Simulation microprocesseur

Version 1

Maxence Klein, Guillaume Desjouis, Véronique Reynaud

avr. 15, 2020

Contents:

CHAPITRE 1

uPSimulator

1.1 arithmeticexpressionnodes module

Note : Les noeuds ne sont jamais modifiés. toute modification entraîne la création de clones.

class arithmeticexpressionnodes.**ArithmeticExpressionNode**

Bases : object

abstract clone() → arithmeticexpressionnodes.ArithmeticExpressionNode

Fonction par défaut

Renvoie l'objet lui-même

Type renvoyé *ArithmeticExpressionNode*

abstract compile(*CEMObject : compileexpressionmanager.CompileExpressionManager*) → None

Exécute la compilation

Paramètres **CEMObject** (CompileExpressionManager) – gestionnaire de compilation pour une expression

abstract getRegisterCost(*engine : processorengine.ProcessorEngine*) → int

Calcule le nombre de registre nécessaire pour l'évaluation d'un noeud

Renvoie nombre de registres

Type renvoyé int

class arithmeticexpressionnodes.**BinaryArithmeticNode**(*operator : str, operand1 : arithmeticexpressionnodes.ArithmeticExpressionNode, operand2 : arithmeticexpressionnodes.ArithmeticExpressionNode*)

Bases : *arithmeticexpressionnodes.ArithmeticExpressionNode*

clone() → arithmeticexpressionnodes.BinaryArithmeticNode

Produit un clone de l'objet avec son arborescence

Renvoie clone

Type renvoyé *BinaryArithmeticNode*

compile(*CEMObject : compileexpressionmanager.CompileExpressionManager*) → None
Procédure d'exécution de la compilation

Paramètres **CEMObject** (CompileExpressionManager) – objet prenant en charge la compilation d'une expression

Renvoie None

getRegisterCost(*engine : processorengine.ProcessorEngine*) → int
Calcul du nombre de registre nécessaires pour évaluer ce noeud

Paramètres **engine** (ProcessorEngine) – modèle de processeur

Renvoie nombre de registres

Type renvoyé int

Example

```
>>> engine = ProcessorEngine()
>>> oLitteral1 = ValueNode(Litteral(4))
>>> oLitteral2 = ValueNode(Litteral(-15))
>>> oLitteral3 = ValueNode(Litteral(-47))
>>> oAdd1 = BinaryArithmeticNode('+', oLitteral1, oLitteral2)
>>> oAdd2 = BinaryArithmeticNode('+', oLitteral2, oLitteral3)
>>> oMult = BinaryArithmeticNode('*', oAdd1, oAdd2)
>>> oMult.getRegisterCost(engine)
2
```

```
>>> engine = ProcessorEngine('12bits')
>>> oLitteral1 = ValueNode(Litteral(4))
>>> oLitteral2 = ValueNode(Litteral(-15))
>>> oLitteral3 = ValueNode(Litteral(-47))
>>> oAdd1 = BinaryArithmeticNode('+', oLitteral1, oLitteral2)
>>> oAdd2 = BinaryArithmeticNode('+', oLitteral2, oLitteral3)
>>> oMult = BinaryArithmeticNode('*', oAdd1, oAdd2)
>>> oMult.getRegisterCost(engine)
3
```

class arithmeticexpressionnodes.**InverseNode**(*operand : arithmeticexpressionnodes.ArithmeticExpressionNode*)

Bases : *arithmeticexpressionnodes.ArithmeticExpressionNode*

clone() → arithmeticexpressionnodes.ArithmeticExpressionNode
Crée un noeud clone

Renvoie clone

Type renvoyé UnaryNode

Note : L'aborescence enfant est également clonée.

---

compile(*CEMObject : compileexpressionmanager.CompileExpressionManager*) → None
Exécute la compilation

Paramètres **CEMObject** (*compileExpressionManager*) – gestionnaire de compilation pour une expression

getRegisterCost(*engine : processorengine.ProcessorEngine*) → int
Calcul le nombre de registre nécessaire pour l'évaluation d'un noeud

Renvoie nombre de registres

Type renvoyé int

Note : L'opérande étant placée dans un registre, on peut envisager de placer le résultat au même endroit. L'opération ne nécessite alors pas de registres supplémentaire.

---

class arithmeticexpressionnodes.**NegNode**(*operand : arithmeticexpressionnodes.ArithmeticExpressionNode*)

Bases : *arithmeticexpressionnodes.ArithmeticExpressionNode*

clone() → arithmeticexpressionnodes.ArithmeticExpressionNode

Crée un noeud clone

Renvoie clone

Type renvoyé UnaryNode

Note : L'aborescence enfant est également clonée.

---

compile(*CEMObject : compileexpressionmanager.CompileExpressionManager*) → None

Exécute la compilation

Paramètres **CEMObject** (*compileExpressionManager*) – gestionnaire de compilation pour une expression

Renvoie None

getRegisterCost(*engine : processorengine.ProcessorEngine*) → int

Calcul le nombre de registre nécessaire pour l'évaluation d'un noeud

Renvoie nombre de registres

Type renvoyé int

Note : L'opérande étant placée dans un registre, on peut envisager de placer le résultat au même endroit. L'opération ne nécessite alors pas de registres supplémentaire.

Example

```
>>> engine = ProcessorEngine()
>>> oLitteral1 = ValueNode(Litteral(4))
>>> oLitteral2 = ValueNode(Litteral(-15))
>>> oAdd = BinaryArithmeticNode('+', oLitteral1, oLitteral2)
>>> oNeg = NegNode(oAdd)
>>> oNeg.getRegisterCost(engine)
1
```

```
>>> engine = ProcessorEngine('12bits')
>>> oLitteral1 = ValueNode(Litteral(4))
>>> oLitteral2 = ValueNode(Litteral(-15))
>>> oAdd = BinaryArithmeticNode('+', oLitteral1, oLitteral2)
>>> oNeg = NegNode(oAdd)
>>> oNeg.getRegisterCost(engine)
2
```

class arithmeticexpressionnodes.**ValueNode**(*value : Union[litteral.Litteral, variable.Variable]*)

Bases : *arithmeticexpressionnodes.ArithmeticExpressionNode*

clone() → arithmeticexpressionnodes.ValueNode

Produit un clone de l'objet

Renvoie clone

Type renvoyé BinaryNode

compile(*CEMObject : compileexpressionmanager.CompileExpressionManager*) → None
Procédure d'exécution de la compilation

Paramètres **CEMObject** (CompileExpressionManager) – objet prenant en charge la compilation d'une expression

Renvoie None

getRegisterCost(*engine : processorengine.ProcessorEngine*) → int
Calcule le nombre de registre nécessaire pour l'évaluation d'un noeud

Renvoie nombre de registres

Type renvoyé int

Example

```
>>> v = ValueNode(Litteral(5))
>>> v.getRegisterCost(ProcessorEngine())
1
```

property value
Accesseur

Renvoie valeur

Type renvoyé Union[*Variable*,*Litteral*]

1.2 assembleurcontainer module

class assembleurcontainer.**AssembleurContainer**(*engine : processorengine.ProcessorEngine*)
Bases : object

getAsmSize() → int
Calcule le nombre de lignes instructions.

Renvoie nombre de lignes

Type renvoyé int

getBinary() → str
Produit le code binaire correspondant au code assembleur.

Renvoie code binaire

Type renvoyé str

getDecimal() → List[int]
Produit une version int du code binaire.

Renvoie code assembleur sous forme d'une liste d'entier

Type renvoyé List[int]

getLineLabel(*label : label.Label*) → Optional[int]
Calcul l'adresse d'une étiquette.

Paramètres **label** (Label) – étiquette recherchée

Renvoie adresse de l'étiquette. None si elle n'est pas trouvée.

Type renvoyé Union[int,None]

getLineNumber(*indexAsmLine : int*) → int
index ligne asm -> index ligne origine :param indexAsmLine : index ligne assembleur :type indexAsmLine : int :return : index ligne origine, -1 par défaut :rtype : int

getMemAbsPos (*item : variable.Variable*) → Optional[int]
Calcule l'adresse mémoire d'une variable.

Paramètres **item** (Variable) – variable recherchée

Renvoie adresse de la mémoire. None si elle n'est pas trouvée.

Type renvoyé Union[int,None]

pushCmp (*lineNumber : int, label : Optional[label.Label], operand1 : int, operand2 : int*) → None
Ajoute une commande CMP, comparaison, dans l'assembleur.

Paramètres
- **lineNumber** (*int*) – numéro de la ligne d'origine
- **label** (*Optional[Label]*) – label de l'instruction
- **operand1** (*int*) – registre premier opérande
- **operand2** (*int*) – registre second opérande

Note : Une telle commande doit précéder l'utilisation d'un saut conditionnel.

