UPSimulator

Maxence KLEIN Véronique REYNAUD Guillaume DESJOUIS

2020

Table des matières

1 Présentation générale du projet

1.1 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 fonctionnalité sont développés ci-après.

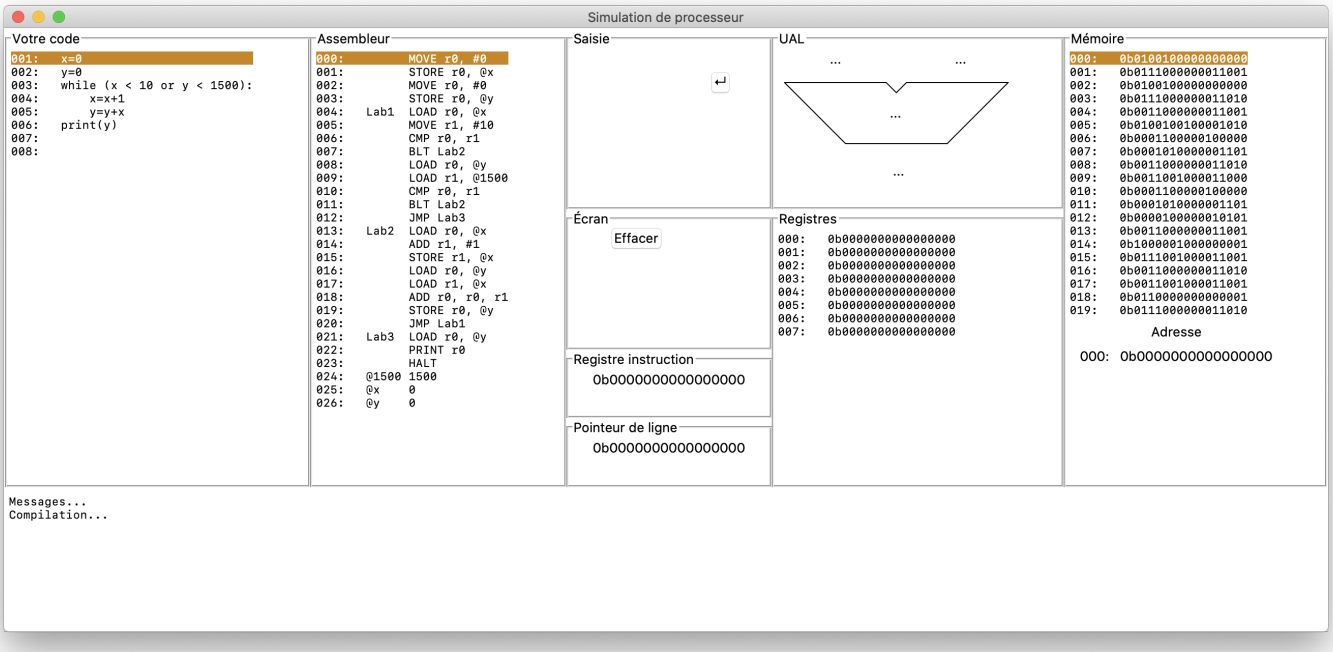


FIGURE 1.1 – Interface Graphique

1.2 Structure

diag UML ou équivalent

2 Choix techniques

2.1 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

2.1.1 Expressions admissibles

Les expression admissibles sont présentées dans la table 2.1 ci-dessous.

TABLE 2.1 – 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`

2.1.2 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

2.1.3 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.

2.1.4 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 Modèle de processeur

2.2.1 ProcessorEngine

Le simulateur doit offrir une certaine modularité afin de permettre d'apprécier l'incidence des choix de conception sur le code assembleur et sur l'exécution.

Les propriétés du processeur sont gérées par la classe `ProcessorEngine`

On pourra donc définir le modèle de processeur retenu à l'aide d'un dictionnaire qui prend pour clés :

- le nom du modèle associé : `'name': str`;
- la taille des registres : `'register_bits': int`;
- la taille des mots : `'data_bits': int`;
- la capacité ou non de réorienter la sortie de l'UAL vers un registre quelconque : `'free_ual_output':bool`. Si `False`, la sortie de l'UAL sera systématique le registre 0. Il convient alors de libérer celui-ci ;
- la liste des commandes pouvant accepter directement des littéraux `'litteralCommands':Dict[str,Commands]` ;
- la liste des commandes admissibles `'commands':Dict[str,Commands]`.

Chaque `Command` correspond à un dictionnaire qui prends pour clés :

- un code binaire `'opcode': str,`. Le choix des `opcode` est fait de telle sorte que la taille des mots soit op
- une commande assembleur `'asm': str`,
- la taille du littéral associé `'litteral_bits': int`

Deux modèles sont implémentés par défaut dans le simulateur.

