PONCELET Clément PSTL 2012

Simulateur ZAM

Remerciements :

Sommaire

Introduction :

Ce PSTL concerne le Simulateur ZAM, avec pour encadrants Mr Peschanski et Mr Canou. Il a pour but d'implémenter un logiciel répondant aux fonctionnalités d'un simulateur.

Qu'est ce qu'un Simulateur ?

Bien souvent les logiciels permettant d'exécuter les codes ne font qu'exclusivement cette fonction. Un simulateur va ajouter certaines utilités à l'exécution d'un programme. Notre but ici était de pouvoir remonter le cours d'une exécution, d'ajouter des break points , de gérer séparément les différentes threads présentes et d'avoir un aperçu des variables présentes dans le programme.
Un simulateur ne se contente pas d'offrir ces fonctionnalités mais implémente aussi la machine virtuelle concernée par un langage. Etant dédiée à Ocaml elle implémente sa machine virtuelle attitrée : *La ZAM.*

Le langage :

Ces besoins nécessitent une bonne architecture Object pour recréer solidement le contexte de la ZAM. Mais de part ses bases sur Ocaml, il se prête aussi à un aspect fonctionnel.

Pour ces deux raisons et pour une curiosité à toute épreuve, un langage dit le *'super Java'* a été choisi : *Le Scala*
La scala est un dérivé de Java et implémente toutes ses fonctionnalités. A cela s'ajoute tout l'aspect fonctionnel d'Ocaml et quelques avantages très pratiques.

Organisons nous :

Afin d'y voir un peu plus clair, nous allons étudier et expliquer les différentes parties de ce projet. Commençons par expliquer ce qu'est ce phénomène de Machine ZAM et comment est elle réalisée. Nous pourrons ensuite l'appliquer sur notre langage Scala et mieux comprendre le code du simulateur.

L'étude de l'implémentation sera présentée sous sa forme générale, pour repérer ses fonctions et ses liens, puis séparée en deux parties. Une pour sa partie *'physique'* et l'autre pour ses vues et son *'swing'* (bibliothèque graphique JAVA). Les explications du programme étant faite, la rubrique 'explication de classe' suit pour montrer l'application concrète de ces notions. Il nous restera pour finir des petites parties et quelques exemples d'utilisation du simulateur.

VM ZAM

Introduction :

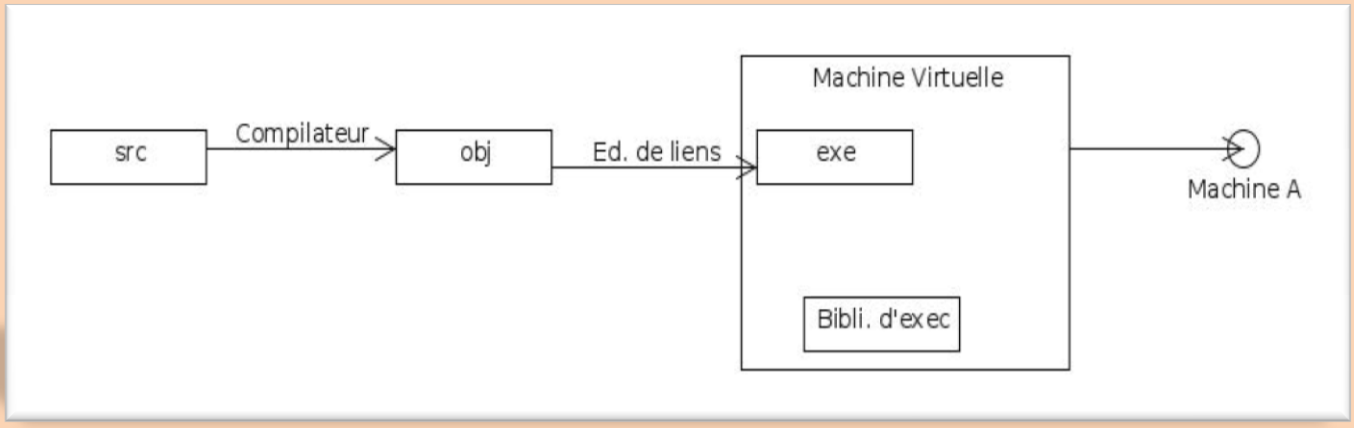
La ZAM est un langage de machine virtuelle, elle est utilisée pour compiler de l'OCaml. Nous allons étudier son fonctionnement pour pouvoir l'intégrer dans le simulateur et pouvoir reproduire son code.

Qu'est ce que la compilation et la ZAM ?

Une compilation est, pour une machine virtuelle (VM), une production de code-octet *(bytecode)* interprété ou compilé à la volée vers du code machine.

Généralement on veut exécuter un programme écrit sous un langage B mais sur une machine (pc) ne comprenant qu'un langage A. Pour se faire nous pouvons utiliser une machine virtuelle écrit sous langage A et pouvant exécuter les programmes du langage B.

Lors d'une compilation il est souvent nécessaire d'avoir une bibliothèque d'exécution *(runtime)* pour le support des langages de haut niveau (gestion mémoire, entrées/sorties, chargement dynamique, appels de méthodes, continuations, etc.….), tout ceci est intégré dans les VM.
Pour résumer en un schéma voici une compilation avec machine Virtuelle.

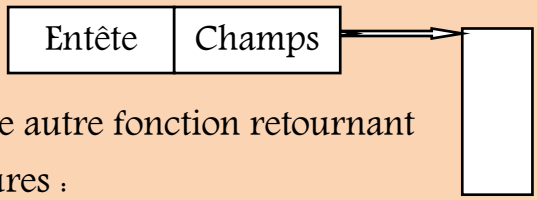


Comment est faite la ZAM ?

Le corps de la ZAM

Regardons premièrement ce que manipule la ZAM pour cela nous débuterons par parcourir le fichier interp.ml et analyser ce qu'elle exécute. Plusieurs types de valeur sont utilisés dans la machine virtuelle *les nombres*, *les pointeurs* et *les labels*. Un pointeur est enfaite un entier Pair, ils sont tous distinct et générés au début de la ZAM. Un pointeur désigne un bloc qui est une suite de donnée matérialisée ainsi :

Blocs : {entête : Int ; champs : tab<Int>}



On trouve avec la fonction génératrice, une autre fonction retournant le type de la valeur en entrée. Voici leurs signatures :

gen_ptr : Pointeur

outTypes : type

Pour pouvoir utiliser une variable il faut pouvoir l'allouer, la ZAM utilise ces fonctions pour allouer un bloc et une constante. Pour un bloc, il est placé dans le tas et le retour de la fonction est le pointeur de ce bloc dans le tas.

In : entête, champs (un entier et un tableau)
Out : Pointeur, clef du bloc inséré dans le tas

alloc_bloc(Int, Nombre) : Pointeur

In : valeur de la constante à créer
Out : Constante allouée

alloc_const (n : Entier | ptr : Pointeur | (n, constlist)) :
(Entier | Pointeur | Pointeur sur constlist dans tas)

Les labels sont des *'pointeurs d'instructions'* ils sont initialisées au début d'une exécution de programme et pour chaque instruction « label ».

Labels : Hashtbl.create 51

Préparation des labels
Int : code, le code à exécuter
prepare_labels (code) : unit

Création d'un label sur une instruction
In : pc, le pointeur de l'instruction
set_label (pc) : unit

La machine virtuelle se sert d'outils pour exécuter le code, on trouve plusieurs déclarations de ces outils lors du parcours du fichier. Un tas est créé pour pouvoir allouer des blocs et un tableau global stocke toutes les variables globales

Tas : Hash<int, blocs> (51)

Globals : Hash<int, Nombre>(89)

Des fonctions permettent d'utiliser le tableau de variables globales :

In : clef, entier
Out : Valeur globale correspondant à la clef
getglobal (Int) : Nombre

In : Clef,
In : n, Nouvelle Valeur à entrer
setglobal(Int ; Nombre) : Unit

In : ptr, pointeur du bloc voulu
Out : Bloc pointé
bloc (ptr) : Bloc

Un extra_arg, une pile et un accumulateur sont aussi utilisés par la machine virtuelle, des fonctions pour manipuler la pile suivent aussi. Pour optimiser le code l'extra_arg est placé dans un registre (il n'est autre qu'un entier), l'accumulateur qui ne stocke lui aussi qu'une valeur de tout type est une *'référence'* (un pointeur OCaml).