---

pushHalt (*label : Optional[label.Label]*) → None
Ajoute une commande HALT, fin de programme, à l'assembleur.

Paramètres **label** (*Optional[Label]*) – label de l'instruction

pushInput (*lineNumber : int, label : Optional[label.Label], destination : variable.Variable*) → None
Ajoute une commande INPUT, lecture entrée, dans l'assembleur.

Paramètres
- **lineNumber** (*int*) – numéro de la ligne d'origine
- **label** (*Optional[Label]*) – label de l'instruction
- **destination** (Variable) – variable cible

pushJump (*lineNumber : int, label : Optional[label.Label], cible : label.Label, operator : Optional[str] = None*) → None
Ajoute une commande JUMP, saut conditionnel ou non, dans l'assembleur.

Paramètres
- **lineNumber** (*int*) – numéro de la ligne d'origine
- **label** (*Optional[Label]*) – label de l'instruction
- **cible** (Label) – étiquette cible
- **operator** (*str / None*) – comparaison parmi <, <=, >=, >, ==, !=. None pour Jump inconditionnel

pushLoad (*lineNumber : int, label : Optional[label.Label], source : Union[variable.Variable, litteral.Litteral], destination : int*) → None
Ajoute une commande LOAD dans l'assembleur.

Réserve l'espace mémoire pour la source.

Paramètres
- **lineNumber** (*int*) – numéro de la ligne d'origine
- **label** (*Optional[Label]*) – label de l'instruction
- **destination** (*int*) – registre destination
- **source** (*Litteral / Variable*) – variable ou littéral source

pushMove (*lineNumber : int, label : Optional[label.Label], source : int, destination : int*) → None
Ajoute une commande MOVE dans l'assembleur

Paramètres
- **lineNumber** (*int*) – numéro de la ligne d'origine
- **label** (*Optional[Label]*) – label de l'instruction
- **destination** (*int*) – registre destination
- **source** (*int*) – registre source

pushMoveLitteral(*lineNumber : int, label : Optional[label.Label], source : litteral.Litteral, destination : int*) → None
Ajoute une commande MOVE avec littéral dans l'assembleur.

Paramètres
- **lineNumber** (*int*) – numéro de la ligne d'origine
- **label** (*Optional[Label]*) – label de l'instruction
- **destination** (*int*) – registre destination
- **source** (Litteral) – littéral source

pushPrint(*lineNumber : int*, *source : int*) → None
Ajoute une commande PRINT, affichage à l'écran, dans l'assembleur.

Paramètres
- **lineNumber** (*int*) – numéro de la ligne d'origine
- **source** (*int*) – registre dont on doit afficher le contenu

pushStore(*lineNumber : int, label : Optional[label.Label], source : int, destination : variable.Variable*) → None
Ajoute une commande STORE dans l'assembleur.

Réserve l'espace mémoire pour la variable.

Paramètres
- **lineNumber** (*int*) – numéro de la ligne d'origine
- **label** (*Optional[Label]*) – label de l'instruction
- **source** (*int*) – registre source
- **destination** (Variable) – variable destination

pushUal(*lineNumber : int, label : Optional[label.Label], operator : str, destination : int, regOperands : Tuple[int, ...], littOperand : Optional[litteral.Litteral] = None*) → None
Ajoute une commande de calcul UAL dans l'assembleur.

Paramètres
- **lineNumber** (*int*) – numéro de la ligne d'origine
- **label** (*Optional[Label]*) – label de l'instruction
- **destination** (*int*) – registre destination
- **operator** (*string*) – opérateur
- **regOperands** (*tuple[int]*) – opérandes de type registre
- **littOperand** (*Optional[Litteral]*) – opérande de type littéral

1.3 assembleurlines module

class assembleurlines.**AsmLine**(*lineNumber : int, label : Optional[label.Label], opcode : str, asmCommand : str, regOperands : Tuple[int, ...], specialOperand : Union[variable.Variable, label.Label, litteral.Litteral, None]*)

Bases : object

getElementsToCode() → List[Union[int, str, label.Label, variable.Variable, litteral.Litteral]]
Retourne une liste d'éléments à coder

Renvoie liste des éléments à coder

Type renvoyé List[Union[int, str, *Variable*, *Litteral*]]

getLastOperandSize(*wordSize : int*, *regSize : int*) → int
Retourne le nombre de bits laissés pour le codage binaire d'un éventuel dernier argument spécial (Litteral, Variable ou str pour label)

Paramètres
- **wordSize** (*int*) – taille du mot en mémoire
- **regSize** (*int*) – taille des opérandes de type registre

Renvoie nombre de bits laissés après les registres

Type renvoyé int

Example

```
>>> AsmLine(-1,"", "1111", "ADD", (0,2), None).
↪getLastOperandSize(16,3)
6
```

getSizeInMemory() → int

Détermine le nombre de lignes mémoires nécessaires pour cette ligne assembleur.

Renvoie 1 pour ligne ordinaire

Type renvoyé int

Example

```
>>> AsmLine(-1,"", "ADD", "111", (0,2,3), None).getSizeInMemory()
1
```

property label

Accesseur :return : étiquette :rtype : Label

property lineNumber

Accesseur

Renvoie numéro de la ligne d'origine

Type renvoyé int

Example

```
>>> AsmLine(12,"Lab1", "1111", "ADD", (0,2), None).lineNumber
12
```

1.4 codeparser module

class codeparser.**CodeParser**(***options*)

Bases : object

Classe CodeParser Parse le contenu d'un fichier passé en paramètre au constructeur Une méthode public parseCode qui construit une liste d'objets LineParser avec organisation des enfants selon indentation TODO - une méthode de contrôle de cohérence du code ATTENTION - pas de gestion particulière du if, elif, else -> pas de tuple pour le noeud if et else –> voir structureelements

getFinalStructuredList() → List[structuresnodes.StructureNode]

parseCode(*lignesCode : List[str]*) → None

1.5 comparaisonexpressionnodes module

Note : Les noeuds ne sont jamais modifiés. toute modification entraîne la création de clones.

class comparaisonexpressionnodes.**ComparaisonExpressionNode**(*operator : str*, *operand1 : arithmeticexpressionnodes.ArithmeticExpressionNode*, *operand2 : arithmeticexpressionnodes.ArithmeticExpressionNode*)

Bases : object

adjustConditionClone(*csl : List[str]*) → comparaisonexpressionnodes.ComparaisonExpressionNode

Ajustement des opérateurs de tests en fonction des symboles de comparaison disponibles

Paramètres **csl** (*list[str]*) – symboles de comparaison disponibles

Renvoie clone dont l'expression est adaptée

Type renvoyé *ComparaisonExpressionNode*

Raises AttributesErrors si aucun opérateur ne convient

Example

```
>>> from arithmeticexpressionnodes import ValueNode
>>> from variable import Variable
>>> from litteral import Litteral
>>> oLitteral = ValueNode(Litteral(4))
>>> oVariable = ValueNode(Variable('x'))
>>> oComp = ComparaisonExpressionNode('>', oLitteral, oVariable)
>>> oComp2 = oComp.adjustConditionClone(['<', '=='])
>>> str(oComp2)
'(@x < #4)'
```

```
>>> from arithmeticexpressionnodes import ValueNode
>>> from variable import Variable
>>> from litteral import Litteral
>>> oLitteral = ValueNode(Litteral(4))
>>> oVariable = ValueNode(Variable('x'))
>>> oComp = ComparaisonExpressionNode('>=', oLitteral, oVariable)
>>> oComp2 = oComp.adjustConditionClone(['<', '=='])
>>> str(oComp2)
'not (#4 < @x)'
```

clone() → comparaisonexpressionnodes.ComparaisonExpressionNode

Produit un clone de l'objet avec son arborescence

Renvoie clone

Type renvoyé *ComparaisonExpressionNode*

property comparaisonSymbol

Accesseur

Renvoie symbole de comparaison

Type renvoyé str

compile(*CEMObject : compileexpressionmanager.CompileExpressionManager*) → None

Procédure d'exécution de la compilation

Paramètres **CEMObject** (CompileExpressionManager) – objet prenant en charge la compilation d'une expression