TABLE 2.2 – Processeur 16 bits

register_bits	3
free_ual_output	True
data_bits	16

litteralCommands

Nom	OPCODE	ASM
neg	010110	NEG
move	01001	MOVE
+	1000	ADD
-	1001	SUB
*	1010	MULT
/	1011	DIV
%	1100	MOD
&	1101	AND
|	1110	OR
^	1111	XOR
~	010111	NOT

Commands

Nom	OPCODE	ASM
halt	00000	HALT
goto	000001	JMP
!=	0001000	BNE
==	0001001	BEQ
<	0001010	BLT
>	0001011	BGT
cmp	00011	CMP
print	00100	PRINT
input	00101	INPUT
load	0011	LOAD
move	01000	MOVE
neg	010100	NEG
~	010101	NOT
+	0110000	ADD
-	0110001	SUB
*	0110010	MULT
/	0110011	DIV
%	0110100	MOD
&	0110101	AND
|	0110110	OR
^	0110111	XOR
store	0111	STORE

TABLE 2.3 – Processeur 12 bits

register_bits	2
free_ual_output	False
data_bits	12

litteralCommands

Nom	OPCODE	ASM
NONE		

Commands

Nom	OPCODE	ASM
halt	0000	HALT
goto	0001	JMP
==	0010	BEQ
<	0011	BLT
cmp	11110101	CMP
print	0100	PRINT
input	0101	INPUT
load	100	LOAD
move	11110110	MOVE
~	11110111	NOT
+	11111000	ADD
-	11111001	SUB
*	11111010	MULT
/	11111011	DIV
%	11111100	MOD
&	11111101	AND
|	11111110	OR
^	11111111	XOR
store	101	STORE

La classe `ProcessorEngine` a la responsabilité entre autre d'assurer que le modèle de processeur soit consistant, d'assurer la conversion entre code assembleur et code binaire.

