

Explication des classes

Introduction :

Ce document recense toutes les classes présentes dans implémentation de la ZAM. La partie graphique est traitée dans « implantationGraphique ». Les méthodes et variables de ces classes sont expliqués ici dans le style suivant :

Le code tel qu'il est dans le programme.

Les explications.

Cette classe MaClasse est un exemple.

```
class MaClasse( variables ) extends Classe {
```

Elle possède une constante et une variable

```
val constante /
```

```
var variable /
```

Et une méthode très étrange

```
def méthode( arguments ) : retour = {  
}
```

```
}
```

Sommaire

AST

Classe AST : Cette classe s'occupe de gérer le programme courant à exécuter. Celui ci est stocké sous une suite d'instruction que l'on peut augmenter ou accéder grâce à un pointeur de code.

Tree est ce tableau d'instruction

```
class AST (tree : ArrayBuffer[Instruction]){
```

Retourne sous forme d'une String, l'AST courant.

```
override def toString
```

Ajout d'instruction en fin d'arbre.

@in : instr, l'instruction à ajouter

```
def add(instr : Instruction)
```

Retourne une instruction pour un indice donné.

@in : pc : Pointeur de code, l'indice voulu

```
def get(pc : Int) : Instruction
```

```
}
```

BlockT

Classe BlockT : Énumération des types de bloc possibles. Un bloc est une valeur particulière.

```
object BlockT extends Enumeration {  
  type blockT = Value  
  val foward_t, infix_t, object_t, closure_t, lazy_t,  
  abstract_t, string_t, double_t, doublearray_t, custom_t  
  = Value  
}
```

Evaluator

Classe Evaluator : Elle est affectée à l'évaluation des instructions et gère l'exécution des instructions sur l'environnement de la ZAM. Il possède le code de toutes les instructions exécutable par le simulateur.

```
class Evaluator {
```

Executer une instruction. Une exécution nécessite un environnement et la thread qui exécute.

@in : inst, l'instruction à exécuter

@in : env, l'environnement d'exécution (global et local)

@in : itT, l'indice de la thread qui s'exécute, cela permet de retrouver et de ne modifier que l'environnement local concerné.

```
def exec(inst : Instruction, env : GlobalState, itT : Int)
```

```
}
```

GlobalState

Classe GlobalState : L'environnement global, il stocke toutes les données du programme en exécution, aussi bien les globales que les locales.

```
class GlobalState {
```

HashMap représentant les variables globales (Tas)

```
  val glob = new HashMap[Int, Value]()
```

Tableau des threads présentes, elles représentent les différentes variables locales.

```
  val Threads = new ThreadTable(new ArrayBuffer[ThreadState])
```

Fonction de gestion du tas.

Accesseur au tas

```
  def getglob
```

Ajout ou modification d'une variable globale

@in i, la clef de la valeur

@in v, la nouvelle valeur

```
  def addglob(i : Int, v : Value)
```

Accesseur à une valeur globale, cette valeur est le retour de fonction.

@in i, la clef de la valeur

```
  def atglob(i : Int) : Value
```

Fonction de gestion des threads.

Ajout d'une thread

@in t, la thread à ajouter

```
  def pushthread(t : ThreadState)
```

Suppression d'une thread (déprécié pour le moment)

@in i, l'indice de la thread à supprimer

```
  def removethread(i : Int)
```

Accesseur d'une thread, retourne la thread à l'indice donné

@in i, l'indice de la thread

```
  def getthread(i : Int) : ThreadState
```

Retourne l'état de l'environnement globale (uniquement) sous forme d'une chaîne.

```
override def toString()
```

Retourne l'état de la thread (uniquement) sous forme d'une chaîne.

@in : t, l'indice de la thread

```
def printT(t : Int)  
}
```

Instruction

Classe Instruction : Cette classe s'occupe de définir les instructions présentes dans le simulateur. Toute instruction est fille de cette classe, grâce au case class elle retourne sous forme de chaîne les instructions à afficher. On trouve dans cette page toutes les instructions et leur formulation.

```
abstract class Instruction () {
```

Retourne l'instruction sous forme de chaîne

```
override def toString
```

Utilisé par toString pour gérer le case class

```
def print(t : Instruction) : String
```

```
}
```