Extra_args : Un registre

Accu = ref

pile : Modifstack.creat _any

Ajouter dans la pile
In : v : valeur
push (v) : unit

Enlever la de tête de pile
Out : Valeur anciennement en tête de pile
pop() : v

Enlever n fois la tête de pile
In : n, le nombre de pop
Out : La dernière valeur de pop
popn (n) : v

Réinitialiser la pile
reset() : unit

Accesseur au n élément de la pile
In: n, indice de la valeur dans la pile
Out : v, pile[n]
get(n) : v

Remplacer le n° élément de la pile
In : n, l'indice à remplacer
v, la nouvelle valeur
set(n, v) : unit

Des fonctions auxiliaires sont aussi implantées pour gérer les opérations indispensables, comme les opérations arithmétiques, binaires ou les comparaisons.

Permet les opérations arithmétiques, ces opérations entrent le résultat dans l'accumulateur.
Opération Unaire sur entier dans l'accumulateur :
In : f, la fonction

```
do_arith_unop (f) : unit
```

Opération binaire entre l'accumulateur et la tête de pile
In : f, la fonction

```
do_arith_unop (f) : Unit
```

Les comparaisons sont identiques qu'une opération Binaire hormis quelle renvoie le résultat sous forme de booléen
In : f, la fonction

```
do_arith_unop (f) : Entier (boolean)
```

L'Interpréteur

L'interpréteur s'occupe de traduire les fonctions de la ZAM dans son contexte précédemment défini. Il y a deux types d'instructions, les simples et les complexes, une instruction est simple si elle ne change pas le flot d'exécution du programme.

do_simpleinstr (instruction)

Les instructions simples sont :

Klabel	()
Knop	()
Kreadint	Lit un entier et le place dans l'accumulateur
Kreadstring	Non supporté
Kprintint	Affiche l'entier dans l'accumulateur
Kprintstring	Non supporte
Kgetglobal n (clef)	Place la valeur de la global à n dans l'accumulateur
Ksetglobal n (clef)	Place la valeur de l'accumulateur en global et l'associe à la clef n (accu = unit)
Kacc n	Place le n° élément de la pile dans l'accumulateur
Kpush	Place la valeur de l'accumulateur dans la pile
Kpop n	Enlève n valeur dans la pile (pop en tête)
Kassign n	Place dans le n° élément de la pile la valeur de l'accumulateur
Kconst cst	Alloue une constante, la valeur est placée dans l'accumulateur
Kmakeblock size, tag	Alloue un tableau (tab) de taille size du type dans l'accumulateur, il est rempli par les size champs de la pile (dans l'ordre de pop). Le bloc créé a un entête tag et un champs tab. Le pointeur est ensuite placé dans l'accumulateur.
Kgetfield n	Accès au n° champs du bloc pointé par l'accumulateur. Le résultat est mis dans l'accumulateur.
K setfield n	Place au n° champs du bloc pointé par l'accumulateur, la valeur de la tête de pile (pop). L'accumulateur est mis à unit.
Kvectlength	Entre la taille du bloc pointé par l'accumulateur dans l'accumulateur

Kgetvectitem (arg en pile)	Place le n° élément du bloc pointé par l'accumulateur dans l'accumulateur. N est pris en tête de pile (pop).
Ksetvectitem (arg en pile)	Place dans le n° élément du bloc pointé par l'accumulateur la valeur v. L'accumulateur est mis à unit et n et v sont pris dans la tête de pile (dans cette ordre).
Kisint	Teste si la valeur de l'accumulateur est un entier. Place _true (1) si oui _false (0) sinon
Kenvacc n	Place dans l'accumulateur la valeur du n° champ du bloc pointé par l'environnement
Kclosure lbl, n	Crée une fermeture. Un bloc tagué 247 comprenant un tableau de n valeurs. L'accumulateur est placé dans champ [1] (si n <> 0). Les champs de 2 à n sont remplis par la tête de pile (n-1 pop). L'accumulateur prend le pointeur du bloc de fermeture créé.
Kclosurerec lblist, n	(Non géré, idem que closure, l'accu est seulement ajouté dans la pile)
Kpush_retaddr l	Met extra_args, env et un label dans la pile
Krestart	Restauration du contexte, place les variables globales (env) dans la pile (sauf la 1° qui est mis dans env). On ajoute ensuite le nombre d'argument à extra_args.
Koffsetclosure n	Place le pointeur d'environnement (env) dans l'accumulateur. (les fonctions rec n'étants pas prises en compte)

On trouve ensuite les opérations, les opérations sont faites entre l'accumulateur et la tête de pile. Le résultat est ensuite placé dans l'accumulateur.

Knegint	~- Opération unaire de négation
Kaddint	+ Opération binaire d'addition
Ksubint	- Opération binaire de soustraction
Kmulint	* Opération binaire de multiplication
Kdivint	/ Opération binaire de division
Kmodint	Mod Opération binaire de modulo
Kandint	Land Opération binaire and
Korint	Lor Opération binaire or
Kxorint	Lxor Opération binaire xor

Klslint	Lsl Opération binaire de décalage à gauche
Klsrint	Lsr Opération binaire de décalage à droite
Kasrint	Asr Opération binaire de décalage à droite arithmétique
Koffsetint n	Décalage à droite unaire (n fois)
Kboolnot	Non booléen
Koffsetref n	Ajoute n au premier champ du bloc pointé par l'accumulateur et le place à unit

Viennent les instructions de comparaisons :

Kintcomp comp	Les opérations de comparaisons sont identiques que celle des opérations. Comp peut prendre les valeurs :
Ceq	==
Cneq	<>
Clt	<
Cgt	>
Cle	<=
Cge	>=

Pour finir on trouve les instructions complexes qui modifient le pointeur de code :

Kbranch l	Exécute le saut et fait modifier le pointeur de code sur le label
Kbranchif l	Exécute le saut si accumulateur = vrai
Kstrictbranchif l	Exécute le saut si accumulateur = vrai
Kbranchifnot l	Exécute le saut si accumulateur = faux
Kstrictbranchifnot l	Exécute le saut si accumulateur = faux
Kswitch tbl_const tbl_bloc	Exécute le saut de l'indice présent dans l'accumulateur
Kapply nbargs	Applique une fonction à nbargs arguments
Kappterm (nbargs, slotsize)	Applique une fonction en remplaçant les arguments de la précédente
Kgrab n	Prend des arguments et exécute la fonction si elle possède tous ses arguments
Kstop	()
Kreturn slotsize	Retour de fonction

Les instructions étant écrites il ne reste plus que la fonction pour lancer l'exécution du programme

execute(code) : Unit

En fin de fichier on trouve également le pointeur sur la dernière instruction, l'exécution des instructions et la commande pour lancer la machine ZAM :

```
fast

In : pc, le pointeur de l'instruction courante
instr (pc) do_

##Lancement de la ZAM (main) ##
In : -v ou –verbose fichier ou fichier
Interp.execute (Parser.programme Lexer.token (Lexing.from_channel f))
```

Les Instructions ZAM : (lexer.mll / parser.mly)

L'organisation interne de la machine virtuelle ne donne pas tout à fait la syntaxe et la grammaire attendue et comprise par celle-ci. Le programme (comme on peut le voir dans la commande) passe déjà par une analyse lexicale et syntaxique. Les analyseurs syntaxique et lexical de ZAM définissent les mots clés et la syntaxe que doit avoir ce programme pour être compris par la ZAM. On trouve ces informations dans les fichiers lexer.ml et parser.mly :

Lexer :

Le lexer est celui qui indique les mots clefs du langage. Il défini ce que la ZAM reconnaît et va ensuite transformer chaque clef en token pour l'analyse syntaxique. On trouve une table qui associe les instructions du programme à un token, c'est la table keyword_table. Nous allons voir ce que la machine virtuelle comprend et en quoi elle les transforme :

Les chiffres : 0-9 : digit
Les lettres : a-zA-Z : alpha
Les alphanumériques : digit | alpha | _ | '
Les identifiants : alpha+
Les entier : (-?) digit+
Début de commentaire ((*) et les fin de commentaire (*))
Les séparateurs : ' '\t\n (ils sont ignorés)

Plusieurs niveaux de commentaires sont gérés, ils sont aussi ignorés, car ne comptent pas pour exécuter le programme.