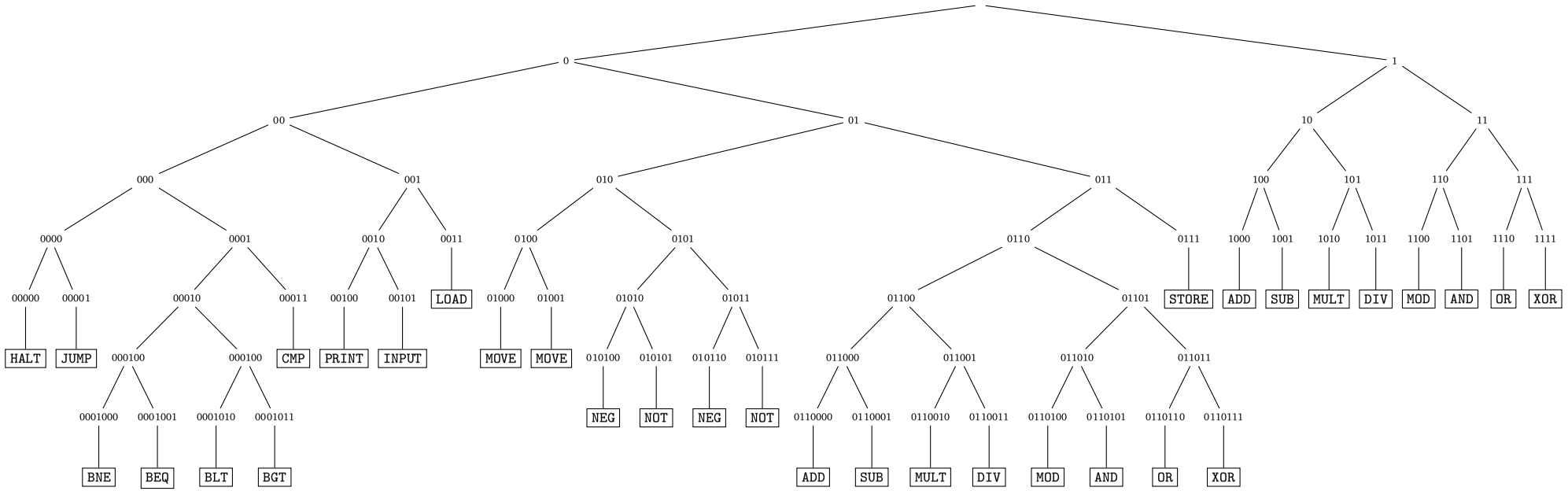


FIGURE 2.2 – Langage 16 bits - OPCODE et ASM

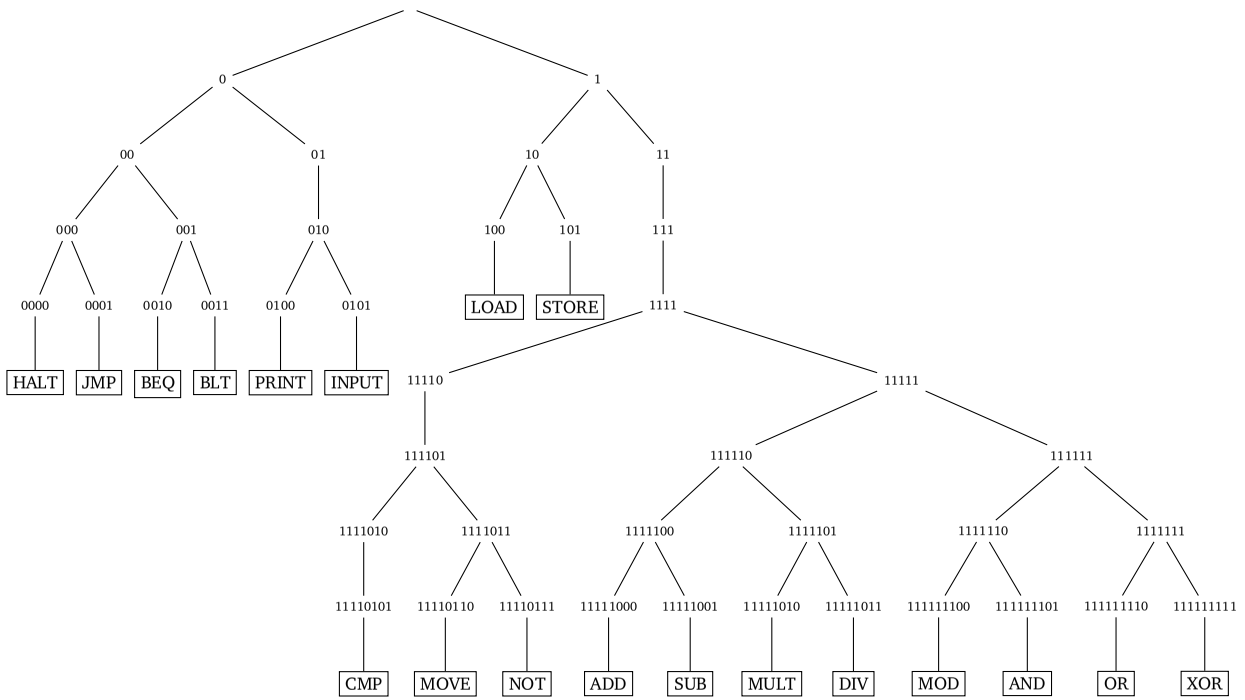


FIGURE 2.3 – Langage 12 bits - OPCODE et ASM

2.2.2 Executeur

Lors de l'exécution, le processeur modèle est représenté par un objet de classe `Executeur` (figure 2.4). Les différents paramètres (taille des registres, taille mémoire, fonctionnement UAL) sont définis par la classe `ProcessorEngine` associée. Afin de permettre le suivi de l'exécution, l'`Executeur` implémente entre autre :

- 2 bus de données : `_DATA_BUS` et `_DATA_BUS_2` ;
- une mémoire : `_MEMORY`
- un registre adresse mémoire `_MEMORY_ADDRESS` et un registre instruction `_INSTRUCTION_REGISTER`
- un pointeur de ligne `_LINE_POINTER`
- une sortie affichage `_PRINT`
- un buffer `_BUFFER`
- une UAL `_UAL`

Chaque composant est modélisé par une instance d'une classe dédiée (`ScreenComponent`, `UalComponent`, `RegisterGroup`, etc...) implémentant les méthodes associées au comportement de chaque composant physique, par exemple pour l'UAL :

- définir l'opération à venir : `setOperation()` ;
- mémoriser le premier opérande :`writeFirstOperand()` ;
- mémoriser le second opérande : `writeSecondOperand()` ;
- exécuter le calcul : `execCalc()` ;
- lire le résultat : `read()` ;