Les instructions de la ZAM. Certaines possèdent des arguments pour pouvoir les factoriser un minimum. (ex : acc0, acc1 = acc(int))

```
case class Acc(arg : Int) extends Instruction
```

```
case class Push extends Instruction
```

```
case class Pushacc(arg : Int) extends Instruction
```

```
case class Pop(arg : Int) extends Instruction
```

```
case class Assign(arg : Int) extends Instruction
```

```
case class Envacc(arg : Int) extends Instruction
```

```
case class Pushenvacc(arg : Int) extends Instruction
```

```
case class Push_Retaddr(pc : Int) extends Instruction
```

```
case class Apply(arg : Int) extends Instruction //check_stacks
```

```
case class Appterm(nargs : Int, slotSize : Int) extends Instruction
```

```
case class Return(sp : Int) extends Instruction
```

```
case class Restart extends Instruction
```

```
case class Grab(required : Int) extends Instruction
```

```
case class Closure(nvars : Int) extends Instruction
```

```
case class Closurerec(nfuncs : Int, nvars : Int) extends Instruction
```



```
case class Pushoffsetclosure(arg : Int) extends Instruction
case class Offsetclosure(arg : Int) extends Instruction
case class Pushoffsetclosurem(arg : Int) extends Instruction
case class Offsetclosurem(arg : Int) extends Instruction
```

```
case class Pushgetglobal(arg : Int) extends Instruction
case class Getglobal(arg : Int) extends Instruction
case class Getglobalfield(arg : Int, field : Int) extends Instruction
case class Setglobal(arg : Int) extends Instruction
case class Pushgetglobalfield(arg: Int, field : Int) extends Instruction
```

```
case class Getfield(arg : Int) extends Instruction
case class Getfloatfield(arg : Int) extends Instruction
case class Setfield(arg : Int) extends Instruction
case class Setfloatfield(arg : Int) extends Instruction
case class Pushatom(arg : Int) extends Instruction
case class Atom(arg : Int) extends Instruction
case class Makeblock(size : Int, typ : Int) extends Instruction
case class Makefloatblock(size : Int) extends Instruction
```

```
case class Vectlength extends Instruction
case class Getvectitem extends Instruction
case class Setvectitem extends Instruction
```

```
case class Branch(flag : Int) extends Instruction
case class Branchif(flag : Int) extends Instruction
case class Branchifnot(flag : Int) extends Instruction
case class Switch(sizes : Int, flags : Array[Int]) extends Instruction
case class Boolnot extends Instruction
```

```
case class C_Call(narg : Int) extends Instruction
```

```
case class Const(arg : Int) extends Instruction
case class Pushconst(arg : Int) extends Instruction
```

```
case class Pushconstint(arg : Int) extends Instruction
```

```
case class Negint() extends Instruction
```

```
case class Addint() extends Instruction
```

```
case class Subint() extends Instruction
```

```
case class Mulint() extends Instruction
```

```
case class Divint() extends Instruction
```

```
case class Modint() extends Instruction
```

```
case class Andint() extends Instruction
```

```
case class Orint() extends Instruction
```

```
case class Xorint() extends Instruction
```

```
case class Lslint() extends Instruction
```

```
case class Lsrint() extends Instruction
```

```
case class Asrint() extends Instruction
```

```
case class Integer_comparision(tst : String) extends Instruction
```

```
case class Integer_branch_comparision(arg : Long, pc : Int, tst: String) extends  
Instruction
```

```
case class Offsetint(arg : Int) extends Instruction
```

```
case class Offsetref(arg : Int) extends Instruction
```

```
case class Isint() extends Instruction
```

Simulator

Classe Simulator : Le simulateur est la classe qui lie les composants entre eux. Il possède un AST, un environnement global, un gestionnaire de sauvegarde et un évaluateur d'instruction. (Peut être fera t il aussi le lien entre la partie graphique et la ZAM...) C'est le point d'entrée de la ZAM physique.

```
class Simulator {
```

Les données représentant les différents composants de la ZAM.

```
val MyAST = new AST(new ArrayBuffer[Instruction])
```

```
val MyEnv = new GlobalState
```

```
val MySManager = new StateManager
```

```
val MyEval = new Evaluator
```

Accesseurs aux composants

```
def AST
```

```
def Env
```

```
def Manager
```

```
def Eval
```

Prepare le programme (plus tard crée un AST à partir d'un fichier ZAM ou OCAML)

```
def Preparer()
```

Fait avancer de n instructions la thread t (et donc son pc)

@in : t, l'indice de la thread à avancer

@in : n, le pas, nombre d'instructions à exécuter

```
def Avancer(t : Int, n : Int)
```

Fait reculer de n instructions la thread t (et donc son pc)

@in : t, l'indice de la thread à reculer

@in : n, le nombre d'instruction à remonter

```
def Revenir(t : Int, n : Int)
```

Retourne l'AST sous forme d'une chaîne.

override def toString

Retourne la thread t sous forme de chaîne

@in : t, la thread à afficher

def printthread(t: Int)

Retourne l'environnement global sous forme de chaîne

def printenv()

}

State

Classe State : La classe state est un état de l'environnement pour un instant donné. Il stocke donc toutes les données de l'environnement global et de la thread concernée.

```
class State (pc : Int, sp : Array[Value], accu : Value, env : Array[Value],  
  glob : HashMap[Int, Value], extra_args : Int){
```

Les accesseurs pour retrouver la valeur d'un champ lors de la sauvegarde.