Entrée	Token de remplacement (valeur associée)
identifiant	Instruction associée à la table (voir plus bas)
L entier	Tlabel (entier)
L entier :	Tdef_Label (entier)
[entier]	Tatom (entier)
[entier :	Topen_block (entier)
]	Tclose_block
Entier a	Tnum (entier)

Entier	Tnum (entier)
«	Tstring
‘	Tchar
,	Tcomma
/	Tslash
Eof	TEOF

D’autres entrées entraineront une levée d'exception

Parser :

Les tokens fixés c’est au tour de l’analyseur syntaxique de prendre le relai. Il nous informe que le programme doit commencer par une instruction et finir par un TOEF (end of file). Le programme est ensuite transformé en tableau d’instruction (code) :

```
programme : instructions TEOF { Array.of_list (List.rev $1)}
```

Une instruction peut être vide ou suivi d'une autre instruction, une file est alors crée avec instruction :: instructions. Les instructions sont listées en sens inverse et sont donc inversées au dessus.

```
Instructions :
    {[]}
 | instructions instruction {$2 :: $1}
;
```

La syntaxe des listes et des constantes est explicitée, on trouve donc :

La liste de nombre :

```
num_list : { [] }
        | num_list Tnum {$2 ::$1}
;
```

La liste de nombre non vide :

```
num_ne_list : Tnum { [$1] }
        | num_list Tnum {$2 ::$1}
;
```

Les constantes, nombre, char, block ou listes :

```
constant : Tnum { Const_char $1) }
        | Tatom { Const_block ($1, []) }
        | Topen_block constant_list Tclose_block { Const_block ($1,List.rev $2)}
;
```

La liste de constante

```
constant_list : {[]}
        | constant_list constant { $2 ::$1 }
;
```

Pour finir nous pouvons retrouver la syntaxe de toutes les instructions et voir en quoi elle est transformée, un tableau associant la syntaxe au label ZAM final a été fait, les $ correspond aux valeurs associées. Pour le label il est nécessaire d'avoir un 'L' puis un entier, celui-ci est transformé en TLabel (entier) et parser en Klabel $1. Le dollar 1 est la valeur de cet entier.

programme | token | ZAM

L entier	TLabel	Klabel $1
L entier :	Tdef_label	Knop
nop	Tnop	Knop
stop	Tstop	Kstop
readint	Treadint	Kreadint
readstring	Treadstring	Kreadstring
printint	Tprintint	Kprintint
printstring	Tprintstring	Kprintstring
getglobal entier	Tgetglobal Tnum	Kgetglobal $2
setglobal entier	Tsetgobal Tnum	Ksetglobal $2
acc entier	Tacc Tnum	Kacc $2
envacc entier	Tenvacc Tnum	Kenvacc $2
push	Tpush	Kpush
pop entier	Tpop Tnum	Kpop $2
assign entier	Tassign Tnum	Kassign $2
push_retaddr label	Tpush_retaddr Tlabel	Kpush_retaddr $2
apply entier	Tapply Tnum	Kapply $2
appterm entier, entier	Tappterm Tnum Tcomma Tnum	Kappterm ($2, $4)
return entier	Treturn Tnum	Kreturn $2
restart	Trestart	Krestart
grab entier	Tgrab Tnum	Kgrab $2
closure label, entier	Tclosure Tlabel Tcomma Tnum	Kclosure ($2, $4)

closurerec entier, entier …	Tclosurerec num_ne_list Tcomma Tnum	Kclosurerec (list.rev $2, $4)
offsetclosure entier	Toffsetclosure Tnum	Koffsetclosure $2
offsetref entier	Toffsetref Tnum	Koffsetref $2
const constant	Tconst constant	Kconst $2
makeblock entier, entier	Tmakeblock Tnum Tcomma Tnum	Kmakeblock ($2, $4)
getfield entier	Tgetfield Tnum	Kgetfield $2
setfield entier	Tsetfield Tnum	Ksetfield $2
vectlength	Tveclength	Kveclength
getvectitem	Tgetvectitem	Kgetvectitem
setvectitem	Tsetvectitem	Ksetvectitem
branch label	Tbranch Tlabel	Kbranch $2
branchif label	Tbranchif Tlabel	Kbranchif $2
branchifnot label	Tbranchifnot Tlabel	Kbranchifnot $2
strictbranchif label	Tstrictbranchif Tlabel	Kstrictbranchif $2
strictbranchifnot label	Tstrictbranchifnot Tlabel	Kstrictbranchifnot $2
switch entier, entier \ entier, entier	Tswitch num_list Tslash num_list	Kswitch (list.rev $2, list.rev $4)
boolnot	Tboolnot	Kboolnot
negint	Tnegint	Knegint
addint	Taddint	Kaddint
subint	Tsubint	Ksubint
mulint	Tmulint	Kmulint
divint	Tdivint	Kdivint
modint	Tmodint	Kmodint
andint	Tandint	Kandint
orint	Torint	Korint
xorint	Txorint	Kxorint
lslint	Tlslint	Klslint
lsrint	Tlsrint	Klsrint
asrint	Tasrint	Kasrint
offsetint entier	Toffsetint Tnum	Koffsetint $2
isint	Tisint	Kisint
eqint	Tcomparison(Ceq)	Kintcomp $1
neqint	Tcomparison(Cneq)	Kintcomp $1
ltint	Tcomparison(Clt)	Kintcomp $1
gtint	Tcomparison(Cgt)	Kintcomp $1
leint	Tcomparison(Cle)	Kintcomp $1
geint	Tcomparison(Cge)	Kintcomp $1

Projet en gros

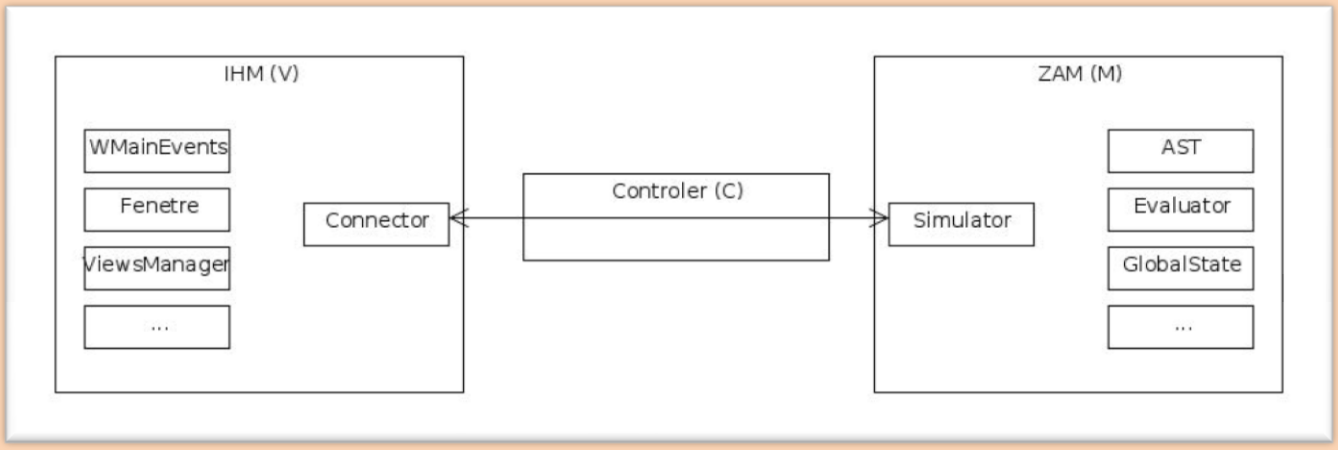
Nous voilà dans l'univers de Scala, après cette présentation de la ZAM, une adaptation et une refonte doit être entreprise pour faire tourner les rouages de la ZAM en Scala. Le Simulateur ZAM est créé sous le model MVC qui sépare le logiciel en trois parties.

Le M(odèle) représenté par la classe Simulateur. Elle est dite partie Physique ou Donnée car elle représente le corps du logiciel. Toutes les fonctionnalités voulues et exécutées par la ZAM sont ici. Il est en réalité le package ZAM. Aucuns liens avec JAVA n'a été fait ici pour garantir une indépendance.

La V(ue) représentée par la classe Connector, c'est le package IHM. Il utilise la bibliothèque Swing de JAVA pour la partie graphique. Toutes les fonctionnalités liées au vues et à l'affichage sont ici, il n'y a pas de modifications directes sur la partie Modèle du logiciel. Ceci explique la séparation des deux parties.

Le C est pour le Controleur, il répartie et aiguille les fonctionnalités requises par la Vue sur le Modèle.

Pour visualiser un peu mieux ces trois parties un petit dessin nous schématise le labyrinthe de classe du simulateur.