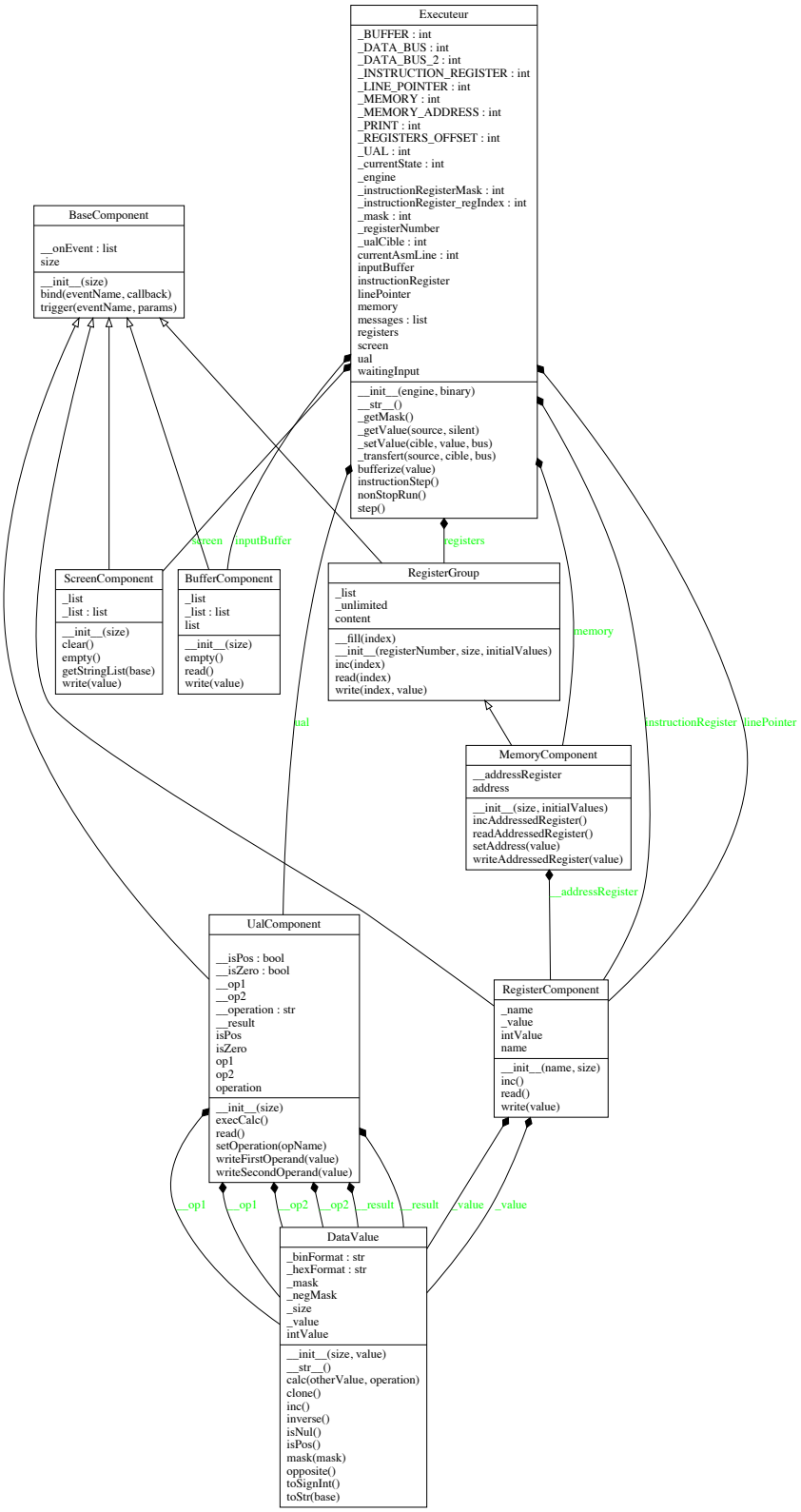


FIGURE 2.4 – Diagramme de classe - Executeur

2.3 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

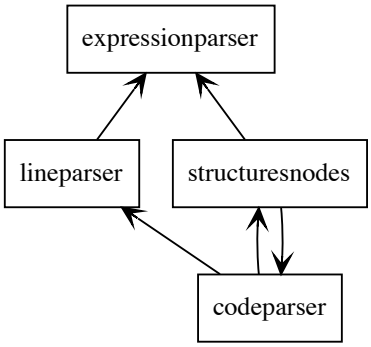


FIGURE 2.5

```
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 2.6 – Exemple simpliste de parse

2.3.1 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 `__str__()` et `__recursiveStringifyLine()`.

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

2.3.2 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`.

2.3.3 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 représentant l'expression en notation polonaise inverse.

Pour cela la chaine de caractère représentant l'expression est convertie en une liste de Tokens (`__buildTokensList()`) représentant chaque type admissible dans la chaine de caractère. Ceux-ci peuvent correspondre à :

- une variable `TokenVariable`
- un nombre `TokenNumber`
- un opérateur binaire `TokenBinaryOperator`