Retourne un Int

```
  def getpc
```

Retourne un tableau de Valeur

```
  def getsp
```

Retourne une Valeur

```
  def getaccu
```

Retourne un tableau de Valeur

```
  def getenv
```

Retourne une Hashmap

```
  def getglob
```

Retourne un Int

```
  def getextra
```

```
}
```

StateManager

Classe StateManager : Le gestionnaire d'états possède toutes les sauvegardes d'état. Elles sont stockées sous forme d'un double tableau, chaque premier indice correspond à une thread. Pour chaque thread un état est stocké par pas dans l'AST.

```
class StateManager {
```

Le double tableau

```
val chemin = new ArrayBuffer[ArrayBuffer[State]](0)
```

Sauver un état et une thread, seul les informations locales à la thread et l'environnement global est sauvé.

@in : MyEnv, l'environnement global

@in : itT, la thread concernée

```
def save(MyEnv : GlobalState, itT : Int)
```

Restaurer un état. Les données actuelles de la thread sont remplacées par les sauvegardes. Les variables globales sauvegardées sont remises en état courantes. Les données globales présentes mais non sauvegardées ne sont pas touchées.

@in : MyEnv , l'environnement global

@in : itT, l'indice de la thread concernée

@in : n, l'indice de l'état à restaurer

```
def restaurer(MyEnv : GlobalState, itT : Int, n : Int)
```

```
}
```

ThreadState

Classe ThreadState : L'état d'une thread concerne toutes les variables locales dont elle a besoin. Chaque thread possède les siennes et ne peuvent modifier celles des autres. Seules les variables globales sont communes aux threads.

```
class ThreadState {
```

La pile

```
var sp = new ArrayBuffer[Value](0)
```

L'accumulateur

```
var accu : Value = new Zamentier(0) //unit
```

L'environnement

```
var env = new Array[Value](4)
```

Le pointeur de code

```
var pc = 0
```

L'extra arg

```
var extra_args = 0
```

Gestion de l'environnement

Accesseur, retourne une valeur stockée à l'indice i

@in : i, l'indice de la valeur

```
def getenv(i : Int)
```

Modifieur, place la valeur v à l'indice i de l'environnement

@in : i, indice voulu

@in : v, Nouvelle Valeur

```
def setenv(i : Int, v : Value)
```

Gestion du pointeur de code

Accesseur, retourne le pointeur de code. L'indice de l'instruction courante dans l'AST.

```
def getpc
```

Modifieur, place le nouveau pointeur de code. @in : newpc,

```
def setpc(newpc : Int)
```

Gestion de l'accumulateur

Accesseur, retourne la Valeur de l'accumulateur

def getaccu

Modifieur, place la nouvelle Valeur dans l'accumulateur

@in : v, nouvelle valeur

def setaccu(v : Value)

Gestion de l'extra arg

Accesseur, retourne l'entier correspondant à l'extra arg

def getextra

Modifieur, place le nouvel extra arg

@in : v, nouvelle valeur

def setextra(v : Int)

Retourne sous forme de chaine l'état de la thread

override def toString()

}

ThreadTable

Classe ThreadTable: Cette classe s'occupe de gérer les threads présentes dans le programme. Elle sont stockée sous forme d'un tableau d'état de thread.

```
class ThreadTable(table : ArrayBuffer[ThreadState]) {
```

Ajouter une thread à l'arrière du tableau

@in : t, la nouvelle thread à ajouter

```
def push(t : ThreadState)
```

Supprimer une thread à l'indice i

@in : i, l'indice de la thread à supprimer (déprécié)

```
def remove(i : Int)
```

Accesseur d'une thread à l'indice i

@in : i, l'indice de la thread voulue

```
def get(i : Int)
```

```
}
```

Value

Classe Value: Cette classe représente n'importe quelle valeur de la ZAM.

Une valeur est une sous classe de la classe Value

```
abstract class Value {  
}
```

Classe Zamlong: Cette classe représente un long

```
class Zamlong (value : Long) extends Value {  
    Retourne la valeur de ce long  
    def getval  
    Retourne la valeur dans une chaine  
    override def toString  
}
```

Classe Zamentier: Cette classe représente un entier

```
class Zamentier (value : Int) extends Value {  
    Retourne la valeur de cet entier  
    def getval  
    Retourne la valeur dans une chaine  
    override def toString  
}
```

Classe Zamblock: Cette classe représente un bloc. Il comporte un entête composé d'un tag, de sa taille et de sa valeur

```
class Zamblock (tag : BlockT.blockT, size : Long, value : Array[Value] )  
extends Value {
```

Retourne la valeur de ce bloc

```
def getval
```

Retourne la taille de ce bloc

```
def getsize
```

Retourne le tag de ce bloc

```
def gettag
```

Accesseur à la valeur d'un indice du bloc

@in : n, l'indice

```
def at(n : Int)
```

Modifier la valeur d'un indice du bloc

@in : n, l'indice

@in : v, la nouvelle Valeur

```
def set(n : Int, v : Value)
```

Retourne le bloc sous forme d'une chaîne

```
override def toString
```

```
}
```