Un Physique « ZAM »

Maintenant qu'une vue générale a été faite voyons comment s'organise la ZAM dans ce langage. Pour faire tourner le simulateur avec toutes ses fonctions, plusieurs composants sont créés et reliés par le Simulateur, la classe principale de cette partie physique. Nous allons nous intéresser à l'implémentation des valeurs et des instructions ZAM. Ensuite nous verrons comment le simulateur s'exécute et étudierons le contexte de la machine virtuelle.

Comment Organiser et implanter les valeurs ZAM ?

Etant un langage objet chaque valeur de ZAM doit avoir sa classe et être de type valeur. En effet la machine virtuelle possède une spécification particulière de ses valeurs, cela peut être un champ décomposé ainsi :

Structure de l'entête d'un champ
For 16-bit and 32-bit architectures: (31 = 63 pour les architectures à 64 bits)

```
        +--------+-------+-----+
        | wosize | color | tag |
        +--------+-------+-----+
Bit     31       10 9     8 7   0
```

Une différence du premier bit de chaque champ permet de faire la différence entre un entier ou un pointeur de bloc. Un bloc peut être une multitude de valeur et a différents types :

Enumération des Tags :

Signification des Tags : < 251

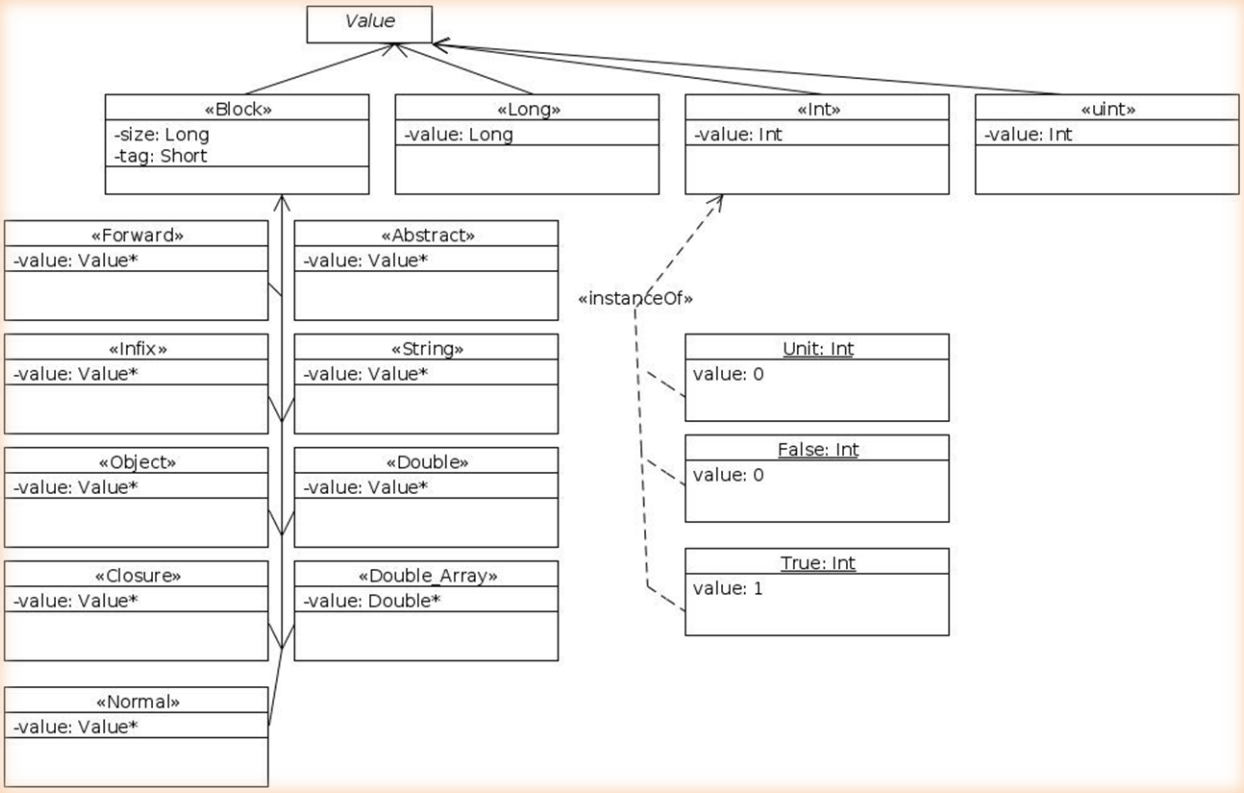
250 : Forward_tag :
249 : Infix_tag :
248 : Object_tag :
247 : Closure_tag :
246 : Lazy_tag :

Signification des Tags : >= 251

251 : Abstract_tag
252 : String_tag
253 : Double_tag
254 : Double_array_tag
255 : Custom_tag

En reprenant cette configuration et en suivant celle de la ZAM, le simulateur implémente les valeurs comme ci suit. Cela permet de retrouver et de stocker n'importe quelle valeur ZAM. Un normal_tag étant ajouté pour un tableau de n'importe quelles valeurs.

Implantation :

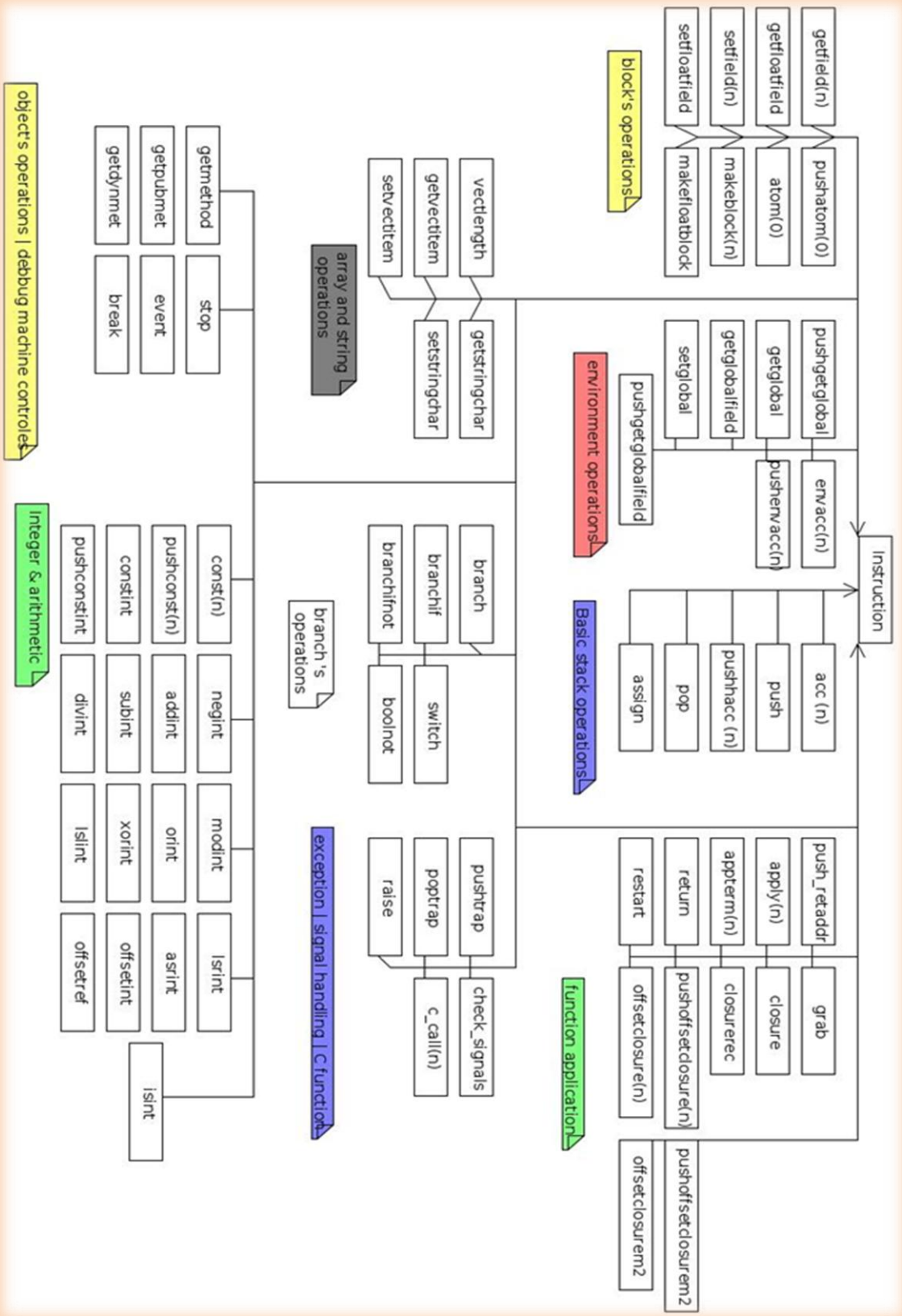


Amélioration : Un logiciel n'étant jamais au point, (par optimisation ou par manque de temps) les idées d'amélioration ou d'ajout sont commentées comme ceci.