- un opérateur unaire `TokenUnaryOperator`
- une parenthèse `TokenParenthesis`

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

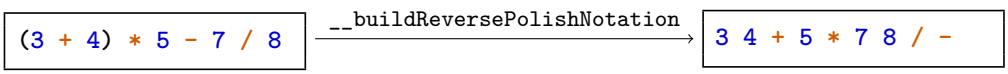
- `strIsExpression()` s'assure que la chaîne de caractère est une expression régulière ;
- `testBrackets()` teste l'équilibre des parenthèses ;
- `__tokensListIsLegal()` teste si l'enchaînement de Token est correct à partir de la table de vérité (2.4)
- `__consolidAddSub()` doit permettre de simplifier la liste de Token pour des enchainements de type `'(+'`, `'+-'`,...

TABLE 2.4 – Enchainements autorisés de Token

Suivant / Précédent	None	Opérateur Binaire	Opérateur Unaire	Opérande	(	)
None	1	0	1	1	1	0
Opérateur Binaire	0	0	1	1	1	0
Opérateur Unaire	0	0	0	1	1	0
Opérande	1	1	0	0	0	1
(	0	0	1	1	1	0
)	1	1	0	0	0	1

A partir de la liste de Token, la construction de l'arbre associé à l'expression nécessite :

- `__buildReversePolishNotation()` : réorganisation de la liste de Token en notation polonaise inverse