getRegisterCost(*engine : processorengine.ProcessorEngine*) → int

Calcul du nombre de registre nécessaires pour évaluer ce noeud

Paramètres **engine** (ProcessorEngine) – modèle de processeur

Renvoie nombre de registres

Type renvoyé int

Example

```
>>> engine = ProcessorEngine()
>>> from arithmeticexpressionnodes import ValueNode,␣
↪BinaryArithmeticNode
>>> from litteral import Litteral
>>> oLitteral1 = ValueNode(Litteral(4))
>>> oLitteral2 = ValueNode(Litteral(-15))
>>> oLitteral3 = ValueNode(Litteral(-47))
>>> oAdd1 = BinaryArithmeticNode('+', oLitteral1, oLitteral2)
>>> oAdd2 = BinaryArithmeticNode('+', oLitteral2, oLitteral3)
>>> oComp = ComparaisonExpressionNode('<', oAdd1, oAdd2)
>>> oComp.getRegisterCost(engine)
2
```

```
>>> engine = ProcessorEngine('12bits')
>>> from arithmeticexpressionnodes import ValueNode,␣
↪BinaryArithmeticNode
>>> from litteral import Litteral
>>> oLitteral1 = ValueNode(Litteral(4))
>>> oLitteral2 = ValueNode(Litteral(-15))
>>> oLitteral3 = ValueNode(Litteral(-47))
>>> oAdd1 = BinaryArithmeticNode('+', oLitteral1, oLitteral2)
>>> oAdd2 = BinaryArithmeticNode('+', oLitteral2, oLitteral3)
>>> oComp = ComparaisonExpressionNode('<', oAdd1, oAdd2)
>>> oComp.getRegisterCost(engine)
3
```

property inversed
Accesseur

Renvoie la comparaison doit être inversée

Type renvoyé bool

logicNegateClone() → comparaisonexpressionnodes.ComparaisonExpressionNode
Complément

Renvoie noeud dont les l'expression est complémentaire, ou le noeud lui même si pas de changement

Type renvoyé *ComparaisonExpressionNode*

Example

```
>>> from arithmeticexpressionnodes import ValueNode
>>> from variable import Variable
>>> from litteral import Litteral
>>> oLitteral = ValueNode(Litteral(4))
>>> oVariable = ValueNode(Variable('x'))
>>> oComp = ComparaisonExpressionNode('>', oLitteral, oVariable)
>>> oComp2 = oComp.logicNegateClone()
>>> str(oComp2)
'not (#4 > @x)'
```

1.6 compileexpressionmanager module

class compileexpressionmanager.**CompileExpressionManager**(*engine : processorengine.ProcessorEngine, asmManager : assembleurcontainer.AssembleurContainer, lineNumber : int, label : Optional[label.Label]*)

Bases : object

property engine

Accesseur

Renvoie modèle de processeur utilisé

Type renvoyé *ProcessorEngine*

getNeededRegisterSpace(*cost : int*, *needUAL : bool*) → None

Déplace des registres au besoin * Déplace le registre 0 s'il est nécessaire pour l'UAL. * Déplace les registres vers la mémoire autant que possible ou nécessaire

Paramètres

— **cost** (*int*) – cout en registre du noeud d'opération nen cours

— **needUAL** (*bool*) – le noeud nécessitera-t-il l'utilisation de l'UAL ?

getResultRegister() → int

Renvoie registre en haut de la pile

Type renvoyé int

pushBinaryOperator(*operator : str*, *directOrder : bool*) → None

Ajoute une opération binaire. Libère les 2 registres au sommet de la pile, ajoute l'opération, Occupe le premier registre libre pour le résultat

Paramètres

— **operator** (*str*) – opération parmi +, −, *, /, %, &, |, ^

— **directOrder** (*bool*) – vrai si le calcul est donné dans l'ordre, c'est à dire si le haut de la pile correspond au 2e opérande

pushBinaryOperatorWithLitteral(*operator : str*, *litteral : litteral.Litteral*) → None

Ajoute une opération binaire dont le 2e opérand est un littéral Libère le registre au sommet de la pile comme 1er opérande. Occupe le premier registre libre pour le résultat.

Paramètres

— **operator** (*str*) – opération parmi +, −, *, /, %, &, |, ^

— **litteral** (Litteral) – littéral

pushUnaryOperator(*operator : str*) → None

Ajoute une opération unaire Libère le registre au sommet de la pile comme opérande. Occupe le premier registre libre pour le résultat.

Paramètres **operator** (*str*) – opération parmi ~, −

pushUnaryOperatorWithLitteral(*operator : str*, *litteral : litteral.Litteral*) → None

Ajoute une opération unaire dont l'opérande est un littéral Occupe le premier registre libre pour le résultat

Paramètres

— **operator** (*str*) – opération parmi ~, −

— **litteral** (Litteral) – littéral

pushValue(*value : Union[litteral.Litteral, variable.Variable]*) → None

Charge une valeur dans le premier registre disponible

Paramètres **value** (*Union[Litteral,Variable]*) – valeur à charger

resetMemory() → None
Réinitialise les items mémoires du compilateur.

stringMemoryUsage() → str

Renvoie représentation de l'occupation actuelle de la mémoire.

Type renvoyé str

1.7 compilemanager module

class compilemanager.**CompilationManager**(*engine : processorengine.ProcessorEngine, listOfStructureNodes : List[structuresnodes.StructureNode]*)

Bases : object

property asm
Accesseur

Renvoie objet assembleur contenant le résultat de la compilation

Type renvoyé *AssembleurContainer*

1.8 errors module

Gestionnaire d'erreurs

exception errors.**AttributesError**(*message*, *errors=None*)
Bases : Exception

exception errors.**CompilationError**(*message*, *errors=None*)
Bases : Exception

exception errors.**ExpressionError**(*message*, *errors=None*)
Bases : Exception

exception errors.**ParseError**(*message*, *errors=None*)
Bases : Exception

1.9 example module

class example.**CompilationTest**(*methodName='runTest'*)
Bases : unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

test_affectation_12bits()

test_affectation_16bits()

test_example_1_16bits()

1.10 executeur module

class executeur.**Executeur**(*engine : processorengine.ProcessorEngine, binary : Union[List[int], List[str]]*)

Bases : object

Identifiants des bus : - DATA_BUS : bus de données - DATA_BUS_2 : bus secondaire entre les registres et la 2e opérande UAL

Identifiants des variables internes : - MEMORY : mémoire - MEMORY_ADDRESS : registre adresse mémoire - INSTRUCTION_REGISTER : registre instruction - LINE_POINTER : registre pointeur de ligne - PRINT : affichage - BUFFER : buffer - UAL : Unité Arithmétique et Logique - REGISTERS_OFFSET : registre 0

BUFFER = 5

DATA_BUS = 0

DATA_BUS_2 = 1

INSTRUCTION_REGISTER = 2

LINE_POINTER = 3

MEMORY = 0

MEMORY_ADDRESS = 1

PRINT = 4

REGISTERS_OFFSET = 7

UAL = 6

bufferize(*value : int*) → None

Ajoute un entier au buffer d'entrée

Paramètres **value** (*int*) – valeur à bufferiser

currentAsmLine = 0

getValue(*source : int*, *silent=False*) → Optional[int]

lit la valeur d'un variable interne du processeur virtuel

Paramètres **source** (*int*) – identifiant de la variable

Result valeur de la variable ou False si l'identifiant est inconnu

Type renvoyé Optional[int]

instructionStep() → int

Exécution d'une instruction complète. Commande donc l'exécution de plusieurs step jusqu'à ce que currentState revienne à 0, -1 ou -2

Renvoie état en cours. -1 = halt -2 = attente input 0 = début instruction,

Type renvoyé int

nonStopRun() → int

Exécution du programme en continu Commande donc l'exécution de plusieurs step jusqu'à ce que currentState revienne -1 ou -2

Renvoie état en cours. -1 = halt -2 = attente input

Type renvoyé int

..warning : : Si le programme boucle, l'instruction bouclera aussi. De plus, ce programme prend la main pour toute une exécution. Ne convient donc pas au cas d'une visualisation avec interface graphique devant se remettre à jour en parallèle de l'exécution.

step() → int

Exécution d'un pas. Il s'agit d'un pas élémentaire, il en faut plusieurs pour exécuter l'ensemble de l'instruction. Le nombre de pas nécessaire dépend du type d'instruction.

Renvoie état en cours. -1 = halt -2 = attente input 0 = début instruction, 1 <= état interne du processeur correspondant au déroulement de l'instruction

Type renvoyé int

property waitingInput

Accesseur.