Dans ce dessin une représentation des entiers non signés est inscrite mais non implémentée. Cette amélioration permettrait de coller d'encore plus près la réalité de l'implémentation de la ZAM

Et pour les instructions ZAM ?

L'interprète est de loin la partie principale de la machine virtuelle, elle possède le code de toutes les instructions présentent et exécutables. Le simulateur comporte tous ces codes dans une seule classe «l'Evaluator » qui les exécute selon un environnement Scala-ZAM. Voici un aperçu des instructions présentes dans la ZAM et parsées dans le simulateur.



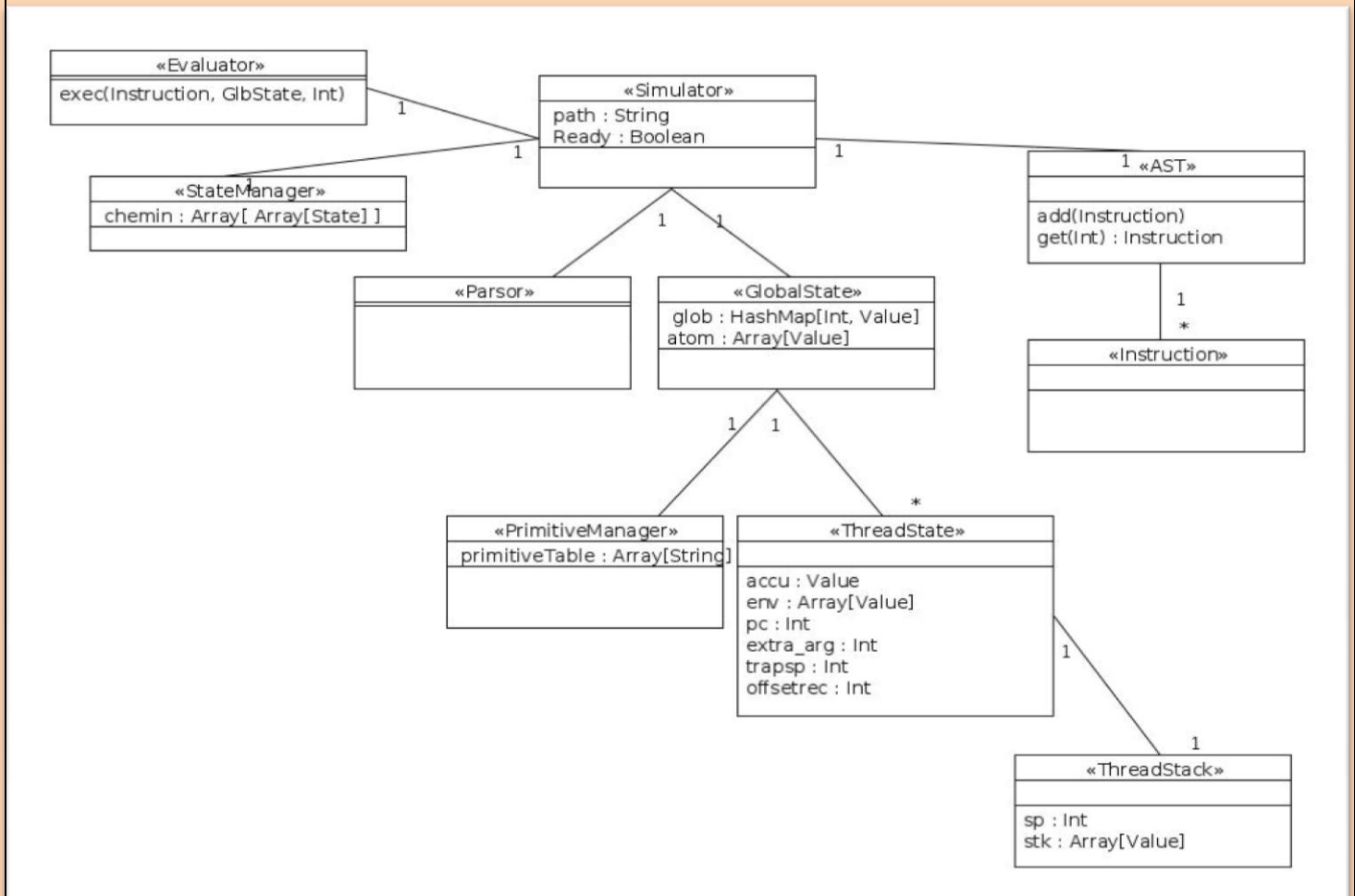
La partie des instructions pour debbug et classes ne sont pas implémentées.

Pour une simplification et une efficacité, certaines des instructions ont été factorisée en leur affectant un argument supplémentaire. Le fonctionnement de la ZAM est spécial pour les arguments des instructions en ce qu'ils sont généralement à la suite du code et donc pointé par le pc.

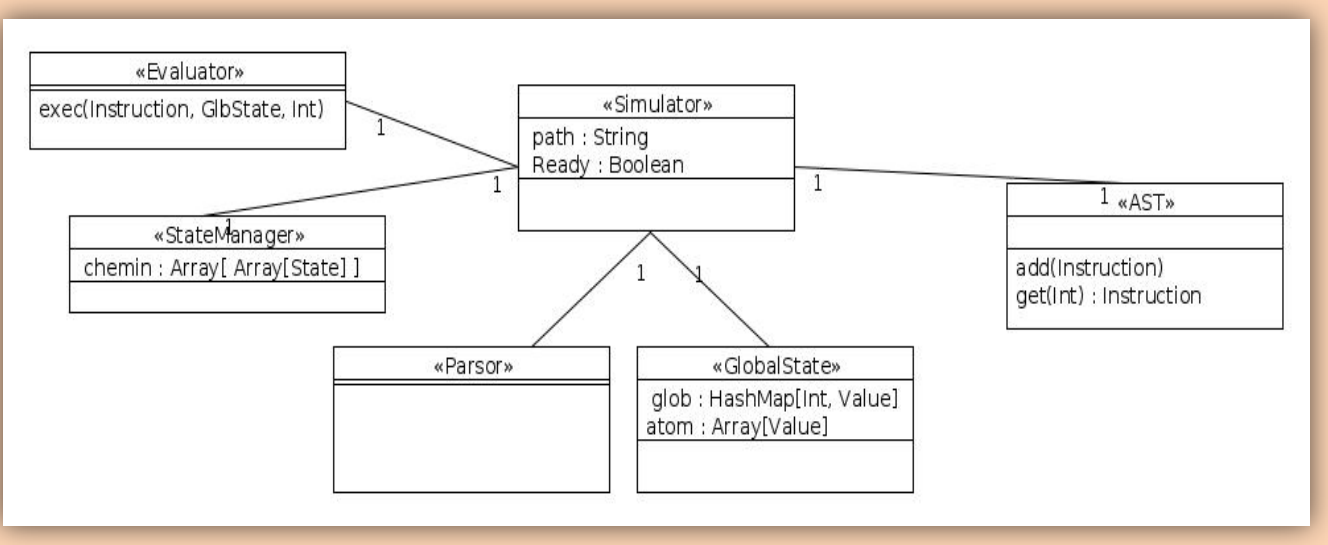
Le simulateur possède un AST comportant uniquement des instructions, tout argument est donc passé explicitement lors de la création de l'instruction et de son ajout dans l'AST. Pour minimiser les écarts entre sous parties les explications des classes elles mêmes et leur implémentation sont dans la rubrique « explication des classes ».

Physique Général :

Les deux plus gros points d'une machine virtuelle étant traités voyons comment s'organisent ce fameux noyau. Voici la partie physique du simulateur, aux deux grandes parties traitées plus haut, s'ajoutent un gestionnaire d'état et un Etat global décrivant le contexte et l'état de la ZAM.



Un cœur Simulateur :



Nous voilà au cœur de la partie physique, comme répété précédemment il connecte les différents composants satisfaisant aux demandes du projet. Il est de ce fait l'entrée et l'encapsulation de la partie physique, c'est la classe Modèle de MVC. Il possède aussi le chemin menant au .zam et/ou au .ml correspondant.