- `__buildTree()` : construction de l'arbre associé à l'expression. Fait appel à la méthode `toNode()` de la classe `Token` pour créer les instances des classes `ArithmeticExpressionNode`, `ComparaisonExpressionNode` ou `LogicExpressionNode` suivant le type d'expression analysée.

2.3.4 Token

Les classes `TokenVariable`, `TokenNumber`, `TokenBinaryOperator`, `TokenUnaryOperator` et `TokenUnaryOperator` héritées de la classe `Token` implémentent l'ensemble des tests nécessaires à l'identification et à l'usage des Token (gestion de priorité, distinction opérateurs/opérandes, etc...)

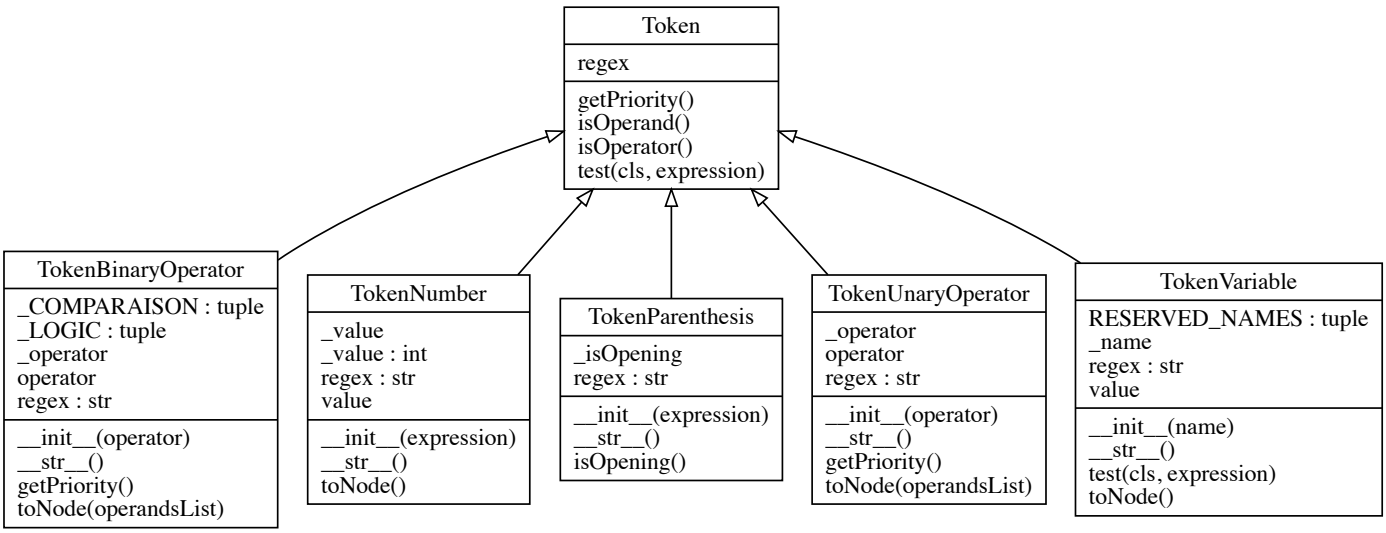


FIGURE 2.7 – Diagramme de classe Token

2.4 Structure de donnée du code analysé

A l'issue de la phase d'analyse, le code est disponible sous la forme d'une liste de `StructureNode` (figure 2.6). C'est à partir de cette liste que sera produit le code assembleur et le code binaire associé.