Renvoie processeur attend une entrée

Type renvoyé bool

1.11 executeurcomponents module

class executeurcomponents.**BaseComponent**(*size : int*)

Bases : object

bind(*eventName : str, callback : Callable[[Dict[str, Union[str, int]]], None]*)

Enregistre un événement

Paramètres

— **eventName** (*str*) – nom de l'événement

— **callback** (*Callable*) – fonction callback

property size

trigger(*eventName : str, params : Dict[str, Any]*) → None

Déclenche un événement

Paramètres

— **eventName** (*str*) – nom de l'événement

— **params** (*Union[str,int]*) – paramètres empaquetés

class executeurcomponents.**BufferComponent**(*size : int*)

Bases : *executeurcomponents.BaseComponent*

Gestion du buffer d'entrée

empty() → bool

Renvoie True si le buffer est vide

Type renvoyé bool

property list

Accesseur :return : clone du contenu du buffer :rtype : List[int]

read() → Union[executeurcomponents.DataValue, bool]

Renvoie premier item du buffer s'il existe, sinon False

Type renvoyé Union(*DataValue*, bool)

write(*value : int*) → None

Paramètres **value** (*int*) – valeur à ajouter dans le buffer

class executeurcomponents.**DataValue**(*size : int*, *value : int = 0*)

Bases : object

mot de donnée

calc(*otherValue : executeurcomponents.DataValue*, *operation : str*) → executeurcomponents.DataValue

calcule l'opération avec une autre valeur

Paramètres **otherValue** (DataValue) – autre valeur

Renvoie résultat

Type renvoyé *DataValue*

clone() → executeurcomponents.DataValue

inc() → None

incrémente la valeur tenant compte du codage CA2

property intValue

inverse() → executeurcomponents.DataValue
calcul l'inverse d'un entier tenant compte du codage CA2

Renvoie inverse de la valeur

Type renvoyé *DataValue*

isNul() → bool
test si c'est entier nul

Result la valeur est nulle

Type renvoyé bool

isPos() → bool
test si c'est entier est positif (en CA2, tenant compte de la base)

Result la valeur est positive (ou nulle)

Type renvoyé bool

mask(*mask : int*) → executeurcomponents.DataValue
Calcule le résultat de la valeur masquée

Paramètres **mask** (*int*) – masque

Renvoie valeur masquée

Type renvoyé *DataValue*

opposite() → executeurcomponents.DataValue
calcul l'opposé d'un entier tenant compte du codage CA2

Renvoie oppposé de la valeur

Type renvoyé *DataValue*

toSignInt() → int
transtypage en int tenant compte de l'éventel signe CA2

Renvoie valeur courante

Type renvoyé int

toStr(*base : str = 'bin'*) → str
transtypage tenant compte que value n'est pas forcément sur 32 bits comme les int ordinaire de python pour lesquels str() est conçu

Paramètres **base** (*str*) – base de l'écriture parmi "bin", "hex", "dec", "udec"

Result valeur sous forme str

Type renvoyé str

class executeurcomponents.**MemoryComponent**(*size : int*, *initialValues : List[Union[str, int]] = []*)
Bases : *executeurcomponents.RegisterGroup*
Gestion de la mémoire équivalent à RegisterGroup avec adresseRegister en plus

property address

incAddressedRegister() → None

readAddressedRegister() → executeurcomponents.DataValue

setAddress(*value : Union[executeurcomponents.DataValue, int]*) → None

writeAddressedRegister(*value : Union[executeurcomponents.DataValue, int]*) → None

class executeurcomponents.**RegisterComponent**(*name : str*, *size : int*)
Bases : *executeurcomponents.BaseComponent*
Gestion d'un registre

inc() → None
incrémente la valeur du registre

property intValue

property name

Accesseur

Renvoie nom du registre

Type renvoyé str

read() → executeurcomponents.DataValue

lecture de value

Renvoie valeur courante du registre

Type renvoyé int

write(*value : Union[executeurcomponents.DataValue, int]*) → None

écrit la valeur dans le registre

Paramètres **value** (*Union[DataValue, int]*) – ajoute valeur à l'écran

class executeurcomponents.**RegisterGroup**(*registerNumber : int*, *size : int*, *initialValues : List[int] = []*)

Bases : *executeurcomponents.BaseComponent*

Gestion d'un groupe de registres

property content

inc(*index : int*) → None

incrémente la valeur du registre

Paramètres **index** (*int*) – indice du registre incrémenté

read(*index : int*) → Optional[executeurcomponents.DataValue]

lecture de la valeur du registre d'index n :param index : indice du registre lu :type index : int :return : valeur courante du registre. -1 par défaut :rtype : int

write(*index : int, value : Union[executeurcomponents.DataValue, int]*) → None

écrit la valeur dans le registre

Paramètres

— **index** (*int*) – indice du registre écrit

— **value** (*Union[DataValue, int]*) – écrit la valeur dans le registre

class executeurcomponents.**ScreenComponent**(*size*)

Bases : *executeurcomponents.BaseComponent*

Gestion de l'écran

clear()

Efface le contenu

empty() → bool

Renvoie True si le buffer est vide

Type renvoyé bool

getStringList(*base : str = 'bin'*) → List[str]

Paramètres **base** (*str*) – base de la lecture, parmi "bin", "dec", "hex", "udec"

Renvoie liste du contenu de l'écran

Type renvoyé List[str]

write(*value : Union[executeurcomponents.DataValue, int]*) → None

Paramètres **value** (*Union[DataValue, int]*) – ajoute valeur à l'écran

class executeurcomponents.**UalComponent**(*size : int*)

Bases : *executeurcomponents.BaseComponent*

Gestion de l'Unité Arithmétique et Logique

execCalc() → executeurcomponents.DataValue

exécute le calcul

Renvoie valeur du résultat

Type renvoyé *DataValue*

property isPos

property isZero

property op1

property op2

property operation

read() → executeurcomponents.DataValue
lit le résultat

Renvoie résultat du dernier calcul

Type renvoyé *DataValue*

setOperation(*opName : str*) → None
Fixe l'opération

Paramètres **opName** (*str*) – opération parmi neg, ~, +, -, *, /, %, &, |, ^

writeFirstOperand(*value : Union[executeurcomponents.DataValue, int]*) → None
Fixe l'opérande 1

Paramètres **value** (*Union[DataValue, int]*) – valeur de l'opérande

writeSecondOperand(*value : Union[executeurcomponents.DataValue, int]*) → None
Fixe l'opérande 2

Paramètres **value** (*Union[DataValue, int]*) – valeur de l'opérande

1.12 expressionparser module

class expressionparser.**ExpressionParser**
Bases : object

TokensList = [<class 'parsertokens.TokenVariable'>, <class 'parsertokens.TokenNumber'>

classmethod buildExpression(*originalExpression : str*) → Union[arithmeticexpressionnodes.ArithmeticExpressionNode, comparaisonexpressionnodes.ComparaisonExpressionNode, logicexpressionnodes.LogicExpressionNode]
À partir d'une expression sous forme d'une chaîne de texte, produit l'arbre représentant cette expression et retourne la racine de cet arbre.

Paramètres **originalExpression** (*str*) – expression à analyser

Renvoie racine de l'arbre

Type renvoyé Union[*ArithmeticExpressionNode*, *ComparaisonExpressionNode*, *LogicExpressionNode*]

Raises ExpressionError si l'expression ne match pas l'expression régulière ou si les parenthèses ne sont pas convenablement équilibrées, ou si l'expression contient un enchaînement non valable, comme +).

classmethod expressionRegex() → str
Donne accès à l'expression régulière d'une expression

Renvoie expression régulière d'une expression

Type renvoyé str

static isLegal(*precedent*, *suivant*)
Test si l'enchaînement de deux Token est possible. Par exemple "*" ne peut pas suivre "("
— un opérateur binaire ne peut être en première ou dernière place

— ")" ne peut être en premier ni "(" à la fin
— deux opérateurs binaires ne peuvent pas se suivre
— un opérateur binaire ne peut pas suivre "(" ou précéder ")"
— une opérande (nombre/variable) ne peut précéder un opérateur unaire
— deux opérandes ne peuvent se suivre
— deux parenthèses différentes ne peuvent se suivre : "()" et ")(" interdits

Renvoie vrai si l'enchaînement est possible

Type renvoyé bool

Exemple

```
>>> ExpressionParser.isLegal(TokenParenthesis('('),␣
↪TokenBinaryOperator('+'))
False
>>> ExpressionParser.isLegal(TokenParenthesis(')'),␣
↪TokenBinaryOperator('+'))
True
>>> ExpressionParser.isLegal(TokenParenthesis('('),␣
↪TokenUnaryOperator('-'))
True
>>> ExpressionParser.isLegal(TokenParenthesis('('),␣
↪TokenBinaryOperator('-'))
False
```

classmethod regex() → str
Concatène les expressions régulière pour les différents constituants d'une expression

Renvoie expression régulière d'un item d'expression

Type renvoyé str

classmethod strIsExpression(*expression : str*) → bool
Teste si une chaîne de caractères est une expression possible.