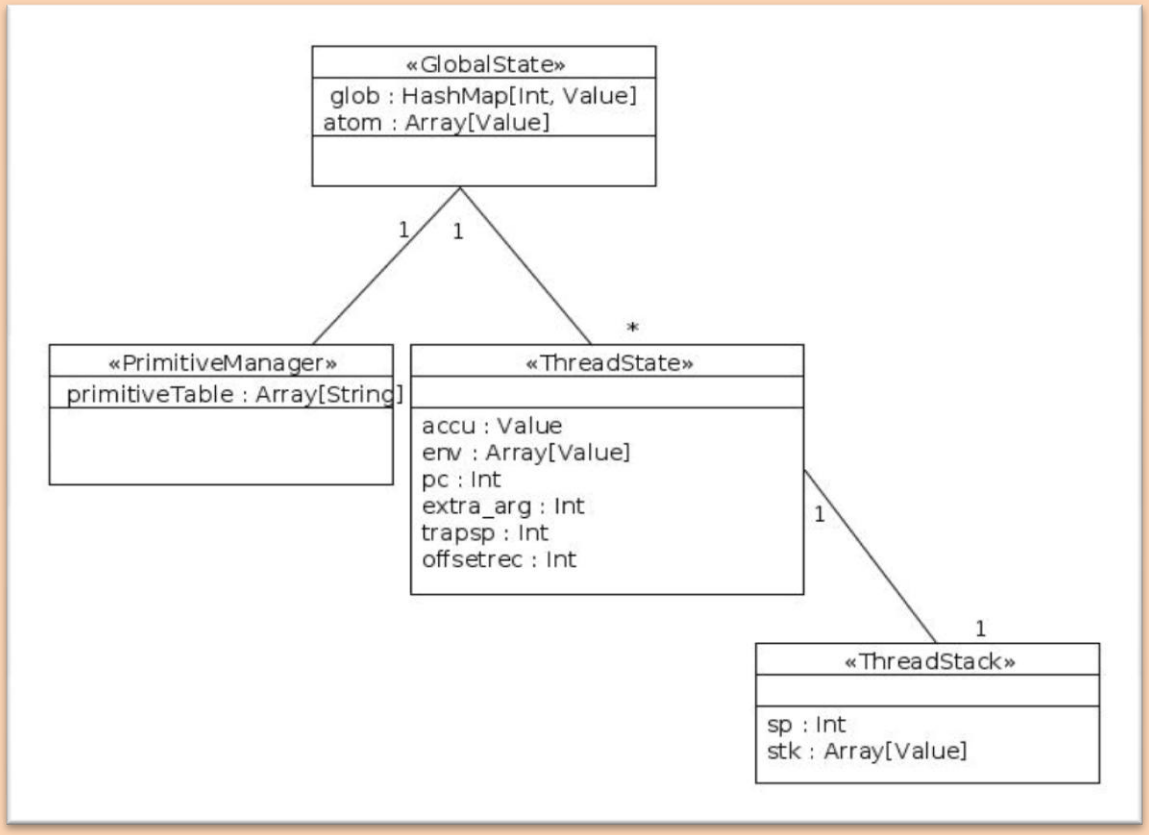
Le booléen Ready est à vrai lorsque que tout est initialisé pour pouvoir exécuter le programme, c'est à dire lorsque le fichier .zam pointé par le chemin (path) a été parsé par le parsor.

Comme le veut sa principale contrainte, il faut pouvoir avancer une thread de N « pas » dans l'AST. Cette fonction est réalisée par Avancer. A l'inverse Revenir fait reculer la thread de n pas dans l'AST.

Amélioration :

La fonction de Parsing n'a pas été implémentée, c'est pourquoi des exemples ont été créés en fin de rapport.

Contexte :



Le contexte est la représentation de l'environnement ZAM dans le simulateur, il prend en compte les multiples threads présentent ainsi qu'un environnement global. Comme vus plus haut, il possède aussi des outils de programmation présents dans les machines virtuelles.

GlobalState

Cette classe représente l'environnement global du simulateur. Il contient toutes les variables globales (associées à une clef dans une hashmap) et un tableau d'atomes pour les différents blocs. L'atome est utilisé pour éviter d'allouer plusieurs fois un tableau vide, cette fonctionnalité est présente dans la ZAM.

La « runtime », une bibliothèque d'exécution, est utilisée pour implanter les primitives C pouvant être utilisées dans le simulateur. La classe *PrimitiveManager* stocke toutes les primitives connues et les exécute comme l'évaluateur pour une instruction ZAM

ThreadState

Notre environnement global contient aussi toutes les variables locales à chaque thread, il contient donc un état de thread pour chaque nouvelle thread. Cet état garde les données relatives à une thread et reflète celui de la machine virtuelle. On y trouve :

Le pointeur de code donnant l'itération courante de la thread dans l'AST du simulateur.

L'accumulateur, une valeur ZAM

L'environnement d'une thread, qui est un tableau de valeur utilisé pour la gestion de fonction

L'extra_arg, un entier utilisé pour les arguments de fonction

Une pile « stack », représentée par la classe ThreadStack, on y trouve son pointeur de pile (sp) et les valeurs présentes dans la pile. Une classe a été créée pour permettre une gestion libre du pointeur de code.

Chaque Thread possède son environnement ZAM et sont indépendantes entre elles, seul l'environnement global les relient.

StateManager :

Cette classe est responsable de la gestion des sauvegardes d'état en vus d'une restitution ultérieure. Elle rempli à elle seule la fonctionnalité de revenir à une instruction précédente. Pour une marche arrière le contexte sauvegardé au moment de l'exécution est simplement restauré.

L'environnement locale de la thread est remis entièrement à son état sauvegardé.

Pour l'environnement global, seules les variables sauvegardées précédemment sont remises en état, les autres sont gardées telle quelles.

Cela permet une stratégie de sauvegarde et de restauration indépendante au reste du programme. Son implémentation actuelle est simple, pour chaque thread et à chaque pas, les états globaux et locaux sont sauvegardés.

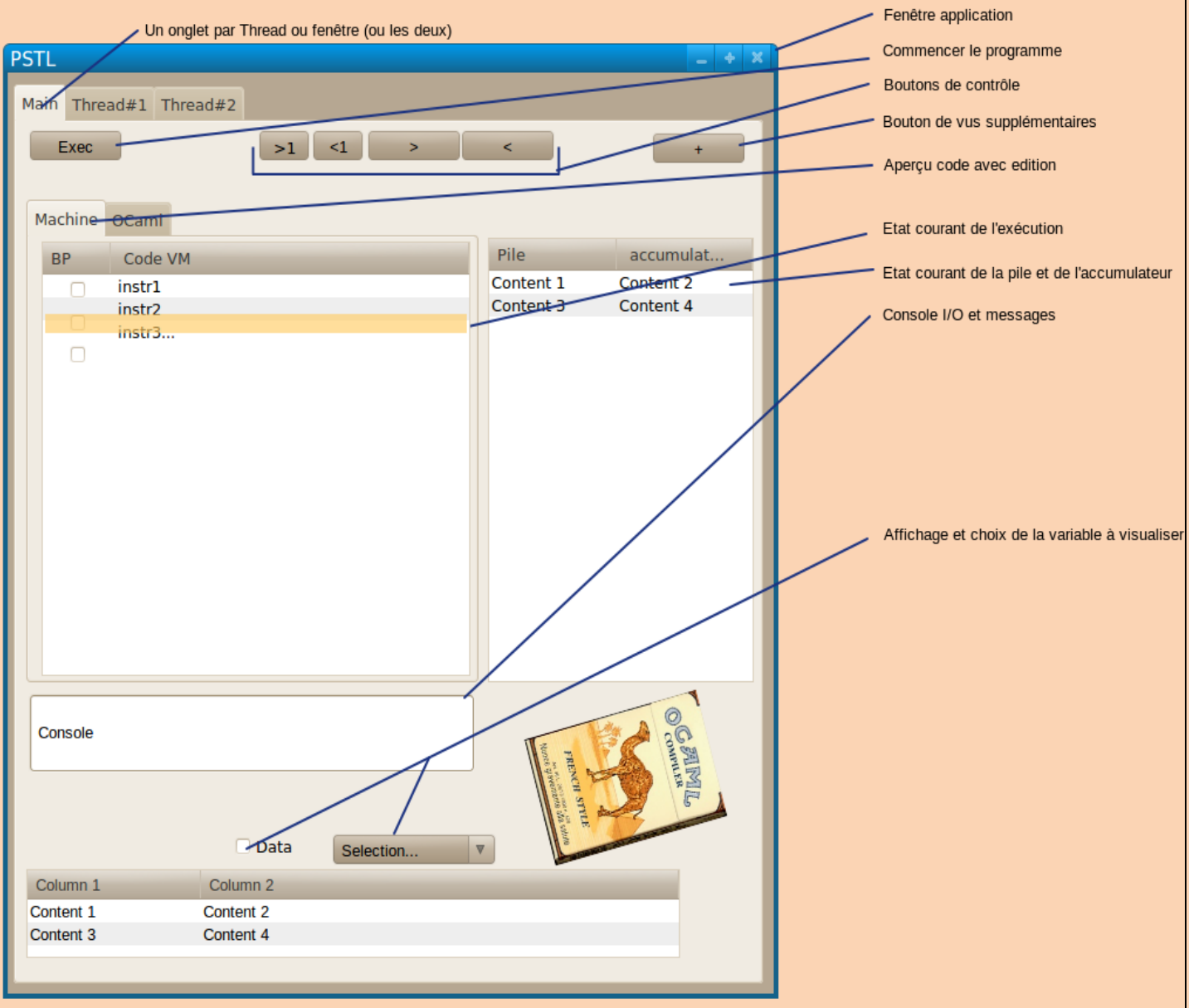
Amélioration :