L'ensemble des classes décrites ci-dessous font l'objet d'un transtypage permettant d'afficher celles-ci sous la forme d'une chaîne de caractères.

2.4.1 Classe StructureNode

Les classes héritées de `StructureNode` sont présentées sur la figure 2.8. On remarquera que les `StructureNode` peuvent être des structures de données récursives, les nœuds de type `IfNode`, `IfElseNode` et `WhileNode` ayant pour attribut `_children` de type `StructureNodeList`.

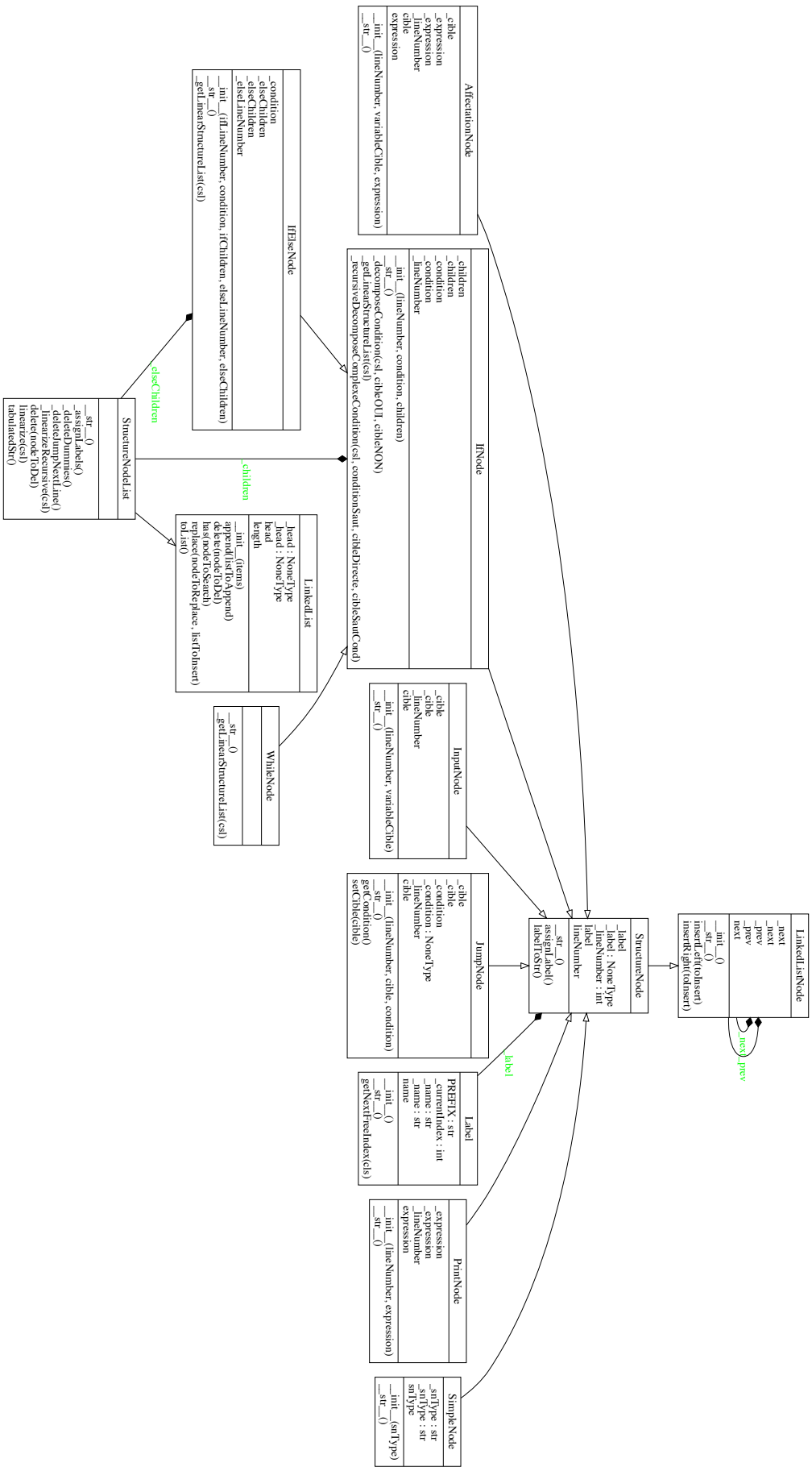


FIGURE 2.8 – Diagramme de classes - StructureNode

2.4.2 Classes ArithmeticExpressionNode, LogicExpressionNode et ComparisonExpressionNode

Les conditions de branchement `if ... then` ou d'arrêt de boucle `while` sont implémentées comme attribut (`_condition`) des nœuds de type `IfNode`, `IfElseNode` et `WhileNode`. Elles sont associées à des instances des classes `LogicExpressionNode` ou `ComparisonExpressionNod`