Paramètres **expression** – expression à tester

Renvoie vrai si l'expression est valable

Type renvoyé bool

Example

```
>>> ExpressionParser.strIsExpression("45x-3zdf = dz")
False
>>> ExpressionParser.strIsExpression("45*x-3+zdf - dz")
True
```

classmethod strIsVariableName(*nomVariable : str*) → bool
Teste si une chaîne de caractères est un nom de variable possible.

Exclut les mots-clefs du langage

Paramètres **expression** – expression à tester

Renvoie vrai si le nom est valable

Type renvoyé bool

static testBrackets(*expression*) → bool
Test l'équilibre des parenthèses

Renvoie vrai si les parenthèses sont équilibrées

Type renvoyé bool

Example

```
>>> ExpressionParser.testBrackets('4*x - ((3 + 2) + 4)')
True
>>> ExpressionParser.testBrackets('( ( ( ) ) ')
False
>>> ExpressionParser.testBrackets('( ( ) ) ) ')
False
>>> ExpressionParser.testBrackets('( ) ) (')
False
```

classmethod variableRegex() → str
Donne accès à l'expression régulière d'une variable

Renvoie expression régulière d'une variable

Type renvoyé str

1.13 graphic module

class graphic.**InputCodeWindow**
Bases : object

BIN_DISPLAY = 0

DEC_DISPLAY = 1

EXEMPLES = [('exemple 1', 'example.code'), ('exemple 2', 'example2.code'), ('exemple 3

HEX_DISPLAY = 2

MODES = ('bin', 'dec', 'hex')

addMessage(*message*)

inEditMode()

show()

1.14 label module

class label.**Label**
Bases : object

PREFIX = 'Lab'

classmethod getNextFreeIndex() → int
génère un nouvel index de numéro de label. Assure l'unicité des numéros.

Renvoie index pour in nouveau label

Type renvoyé int

property name
Assigne le nom si nécessaire et le retourne

Renvoie nom de l'étiquette

Type renvoyé str

1.15 lineparser module

class lineparser.**Caracteristiques**
Bases : dict

class lineparser.**LineParser**(*originalLine : str*, *lineNumber : int*)
Bases : object
Classe LineParser Une ligne qui passe par LineParser : .__originalLine (contient la ligne d'origine) .__cleanLine (contient la ligne épurée des commentaires et des éventuels espaces en fin de ligne) .__caracteristiques dictonnaire

lineNumber : contient le n° de ligne traitée, passé en paramètre au constructeur indentation : contient le nombre d'espace pour l'indentation emptyLine : True si ligne est vide, False sinon type : correspond au motif identifié (if, elif, while, else, print, input, affectation) condition : contient un objet LogicExpressionNode ou ComparaisonExpressionNode pour les motifs attendant une condition expression : contient un objet ArithmeticExpressionNode s'il s'agit d'une affectation ou d'un print variable : contient un objet Variable s'il s'agit d'une affectation ou d'un input

Une méthode getCaracs() pour retourne le dictionnaire __caracteristiques

getCaracs() → lineparser.Caracteristiques
Accesseur

Renvoie caractéristiques de la ligne

Type renvoyé *Caracteristiques*

1.16 linkedlistnode module

class linkedlistnode.**LinkedList**(*items : List[LinkedListNode]*)
Bases : object

append(*listToAppend : Union[LinkedListNode, LinkedList]*) → None
ajoute le contenu de listToAppend à la suite, listToAppend s'en trouve vidée

Paramètres **listToAppend** (*Union['LinkedListNode', 'LinkedList']*) – liste à ajouter

delete(*nodeToDel : linkedlistnode.LinkedListNode*) → bool
supprime l'élément node

Paramètres **nodeToDel** (LinkedListNode) – élément à supprimer

Renvoie suppression effectuée

Type renvoyé bool

has(*nodeToSearch : linkedlistnode.LinkedListNode*) → bool

Paramètres **nodeToSearch** (LinkedListNode) – noeud cherché

Renvoie la liste contient le noeud

Type renvoyé bool

property head
Accesseur

Renvoie pointeur sur la tête

Type renvoyé Optional[« LinkedListNode »]

property length
propriété

Renvoie nombre d'item de la liste

Type renvoyé int

replace(*nodeToReplace : linkedlistnode.LinkedListNode, listToInsert : Union[LinkedList, LinkedListNode]*) → linkedlistnode.LinkedList
remplace l'élément par une liste

Paramètres
— **nodeToReplace** (LinkedListNode) – noeud à remplacer
— **listToInsert** – liste à insérer à la place

Type Union[« LinkedList », « LinkedListNode »]

Renvoie liste ayant subit l'insertion

Type renvoyé *LinkedList*

toList() → List[linkedlistnode.LinkedListNode]
Produit une liste des items enfants

Renvoie liste des noeuds

Type renvoyé List[*LinkedListNode*]

class linkedlistnode.**LinkedListNode**
Bases : object

insertLeft(*toInsert : Union[LinkedList, LinkedListNode]*) → linkedlistnode.LinkedListNode
Insert un noeud ou tout une chaîne à gauche

Paramètres **toInsert** (*Union["LinkedList", "LinkedListNode"]*) – point de départ de la chaîne à inserrer

Renvoie noeud tête de l'insertion

:rtype : : LinkedListNode

insertRight(*toInsert : Union[LinkedList, LinkedListNode]*) → linkedlistnode.LinkedListNode
Insert un noeud ou tout une chaîne à droite

Paramètres **toInsert** (*Union["LinkedList", "LinkedListNode"]*) – point de départ de la chaîne à inserrer

Renvoie noeud courant

:rtype : : LinkedListNode

property next
Accesseur

Renvoie noeud suivant

Type renvoyé *LinkedListNode*

1.17 litteral module

class litteral.**Litteral**(*value : int*)
Bases : object

isBetween(*minValue : int*, *maxValue : int*) → bool
Retourne True si la valeur courante est comprise entre minValue et maxValue.

Paramètres
— **minValue** (*int*) – valeur minimum
— **maxValue** (*int*) – valeur maximum

Renvoie Vrai si la valeur du littéral est compris entre minValue et maxValue, inclus

Type renvoyé bool

Example

```
>>> Litteral(8).isBetween(4,12)
True
>>> Litteral(25).isBetween(4,12)
False
```

negClone() → litteral.Litteral

Produit un clone du littéral avec valeur opposée

Renvoie clone du littéral avec valeur opposée

Type renvoyé *Litteral*

Example

```
>>> Litteral(8).negClone().value
-8
```

property value

Retourne la valeur du littéral

Renvoie valeur du littéral

Type renvoyé int

Example

```
>>> Litteral(8).value
8
>>> Litteral(-15).value
-15
```

1.18 logicexpressionnodes module

Note : Les noeuds ne sont jamais modifiés. toute modification entraîne la création de clones.

class logicexpressionnodes.**AndNode**(*operand1 : Union[LogicExpressionNode, ComparaisonExpressionNode], operand2 : Union[LogicExpressionNode, ComparaisonExpressionNode]*)

Bases : *logicexpressionnodes.LogicExpressionNode*

clone() → logicexpressionnodes.LogicExpressionNode

Produit un clone de l'objet avec son arborescence

Renvoie clone

Type renvoyé *LogicExpressionNode*

logicNegateClone() → Union[comparaisonexpressionnodes.ComparaisonExpressionNode, logicexpressionnodes.LogicExpressionNode]

Complément

Renvoie noeud dont les l'expression est complémentaire, ou le noeud lui même si pas de changement

Type renvoyé Union[*ComparaisonExpressionNode*, "LogicExpressionNode"]

property operands

Accesseur

Renvoie operand

Type renvoyé Union[*LogicExpressionNode*, *ComparaisonExpressionNode*]

class logicexpressionnodes.**LogicExpressionNode**

Bases : object

abstract clone() → logicexpressionnodes.LogicExpressionNode

Crée un noeud clone

Renvoie clone

Type renvoyé *LogicExpressionNode*

Note : L'aborescence enfant est également clonée.

---

abstract logicNegateClone() → Union[comparaisonexpressionnodes.ComparaisonExpressionNode, logicexpressionnodes.LogicExpressionNode]

Calcul la négation logique de l'expression. Dans le cas not, consiste à enlever le not.

Si pas un not, alors c'est un noeud arithmétique qui n'est pas modifié.