Cette méthode est de loin la plus simple, mais aussi la plus gourmande. Une autre implémentation par 'delta' serait plus efficace. Cette méthode consisterai à sauvegarder les états tous les n pas. Une restauration à une itération m entre ces n consisterait simplement à restaurer l'état n précédent puis à exécuter les instructions entre ce n et le m.

Simulateur ZAM Vue

Le simulateur a pour principal but de simuler une exécution de plusieurs threads simultanément. Dans cette optique chaque thread possède sa fenêtre, la première étant la thread main. Une gestion de vues est aussi faite pour pouvoir choisir et organiser les informations que l'on souhaite.

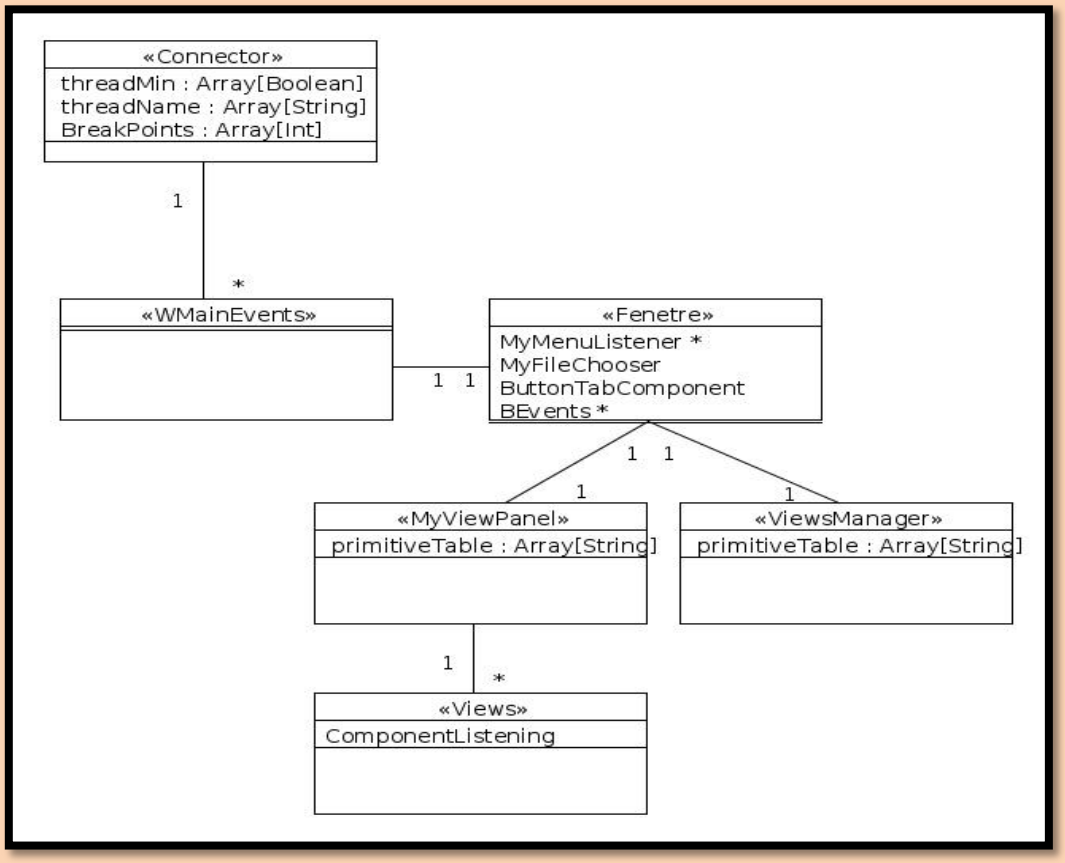
Au commencement du projet un 'patron' de la vue a été fait, visant à définir les affichages et l'organisation visuelle du simulateur. Ces exemples ont été créé sur pencil un logiciel libre sous firefox, voilà les visuels retenus :



La partie Vue du simulateur englobe tout ce qui est Swing, la classe faisant le point et englobant toutes ces classes est le Connector. Nous allons voir comment le swing de JAVA est utilisé et quelle est la responsabilité de chaque classe. Nous verrons donc une vue générale de cette partie pour s'occuper de chaque classe ou greffon en suite.

Une vue générale :

Chaque vue s'occupant d'afficher certaines données de la partie modèle, toute fenêtre et composant possède un lien vers le connecteur. Grâce à son id, elle peut lui demander les informations qu'elle a besoin d'afficher. Pour une raison de clarté ces liens ne sont pas mis dans le schéma suivant.



Comment connecter toutes ces threads ?

Connector

En plus d'être la classe Vue du modèle MVC, le connecteur garde pour chaque thread son nom et son état. Il garde aussi les breaks points voulus par l'utilisateur. Cela permet un affichage rapide et une gestion facile des pas effectués par une thread. Une thread a deux état, elle peut avoir sa propre fenêtre ou se faire « minimiser », elle apparaîtra dans un onglet sur la fenêtre de la thread main. A l'inverse elle peut se voir maximiser et retourner dans sa propre fenêtre.

Pour les demandes qu'il ne peut satisfaire, il retransmet au controler.

Fenêtre

WmainEvent et *Fenêtre* sont les deux classes responsables des fenêtres. La première s'occupe des évènements qui peuvent survenir sur la fenêtre, c'est pour cette raison qu'elle apparaît avant la classe fenêtre. Cette deuxième gère le visuel et possède plusieurs parties.

Le menu permet de gérer les vues désirées pour la thread courante, ce choix concerne seulement la thread de la fenêtre (donc main si vous êtes dans la première fenêtre). Pour la fenêtre main une gestion de chargement de fichier est aussi effectuée.

Cette gestion concerne les classes :

MyMenuListener : Elle gère les évènements survenus sur le menu pour la gestion du chemin et de la fermeture.

MyFileChooser : Il est utilisé par MyMenuListener pour gérer le chargement de fichier.

ViewsManager : Il s'occupe des vues que l'utilisateur à choisi. Les vues et leur ordre sont gérés par cette classe, qui stocke tout ceci dans un tableau. Ce tableau peut être donné en argument ce qui permet de sauvegarder le choix de l'utilisateur pour chaque thread. Il est aussi responsable du changement de décision dans le menu view.

Les boutons permettent de gérer l'état de la thread. Il est possible grâce à eux d'avancer d'un ou de plusieurs pas et à l'inverse de reculer. Les pas s'effectuent dans l'AST correspondant au code ZAM du programme.
Le bouton exec permet de recréer l'AST depuis le fichier .zam courant. Attention le programme redémarrera depuis le début.

Amélioration :

Le bouton option permet d'afficher d'autres vus comme par exemple des statistiques ou des données d'une autre thread.

Cette gestion concerne les classes :

Bevents : Gestion des évènements et des boutons.

L'affichage des vues est centralisé par *MyViewPanel*. Une gestion d'onglet est faite selon si la fenêtre est celle du main ou non, la première thread étant automatiquement minimisée. Le panel affiche les vues dans l'ordre donné par ViewManager, chaque vues possédant son id il est possible de retrouver le panel correspondant.

