code assembleur. Cette étape a pour objet :

- d'assurer que la syntaxe du langage jouet est respectée
- de permettre la construction d'un arbre représentant les différentes structures du code source afin de pouvoir produire le code assembleur et le binaire associé

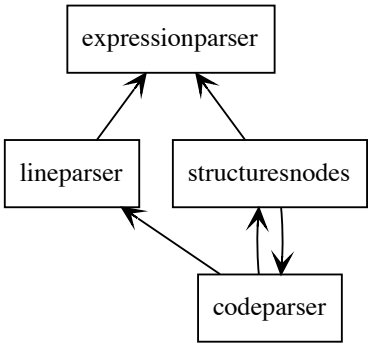

FIGURE 1

```
x=0
y=0
while (x < 10 or y < 1500):
    x=x+1
    y=y+x
print(y)
```

CodeParser →

```
[<structuresnodes.AffectationNode >,
<structuresnodes.AffectationNode >,
<structuresnodes.WhileNode>,
<structuresnodes.PrintNode>]
```

FIGURE 2 – Exemple simpliste de parse

Classe CodeParser

L'analyse du code est gérée par un objet de la classe `CodeParser` dont le constructeur prend en argument :

- soit un nom de fichier `filename = file`
- soit une chaine de caractère contenant un fragment de code `code = fragment`

Un objet de type `CodeParser` a pour attributs :

- `__listingCode` : une liste d'objets de type `LineParser`
- `__structuredListeNode` un arbre d'objets de type StructureNode contenant le code interprété

Lorsque le code est donné sous forme de fichier, la méthode `__parseFile` permet de récupérer la chaîne de caractères correspondante.

La méthode `parseCode` construit une instance de la classe `LineParser` pour chaque ligne de code source. Si la ligne n'est pas vide, les caractéristiques de celles-ci sont ajoutées à la liste `__listingCode`.

Une analyse syntaxique succincte est réalisée avec l'appel successif aux méthodes :

- `__manageElif` : réécriture des branchements `elif`).

```
if e1 :
        c1
elif e2 :
        c2
else c3
```

__manageElif →

```
if e1 :
        c1
else :
        if e2 :
                c2
        else :
                c3
```

- `__blocControl` : test de la syntaxe des structures de contrôle et de l'indentation associée.

Finalement, la construction de l'arbre `__structuredListeNode` nécessite l'appel des méthodes :

- `__buildFinalNodeList()` : construit les nœuds (instances de classe `structuresnodes`) et l'arborescence correspondante à partir des caractéristiques `__listingCode`. Les blocs d'instructions sont ajoutés à `__structuredListeNode`.
- `__structureList` : Parcours du listing `__listingCode` pour ranger les enfants et leur associer le bon niveau d'indentation

L'arborescence des nœuds `__listingCode` peut-être affichée à l'aide des méthodes des `__str__` et `__recursiveStringifyLine`.

L'accès à la liste de nœuds `__structureList` est possible à l'aide de l'accesseur `getFinalParse`.

Classe LineParser

La classe `LineParser` permet de renvoyer les caractéristiques d'une ligne de code sous forme d'un dictionnaire contenant numéro de ligne, niveau d'indentation, caractère vide ou non, motif identifié (if,....), condition, expression ou variable le cas échéant.

Pour une ligne de code donnée elle doit :

- Nettoyer le code des commentaires et espaces terminaux : `__suppCommentsAndEndSpaces`
- Déterminer le niveau d'indentation : `__countIndentation`
- Pour les lignes non vides, identifier le motif : `__identificationMotif`

Lorsque le motif correspond à un branchement conditionnel `if e` ou une boucle `while e` l'identification du motif `__identificationMotif` nécessite de tester que `e` est une expression valide. L'expression correspondante est construite par une instance de la classe `ExpressionParser`.

Classe ExpressionParser

Les objets de la classe `ExpressionParser` permettent l'interprétation d'une chaine de caractère afin de renvoyer un objet de type expression, c'est à dire un arbre dont chaque nœud représente un opérateur binaire, un opérateur unaire, une variable ou un littéral.