Renvoie clone pour obtenir une négation logique

Type renvoyé Union[*ComparaisonExpressionNode*, *LogicExpressionNode*]

Note : Un clone est systèmatiquement créé

---

class logicexpressionnodes.**NotNode**(*operand : Union[LogicExpressionNode, ComparaisonExpressionNode]*)

Bases : *logicexpressionnodes.LogicExpressionNode*

clone() → logicexpressionnodes.LogicExpressionNode

Crée un noeud clone

Renvoie clone

Type renvoyé *LogicExpressionNode*

Note : L'aborescence enfant est également clonée.

---

logicNegateClone() → Union[comparaisonexpressionnodes.ComparaisonExpressionNode, logicexpressionnodes.LogicExpressionNode]

Calcul la négation logique de l'expression. Dans le cas not, consiste à enlever le not.

Si pas un not, alors c'est un noeud arithmétique qui n'est pas modifié.

Renvoie clone pour obtenir une négation logique

Type renvoyé Union[*ComparaisonExpressionNode*, "LogicExpressionNode"]

Note : Un clone est systèmatiquement créé

Example

```
>>> from arithmeticexpressionnodes import ValueNode
>>> from variable import Variable
>>> from litteral import Litteral
>>> oLitteral = ValueNode(Litteral(1))
>>> oVariable = ValueNode(Variable('x'))
>>> oComp = ComparaisonExpressionNode('<', oLitteral, oVariable)
>>> oNot = NotNode(oComp)
>>> negateONot = oNot.logicNegateClone()
>>> str(negateONot)
'(#1 < @x)'
```

property operand
Accesseur

Renvoie operand

Type renvoyé Union[*LogicExpressionNode*, *ComparaisonExpressionNode*]

class logicexpressionnodes.**OrNode**(*operand1 : Union[LogicExpressionNode, ComparaisonExpressionNode], operand2 : Union[LogicExpressionNode, ComparaisonExpressionNode]*)

Bases : *logicexpressionnodes.LogicExpressionNode*

clone() → logicexpressionnodes.LogicExpressionNode
Produit un clone de l'objet avec son arborescence

Renvoie clone

Type renvoyé *LogicExpressionNode*

logicNegateClone() → Union[comparaisonexpressionnodes.ComparaisonExpressionNode, logicexpressionnodes.LogicExpressionNode]
Complément

Renvoie noeud dont les l'expression est complémentaire, ou le noeud lui même si pas de changement

Type renvoyé Union[*ComparaisonExpressionNode*, "LogicExpressionNode"]

property operands
Accesseur

Renvoie operand

Type renvoyé Union[*LogicExpressionNode*, *ComparaisonExpressionNode*]

1.19 parsertokens module

class parsertokens.**Token**

Bases : object

Classe abstraite qui ne devrait pas être instanciée

getPriority() → int
Fonction par défaut

Renvoie priorité de l'opérateur

Type renvoyé int

Example

```
>>> TokenVariable("x").getPriority()
0
```

isOperand() → bool
Le token est-il une opérande ?

Renvoie vrai le token est un nombre ou une variable

Type renvoyé bool

Example

```
>>> TokenBinaryOperator("or").isOperand()
False
>>> TokenVariable("x").isOperand()
True
```

isOperator() → bool
Le token est-il une opérateur de calcul ?

Renvoie vrai le token est un opérateur, binaire ou unaire

Type renvoyé bool

Example

```
>>> TokenBinaryOperator("or").isOperator()
True
>>> TokenVariable("x").isOperator()
False
```

classmethod test(*expression : str*) → bool
Chaque type de noeud est associé à une expression régulière

Paramètres **expression** (*str*) – expression à tester

Renvoie vrai si l'expression valide l'expression régulière

Type renvoyé bool

Example

```
>>> TokenBinaryOperator.test("+")
True
>>> TokenBinaryOperator.test("2 + 3")
False
```

class parsertokens.**TokenBinaryOperator**(*operator : str*)
Bases : *parsertokens.Token*

getPriority() → int

Renvoie priorité de l'opérateur

Type renvoyé int

Example

```
>>> TokenBinaryOperator("or").getPriority()
1
>>> TokenBinaryOperator("and").getPriority()
3
>>> TokenBinaryOperator("<").getPriority()
4
>>> TokenBinaryOperator("+").getPriority()
5
>>> TokenBinaryOperator("|").getPriority()
6
```

property operator
Accesseur

Renvoie opérateur

Type renvoyé str

Example

```
>>> TokenBinaryOperator("or").operator
'or'
>>> TokenBinaryOperator("+").operator
'+'
```

regex = '<=|==|>=|!=|[\\^<>+\\-*\\/%&|]|and|or'

toNode(*operandsList : List[Union[arithmeticexpressionnodes.ArithmeticExpressionNode, comparaisonexpressionnodes.ComparaisonExpressionNode, logicexpressionnodes.LogicExpressionNode]]*) → Union[arithmeticexpressionnodes.ArithmeticExpressionNode, comparaisonexpressionnodes.ComparaisonExpressionNode, logicexpressionnodes.LogicExpressionNode, None]

Conversion en objet noeud

Paramètres **operandsList** (*List[Union[ArithmeticExpressionNode, ComparaisonExpressionNode, LogicExpressionNode, None]]*) – opérandes enfants

Renvoie noeud binaire expression correspondant

Type renvoyé Union[*ArithmeticExpressionNode*, *ComparaisonExpressionNode*, *LogicExpressionNode*, None]

class parsertokens.**TokenNumber**(*expression*)

Bases : *parsertokens.Token*

regex = '[0-9]+'

toNode()

Conversion en objet noeud

Note : Crée un objet Litteral correspondant

Renvoie noeud valeur correspondant

Type renvoyé *ValueNode*

property value

Accesseur

Renvoie valeur

Type renvoyé int

Example

```
>>> TokenNumber("17").value
17
```

class parsertokens.**TokenParenthesis**(*expression*)

Bases : *parsertokens.Token*

isOpening() → bool

Renvoie vrai si la parenthèse est ouvrante

Type renvoyé bool

Example

```
>>> TokenParenthesis("(").isOpening()
True
>>> TokenParenthesis(")").isOpening()
False
```

regex = '\\(|\\)'

class parsertokens.**TokenUnaryOperator**(*operator : str*)

Bases : *parsertokens.Token*

getPriority() → int

Renvoie priorité de l'opérateur

Type renvoyé int

Example

```
>>> TokenUnaryOperator("not").getPriority()
2
>>> TokenUnaryOperator("~").getPriority()
6
```

property operator

Accesseur

Renvoie opérateur

Type renvoyé str

Example

```
>>> TokenUnaryOperator("not").operator
'not'
>>> TokenUnaryOperator("-").operator
'-'
```

regex = '~|not'

toNode(*operandsList : List[Union[arithmeticexpressionnodes.ArithmeticExpressionNode, comparaisonexpressionnodes.ComparaisonExpressionNode, logicexpressionnodes.LogicExpressionNode]]*) → Union[arithmeticexpressionnodes.ArithmeticExpressionNode, comparaisonexpressionnodes.ComparaisonExpressionNode, logicexpressionnodes.LogicExpressionNode, None]

Conversion en objet noeud

Paramètres **operandsList** (*list[Union[ArithmeticExpressionNode, ComparaisonExpressionNode, LogicExpressionNode]]*) – opérandes enfants

Renvoie noeud unaire ou valeur correspondant

Type renvoyé Union[*ArithmeticExpressionNode*, *ComparaisonExpressionNode*, *LogicExpressionNode*, None]

un - unaire sur un littéral est aussitôt convertit en l'opposé de ce littéral

class parsertokens.**TokenVariable**(*name*)

Bases : *parsertokens.Token*

RESERVED_NAMES = ('and', 'or', 'not', 'while', 'if', 'else', 'elif')

regex = '[a-zA-Z][a-zA-Z_0-9]*'

classmethod test(*expression : str*) → bool

Teste si l'expression correspond à nom de variable valide

Paramètres **expression** (*str*) – expression à tester

Renvoie vrai si l'expression valide l'expression régulière

Type renvoyé bool

Example

```
>>> TokenVariable.test("x")
True
>>> TokenVariable.test("if")
False
>>> TokenVariable.test("x+y")
False
```

toNode()
Conversion en objet noeud

Note : Crée un objet Variable correspondant

Renvoie noeud valeur correspondant

Type renvoyé *ValueNode*

property value
Accesseur

Renvoie expression

Type renvoyé str

Example

```
>>> TokenVariable("x").value
'x'
```

1.20 processorengine module

class processorengine.**Commands**
Bases : dict

class processorengine.**EngineAttributes**
Bases : dict

class processorengine.**ProcessorEngine**(*name : str = 'default'*)
Bases : object

AVAILABLE_ENGINES = [('Processeur 16 bits', 'default'), ('Processeur 12 bits', '12bits

property dataBits
Accesseur

Renvoie nombre de bits utilisés pour l'encodage d'une donnée en mémoire

Type renvoyé int

Example

```
>>> ProcessorEngine().dataBits
16
```

getAsmCommand(*commandDesc : str*) → str
Renvoie le nom de commande assembleur de la commande demandée. '' si introuvable.