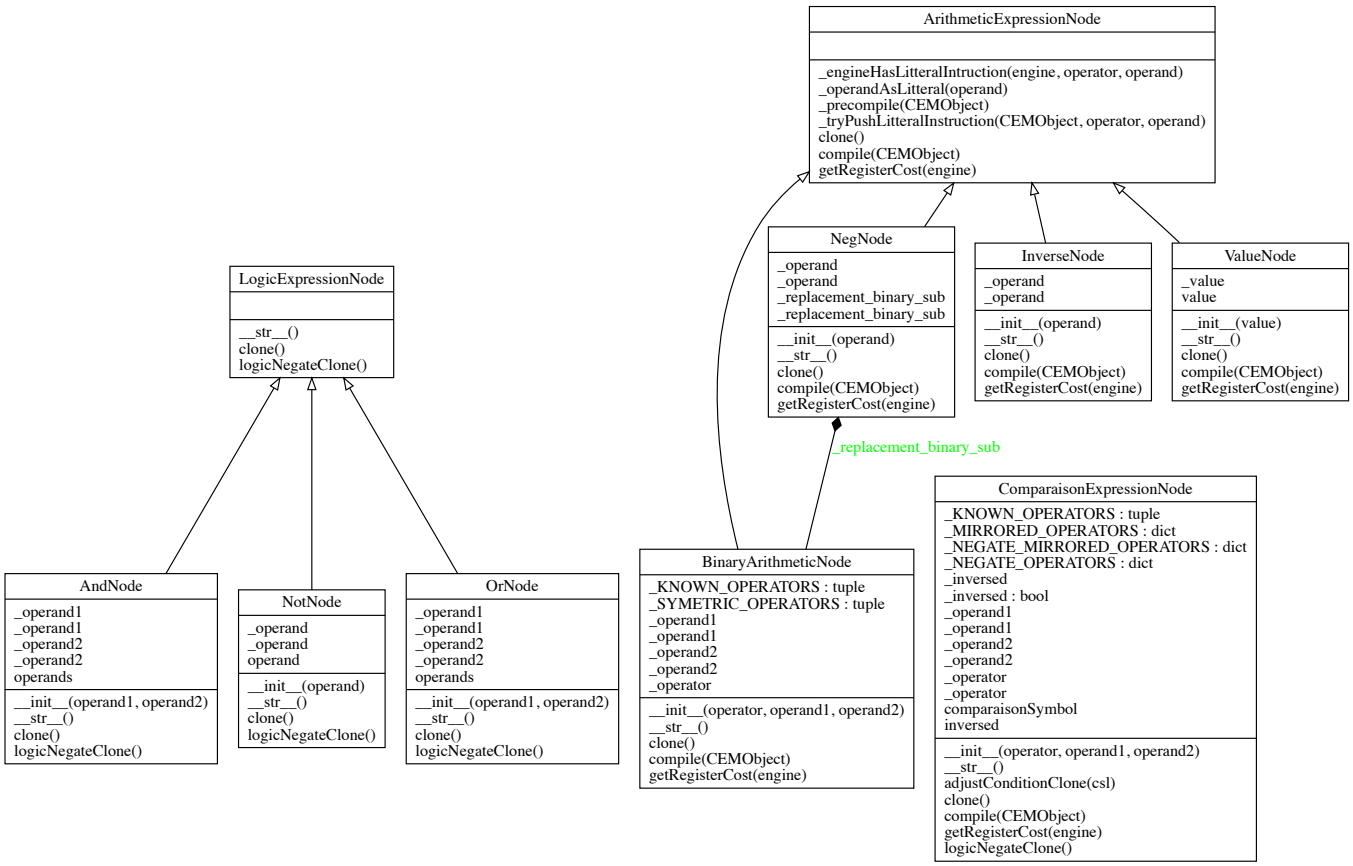


FIGURE 2.9 – Diagramme de classes - ExpressionNode

Les expressions de type comparaisons (`ComparisonExpressionNode`), expression arithmétiques `ArithmeticExpressionNode` ou les affectations (`AffectationNode`) peuvent avoir pour attributs des instances de la classe `ArithmeticExpressionNode`.

2.5 Compilation

2.6 Gestion de la documentation

La gestion de la documentation a initialement été mise en place sous la forme d'un wiki sur le dépôt github du projet : uPSimulator : Simulateur de processeur (`https://github.com/gromax/uPSimulator/wiki`)

Dans un second temps, le choix s'est porté sur Sphinx qui permet de renseigner directement le code source, d'exporter dans de multiples formats et d'inclure des fragments exemples.

3 Organisation

3.1 Planification

3.2 Répartition des tâches

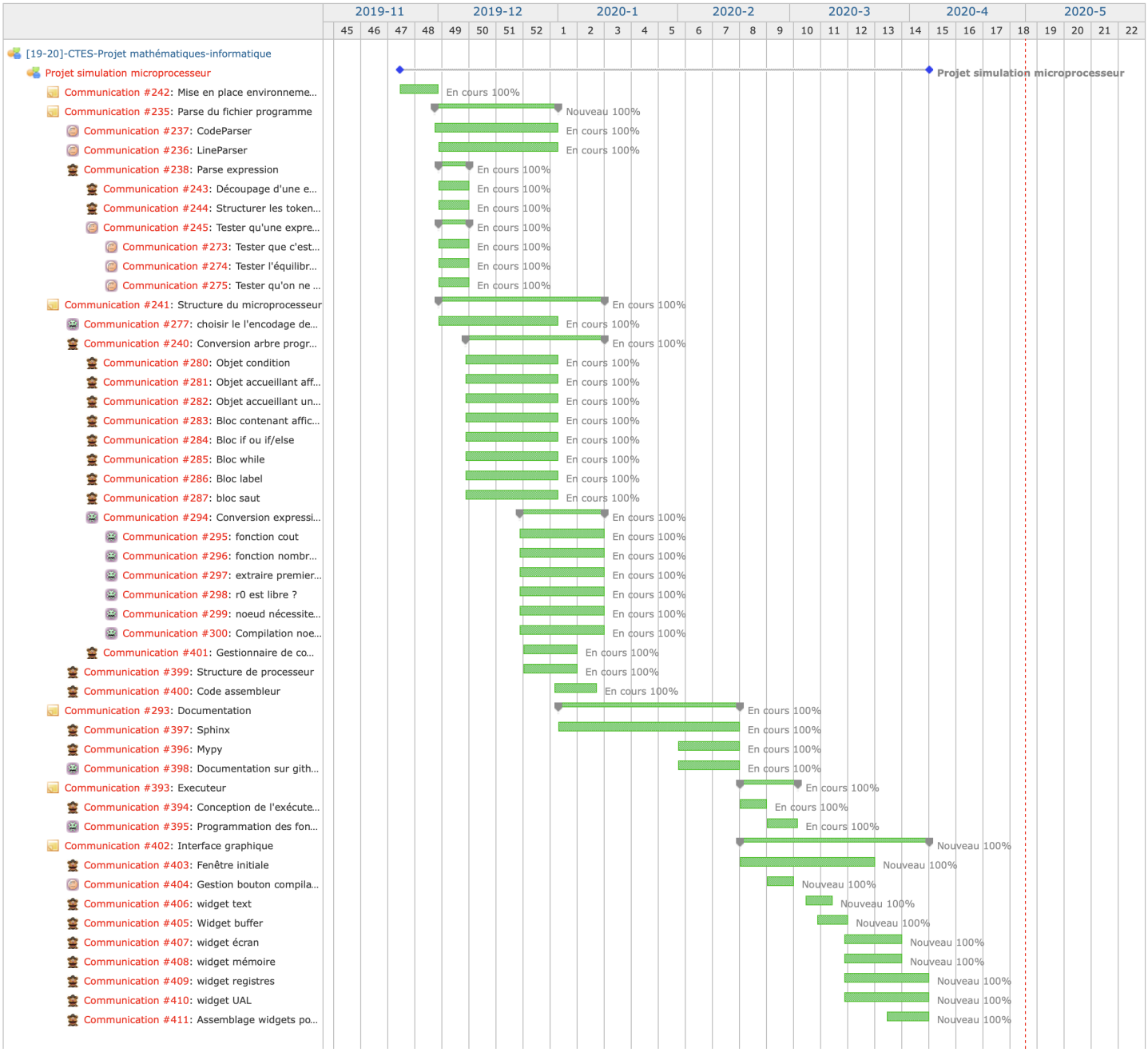


FIGURE 3.10 – Diagramme de Gantt du projet

Table des figures

Liste des tableaux