Pour cela la chaine de caractère représentant l'expression est convertie en une liste de Tokens (`__buildTokensList(cls, expression:str)`) représentant chaque type admissible dans la chaine de caractère

La classe doit permettre de vérifier la syntaxe de l'expression

2.3.0 Code Assembleur

2.4.0 Modèle de processeur

2.5.0 Interface utilisateur

2.6.0 Gestion de la documentation

3 Organisation

3.1.0 Planification

3.2.0 Répartition des tâches

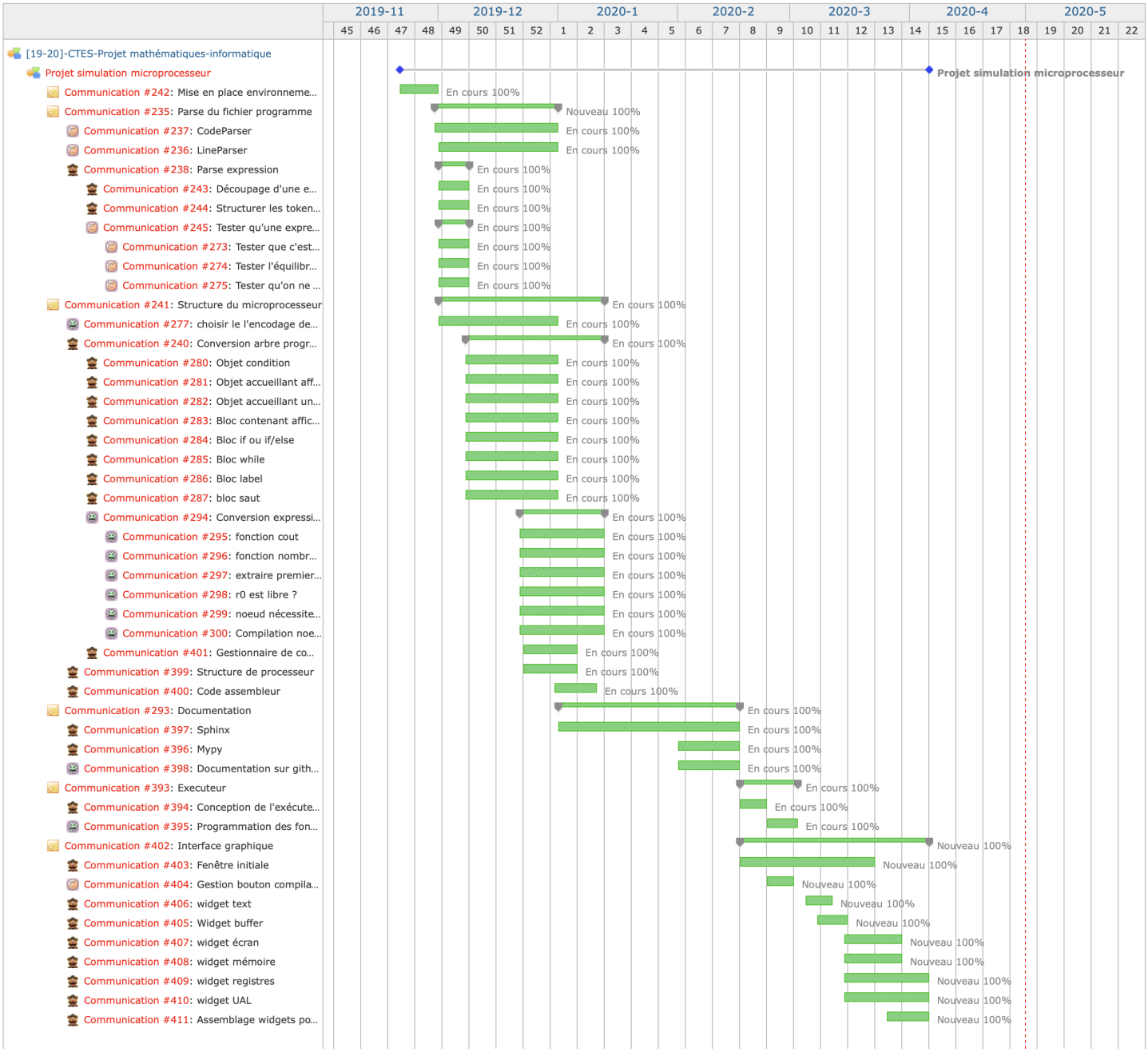

FIGURE 3 – Diagramme de Gantt du projet

Table des figures

Liste des tableaux