Paramètres **commandDesc** (*str*) – nom de la commande

Renvoie commande assembleur

Type renvoyé str

Example

```
>>> ProcessorEngine().getAsmCommand("*")
'MULT'
```

```
>>> ProcessorEngine().getAsmCommand("&")
'AND'
```

```
>>> ProcessorEngine().getAsmCommand("?")
''
```

getComparaisonSymbolsAvailables() → List[str]

Accesseur

Renvoie liste des symboles de comparaison disponibles avec ce modèle de processeur

Type renvoyé list[str]

Example

```
>>> ProcessorEngine().getComparaisonSymbolsAvailables()
['<', '>', '==', '!=']
```

getLitteralAsmCommand(*commandDesc : str*) → str

Renvoie le nom de commande assembleur de la commande demandée, dans sa version acceptant un littéral. `''` si introuvable.

Paramètres **commandDesc** (*str*) – nom de la commande

Renvoie commande assembleur

Type renvoyé str

Example

```
>>> ProcessorEngine().getLitteralAsmCommand("*")
'MULT'
```

```
>>> ProcessorEngine().getLitteralAsmCommand("?")
''
```

getLitteralMaxSizeIn(*commandDesc : str*) → int

Considérant une commande, détermine le nombre de bits utilisés par l'encodage des attributs de la commande et déduit le nombre de bits laissés pour le codage en nombre positif d'un éventuel littéral, et donc la taille maximal de ce littéral.

Paramètres **commandDesc** (*str*) – commande à utiliser

Renvoie valeur maximale acceptable du littéral

Type renvoyé int

Example

```
>>> ProcessorEngine().getLitteralMaxSizeIn("*")
63
```

getLitteralOpcode(*commandDesc : str*) → str

Renvoie l'opcode de la commande demandée dans sa version acceptant un littéral. `''` si introuvable.

Paramètres **commandDesc** (*str*) – nom de la commande

Renvoie opcode sous forme binaire

Type renvoyé str

Example

```
>>> ProcessorEngine().getLitteralOpcode("*")
'1010'
```

```
>>> ProcessorEngine().getLitteralOpcode("?")
''
```

getOpcode(*commandDesc : str*) → str

Renvoie l'opcode de la commande demandée. '' si introuvable.

Paramètres **commandDesc** (*str*) – nom de la commande

Renvoie opcode sous forme binaire

Type renvoyé str

Example

```
>>> ProcessorEngine().getOpcode("*")
'0110010'
```

```
>>> ProcessorEngine().getOpcode("?")
''
```

hasNEG() → bool

Le modèle de processeur possède-t-il un - unaire ?

Renvoie vrai s'il en possède un

Type renvoyé bool

Example

```
>>> ProcessorEngine().hasNEG()
True
```

```
>>> ProcessorEngine("12bits").hasNEG()
False
```

hasOperator(*operator : str*) → bool

Le modèle de processeur possède-t-il l'opérateur demandé ?

Paramètres **operator** (*str*) – nom de l'opérateur

Renvoie Vrai s'il le possède

Type renvoyé bool

Example

```
>>> ProcessorEngine().hasOperator("*")
True
```

```
>>> ProcessorEngine().hasOperator("?")
False
```

instructionDecode(*binary : Union[int, str]*) → Tuple[str, Sequence[int], int, int]

Pour une instruction, fait le décodage en renvoyant le descriptif commande, les opérandes registres et un éventuel opérande non registre

Paramètres **binary** (*int ou str*) – code binaire

Result tuple contenant la commande, les opérandes registres et l'éventuel opérande spéciale (adresse ou littéral), -1 si pas de spéciale, taille en bits de l'éventuel littéral.

Type renvoyé Tuple[str, Tuple[int], int, int]

Example

```
>>> ProcessorEngine().instructionDecode('0110011001010011')
('/', (1, 2, 3), -1, 0)
```

litteralOperatorAvailable(*commandDesc : str*, *litteral : litteral.Litteral*) → bool
Teste si la commande peut s'éxécuter dans une version acceptant un littéral, avec ce littéral en particulier. Il faut que la commande accepte les littéraux et que le codage de ce littéral soit possible dans l'espace laissé par cette commande.

Paramètres
— **commandDesc** (*str*) – commande à utiliser
— **litteral** (Litteral) – littéral à utiliser

Renvoie vrai si la commande est utilisable avec ce littéral

Type renvoyé bool

Example

```
>>> ProcessorEngine().litteralOperatorAvailable("*", Litteral(1))
True
```

```
>>> ProcessorEngine().litteralOperatorAvailable("*",␣
↪Litteral(10000))
False
```

property regBits
Accesseur

Renvoie nombre de bits utilisés pour l'encodage de l'adresse d'un registre

Type renvoyé int

Example

```
>>> ProcessorEngine().regBits
3
```

registersNumber() → int
Calcul le nombre de registres considérant l'adressage disponible

Renvoie nombre de registre

Type renvoyé int

Example

```
>>> ProcessorEngine().registersNumber()
8
```

ualOutputIsFree() → bool
Accesseur

Renvoie Vrai si on peut choisir le registre de sortie de l'UAL

Type renvoyé bool

Example

```
>>> ProcessorEngine().ualOutputIsFree()
True
```

```
>>> ProcessorEngine("12bits").ualOutputIsFree()
False
```

valueFitsInMemory(*value : int*, *posValue : bool*) → bool
Teste si une valeur a une valeur qui pourra être codée en mémoire

Paramètres
— **value** (*int*) – valeur à tester

— **posValue** (*bool*) – la valeur doit être codée en nombre positif

Renvoie la valeur peut être codée en mémoire

Type renvoyé bool

Example

```
>>> ProcessorEngine().valueFitsInMemory(10, False)
True
```

```
>>> ProcessorEngine().valueFitsInMemory(60000, True)
True
```

```
>>> ProcessorEngine().valueFitsInMemory(60000, False)
False
```

1.21 structuresnodes module

class structuresnodes.**AffectationNode**(*lineNumber : int, variableCible : variable.Variable, expression : arithmeticexpressionnodes.ArithmeticExpressionNode*)

Bases : *structuresnodes.StructureNode*

property cible

Accesseur : retourne la variable cible de l'affectation.

Renvoie variable cible.

Type renvoyé *Variable*

property expression

Accesseur : retourne l'expression dont le résultat doit être affecté à la variable cible.

Renvoie expression arithmétique dont le résultat doit être affecté à la variable cible.

Type renvoyé *ArithmeticExpressionNode*

class structuresnodes.**IfElseNode**(*ifLineNumber : int, condition : Union[logicexpressionnodes.LogicExpressionNode, comparaisonexpressionnodes.ComparaisonExpressionNode], ifChildren : List[structuresnodes.StructureNode], elseLineNumber : int, elseChildren : List[structuresnodes.StructureNode]*)

Bases : *structuresnodes.IfNode*

class structuresnodes.**IfNode**(*lineNumber : int, condition : Union[logicexpressionnodes.LogicExpressionNode, comparaisonexpressionnodes.ComparaisonExpressionNode], children : List[structuresnodes.StructureNode]*)

Bases : *structuresnodes.StructureNode*

class structuresnodes.**InputNode**(*lineNumber : int, variableCible : variable.Variable*)

Bases : *structuresnodes.StructureNode*

property cible

Accesseur : retourne la variable cible du input.

Renvoie variable cible.

Type renvoyé *Variable*

class structuresnodes.**JumpNode**(*lineNumber : int*, *cible : structuresnodes.StructureNode*, *condition : Optional[comparaisonexpressionnodes.ComparaisonExpressionNode] = None*)

Bases : *structuresnodes.StructureNode*

property cible

Accesseur

Renvoie cible du saut

Type renvoyé *StructureNode*

getCondition() → Optional[comparaisonexpressionnodes.ComparaisonExpressionNode]

Accesseur

Renvoie condition du saut

Type renvoyé Optional[*ComparaisonExpressionNode*]

setCible(*cible : structuresnodes.StructureNode*) → None

Assigne une nouvelle cible

Paramètres **cible** (StructureNode) – nouvelle cible

class structuresnodes.**PrintNode**(*lineNumber : int*, *expression : arithmeticexpressionnodes.ArithmeticExpressionNode*)

Bases : *structuresnodes.StructureNode*

property expression

Accesseur : retourne l'expression dont le résultat doit être affiché.

Renvoie expression arithmétique dont le résultat doit être affiché.

Type renvoyé *ArithmeticExpressionNode*

class structuresnodes.**SimpleNode**(*snType : str = ''*)

Bases : *structuresnodes.StructureNode*

property snType

Accesseur

Renvoie snType

Type renvoyé str

class structuresnodes.**StructureNode**

Bases : *linkedlistnode.LinkedListNode*

assignLabel() → label.Label

Assigne un label au noeud s'il n'en a pas déjà un :return : label de l'item :rtype : Label

property label

Accesseur

Renvoie label de l'item

Type renvoyé Optional[*Label*]

labelToStr() → str

Renvoie label en str, « » si pas de label

Type renvoyé str

property lineNumber

Accesseur pour le numéro de ligne d'origine de cet élément

Renvoie numéro de ligne d'origine

Type renvoyé int

class structuresnodes.**StructureNodeList**(*items : List[LinkedListNode]*)

Bases : *linkedlistnode.LinkedList*

delete(*nodeToDel : linkedlistnode.LinkedListNode*) → bool
supprime l'élément node

Paramètres **nodeToDel** (LinkedListNode) – élément à supprimer

Renvoie suppression effectuée

Type renvoyé bool

linearize(*csl : List[str]*) → None
Crée la vesion linéaire de l'ensemble de la structure

Paramètres **csl** (*List[str]*) – liste des comparaisons permises par le processeur utilisé

tabulatedStr()
Transtypage :return : chaîne de caractères avec une tabulation à chaque ligne :rtype : str

class structuresnodes.**WhileNode**(*lineNumber : int, condition : Union[logicexpressionnodes.LogicExpressionNode, comparaisonexpressionnodes.ComparaisonExpressionNode], children : List[structuresnodes.StructureNode]*)
Bases : *structuresnodes.IfNode*

1.22 variable module

class variable.**Variable**(*nom : str, value : int = 0*)
Bases : object

getValueBinary(*wordSize : int*) → str
Retourne chaîne de caractère représentant le code CA2 de self._value, pour un mot de taille wordSize bits

Paramètres **wordSize** (*int*) – taille du mot binaire

Renvoie code CA2 de la valeur initiale de la variable, sur wordSize bits

Type renvoyé str

Example

```
>>> Variable("x",45).getValueBinary(8)
'00101101'
```

```
>>> Variable("x",-45).getValueBinary(8)
'11010011'
```

```
>>> Variable("x",1000).getValueBinary(8)
Traceback (most recent call last):
...
errors.CompilationError: @x : Variable de valeur trop grande !
```

property name
Retourne le nom de la variable

Renvoie nom de la variable

Type renvoyé str

Example

```
>>> Variable("x").name
'x'
```

> **Avertissement :** Il est possible que l'on crée plusieurs variables pour un même nom

property value
Retourne la valeur initiale de la variable

Renvoie valeur initiale de la variable

Type renvoyé int

Example

```
>>> Variable("x").value
0
```

```
>>> Variable("x",15).value
15
```

1.23 widgets module

class widgets.**BufferWidget**(*parent*, *bufferComp*, ***options*)
Bases : tkinter.LabelFrame
BACKGROUND = 'white'
BUFFER_LINES = 5
MAX_BUFFER_LENGTH = 30
MODES = ('bin', 'hex', 'dec')
SAISIE_COLS = 18
bufferize(*evt*)
onreadempty(*params*)
onreadwrite(*params*)
refreshStrBuffer()
resetMessage()
selectMode(*mode*)

class widgets.**InputCodeWidget**(*parent*, *textCode*)
Bases : tkinter.LabelFrame
COLS = 50
LINES = 30
MSG_LINES = 3
clearProgramInput()
highlightLine(*lineNumber*)
property textCode
writeProgramInput(*text*)

class widgets.**MemoryWidget**(*parent*, *memory*, ***kwargs*)
Bases : tkinter.LabelFrame
Panneau affichant le contenu d'une mémoire
BACKGROUND = 'white'
HL_BACKGROUND = 'orange3'
HL_COLOR = 'white'
MODES = ('bin', 'hex', 'dec')
cols = 30

highlightLine(*index : int*) → None
Mise en surbrillance d'une ligne

Paramètres **index** (*int*) – numéro de ligne, à partir de 0

lineNumberFormat = '{:03d}: '

lines = 25

onfill(*params : Any*) → None
Rafraichissement de l'affichage quand le composant mémoire associé écrit à une adresse qui n'était pas encore utilisée

Paramètres **params** (*Any*) – paramètres liés à l'événement. Inutiles ici.

onread(*params : Dict[str, Any]*) → None
Callback pour la réaction une lecture mémoire

Paramètres **params** (*Dict[str, Any]*) – paramètres liés à l'événement. On utilise l'entier "index"

onwrite(*params : Dict[str, Any]*) → None
Callback pour la réaction une écriture mémoire

Paramètres **params** (*Dict[str, Any]*) – paramètres liés à l'événement. On utilise l'entier "index" et le DataValue "writed"

onwriteaddress(*params : Dict[str, Any]*) → None
Callback pour la réaction à l'écriture de l'adresse

Paramètres **params** (*Dict[str, Any]*) – paramètres liés à l'événement. On utilise le DataValue "address"

Note : n'a aucun effet si l'executeur n'est pas de type Memory

---

refresh() → None
Rafraichissement de l'affichage, après lecture du contenu du composant d'exécution associé

selectMode(*mode*)
Sélection du mode d'affichage :param mode : mode choisi, parmi "dec", "bin", "hex" :type mode : str

writeAddress(*address : executeurcomponents.DataValue*) → None
Modifie l'adresse dans le registre adresse, met en surbrillance la ligne à l'adresse sélectionnée

Paramètres **address** (DataValue) – adresse choisie

writeValueInLine(*value : executeurcomponents.DataValue*, *index : int*) → None
Écrire une valeur à une certaine ligne

Paramètres
- **value** (DataValue) – valeur à écrire
- **index** (*int*) – numéro de ligne

class widgets.**RegisterWidget**(*parent*, *register*, ***kwargs*)
Bases : tkinter.LabelFrame

BACKGROUND = 'white'

HL_BACKGROUND = 'orange3'

MODES = ('bin', 'hex', 'dec')

onwrite(*params*)

refresh()

selectMode(*mode*)

writeValue(*value*)

class widgets.**ScreenWidget**(*parent*, *screen*, ***options*)
Bases : tkinter.LabelFrame

BACKGROUND = 'white'

MODES = ('bin', 'hex', 'dec')
SCREEN_COLS = 18
SCREEN_LINES = 5
addLine(*line*)
clearScreen(*evt*)
onclear(*params*)
onwrite(*params*)
refresh()
selectMode(*mode*)

class widgets.**SimulationWidget**(*parent*, *parentWidget*, *executeur*, *textCode*, *asm*, *mode*)
Bases : tkinter.Frame
addMessage(*message*)
highlightCodeLine(*currentAsmLine : int*) → None
pour un numéro de ligne en mémoire, trouve le numéro de la ligne correspondante dans le programme d'origine et le met en surbrillance
selectDisplay(*mode*)
show()
stepRun()
property textCode

class widgets.**TextWidget**(*parent*, *text*, ***kwargs*)
Bases : tkinter.Frame
Panneau affichant un texte avec possibilité de mettre une ligne en surbrillance, de numéroter les lignes
BACKGROUND = 'white'
HL_BACKGROUND = 'orange3'
HL_COLOR = 'white'
MIN_TAB_SIZE = 3
clear() → None
Efface le contenu du texte
clearHighlight() → None
Annule toute surbrillance
cols = 30
highlightLine(*lineIndex : int*) → None
Paramètres **lineIndex** (*int*) – ligne à mettre en surbrillance

---

Note : la première ligne a l'index 0

---

insert(*text : str*) → None
Ajoute un texte à la suite
Paramètres **text** (*str*) – texte à ajouter
lineNumberFormat = '{:03d}: '
lineNumberOffset = 0
lines = 25
property text
Accesseur :return : texte en cours d'affichage :rtype : str

class widgets.**UalWidget**(*parent*, *ual*, ***kwargs*)
Bases : tkinter.LabelFrame
BACKGROUND = 'white'

MODES = ('bin', 'hex', 'dec')
oncalc(*params*)
onsetoperation(*params*)
onwriteop1(*params*)
onwriteop2(*params*)
refresh()
selectMode(*mode*)

CHAPITRE 2

Indices and tables

— genindex
— modindex
— search

Index des modules Python