Une vue a un ou plusieurs panel, leur ordre est modifiable via la souris et leur gestion effectuée par le menu *view* ou le bouton *option*. Pour reconnaître la thread affectée à cette vus (si elle en a une) L'idée est défini sur un nombre :
Le nombre des centaines correspond à la thread et celui des unités à la vue.

Pour 101, la vue correspond à la première vue de la thread 1. Les ids des vues sont définis comme suit :

0 → Vue du code
1 → Vue contexte (pile et accumulateur)
2 → Vue de l'environnement global
3 → Vue de la console
4 → Vue des données

Utiliser l'id d'une thread pour la vue de l'environnement global ou de la console est inutile.

Les vues

Vue Code :

La vue code se sépare en deux parties, une Ocaml et une ZAM. Le simulateur ne gérant que la ZAM, les breaks points sont visibles et implémentés pour cette vue uniquement.

Amélioration :

Une ligne en surbrillance indiquant l'emplacement de la thread courante sur le code ZAM et Ocaml est une amélioration utile. Il est aussi pratique de pouvoir modifier le code ZAM et le code Ocaml pour une mise à jour et une prise en compte directe des modifications.

La gestion de modification du code ZAM n'a pas été implémentée. L'idée étant de pouvoir modifier le code courant n'importe où. Une re-compilation et une ré exécution s'effectuant si cette modification est portée au dessus du pc d'une des threads.

Vue Données :

La vue donnée est une vue plus précise des données du simulateur.

Amélioration :

Elle permettrait entre autre, de donner une précision de chaque tableau présent dans le code (simplifier par 'tableau tag size' dans les autres vues). Elle pourrait aussi afficher toute autre données de la thread.

Cette partie désigne les classes :

CodeView : Panel affichant le code Ocaml et ZAM.
ComponentListening : Gérant les évènements Souris sur les vues
ConsoleView : Panel affichant la console du programme.
ContextView : Panel affichant la pile et l'accumulateur d'une thread
DataManager : Gestion des évènements et du tableau de la vue Donnée
DataView : Panel affichant la vue Donnée
EnvView : Panel affichant l'environnement global
OcamlView : Panel affichant le code Ocaml
ZamView : Panel affichant le code ZAM et gérant les évènements liés aux Breaks points

Controleur :

La partie M et V étant présentées il ne reste plus que le controler. La classe controler fait le lien entre les demandes des vues et les données du modèle. La classe Simulateur englobant toutes les données du modèle, le rôle du contrôleur est donc de transmettre les différents ordres au simulateur en s'assurant qu'un programme est bien en train de s'exécuter.

Tests :

Afin d'assurer la qualité du logiciel un package Test a été créé comportant les tests que le simulateur doit passer.

Amélioration :

Cette partie n'est implémentée qu'a titre d'exemple, la couverture est proche du 0. Pour une bonne qualité chaque partie du modèle doit être testée et validée.

Explication des Classes

Oui mais le code dans tout ça ?

Ce document recense toutes les classes présentes dans le logiciel. C'est un complément aux explications des parties précédentes. 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 constante1
    var  variable1

    Et une méthode très étrange
    def méthode( arguments) : retour = {
    }
}
```

Nous allons décrire chaque classe par package en débutant par celui de ZAM, puis IHM et enfin Controler et Tests.

Package ZAM

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
```

Retourne la taille de l'arbre

```
def size : Int

}
```

BlockT

Classe BlockT : C'est une énumération des types de bloc possibles. Un bloc est une valeur particulière. Il peut être vu comme un tableau.

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

ErreurZam

Classe ErreurZam : Exception ZAM, elle représente toutes les exceptions qui peuvent arriver durant le programme. Le message est inscrit dans la console.

Err est le message d'erreur de l'exception

```
Class ErreurZam(err : String) extends Exception(err) {

}
```

Evaluator

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

```
class Evaluator {
```

Executer une instruction. Une exécution nécessite un environnement et la thread qui l'exécute.
@in : inst, l'instruction à exécuter
@in : env, l'environnement d'exécution (global)
@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 ArrayBuffer[ThreadState]
```

Tableau de blocs vides, cela permet d'allouer un seul tableau vide par bloc. Il est utilise pour l'instruction Atom.

```
val atom = Array[value]
```

Runtime servant à lier les primitives de C_Call

```
val runtime = new PrimitiveManager
```

Fonctions 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
```

Fonctions 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 global (uniquement) sous forme d'une chaine.

```
override def toString()
```

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

@in : t, l'indice de la thread

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

Instruction

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

```
abstract class Instruction () {

    Retourne l'instruction sous forme de chaine
    override def toString

    Utilisé par toString pour gérer le class case
    def print(t : Instruction) : String
}
```

Les instructions de la ZAM. Certaines possèdent plus d'arguments pour pouvoir les factoriser. (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 Getstringchar extends Instruction
case class Setstringchar 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 Pushtrap(trappc : Int) extends Instruction
case class Poptrap extends Instruction
case class Raise extend 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 Constint(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
```

Parsor

Classe Parsor : La classe Parseur est chargée de créer un AST depuis un fichier .zam

```
class Parsor {

    Parse le fichier indiqué par le path et crée l'AST correspondant
    @in : path, chemin du fichier a parser
    @in-out ast, l'AST (les instructions sont ajoutées en fin de l'AST)
    def parse(path : String, ast : AST)
}
```

PrimitiveManager

Classe PrimitiveManager: La classe PrimitiveManager contient toutes les primitives C supportées par le Simulateur. Si une instruction n'est pas trouvée une exception ZAM est lancée. Les primitives sont gérées par la fonction C_Call prenant en argument l'indice de la primitive dans le tableau primitiveTable.

```
class PrimitiveManager {

 Tableau contenant les noms des primitives implantées
 val primitiveTable

        Exécute la primitive itprim pour la thread avec narg arguments
        @in : itprim, indice de la primitive voulue
        @in : narg, nombre d'argument stocké dans la pile
        @in : thread, la thread executant la primitive.
        def run(itprim : Int, narg : Int, thread : ThreadState)
}
```

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. C'est le point d'entrée de la ZAM physique.

```
class Simulator {

        Le chemin du fichier .zam // même chemin .ml pour son correspondant ocaml
        var path
        Observateur vérifiant si un AST est présent dans le simulateur
        var Ready

        Les données représentant les différents composants de la ZAM.
        val Parsor = new Parsor
        val AST = new AST(new ArrayBuffer[Instruction])
        val Env = new GlobalState
        val Smanager = new StateManager
        val Eval = new Evaluator
```

Les accesseurs du chemin, du code Ocaml et Zam et de l'observateur

```
def getPath
def getOcmlCode
def getZamCode
def IsReady
```

Mettre un nouveau path

@in : newpath, le nouveau chemin du fichier

```
def setprog(newpath : String)
```

Préparer le programme (création d'un AST à partir d'un fichier ZAM)

```
def Preparer
```

Renvoi un Booléen selon si le simulateur à un AST ou non (accesseur à l'observateur Ready)

```
def IsReady
```

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)
```

Re-initialise le simulateur

```
def Restart
```

Retourne l'AST sous forme d'une chaine.

```
override def toString
```

Retourne la thread t sous forme de chaine

@in : t, la thread à afficher

```
def printthread(t: Int)
```

Retourne l'environnement global sous forme de chaine

```
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 local 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 représentant tous les états de chaque thread

```
val chemin = new ArrayBuffer[ArrayBuffer[State]]
```

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

@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 touchée.

@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)
}
```

ThreadStack

Classe ThreadStack : C'est cette classe qui gère la pile Zam. Chaque thread possède une pile, son pointeur de pile (Sp) peut être changé pour certaines fonctionnalités.

```
class ThreadStack {
```

Les données stockées dans la pile

```
val stk = new Arraybuffer
```

Le pointeur de pile

```
private var sp
```

Accesseur au pointeur de pile

```
def getSp
```

Change le pointeur de pile, il est préférable d'utiliser pop pour ne pas perdre de la place mémoire

@in : newsp, le nouveau pointeur (< sp)

```
def changeSp(newSp : Int)
```

Ajoute un élément en tête de pile (pointé par sp)

@in : v, la valeur zam à ajouter

```
def push(v : Value)
```

Retourne l'élément en tête de pile (pointé par Sp)

```
def getVal
```

Retourne l'élément à l'indice i (en partant de sp)

@in : i, l'indice de la valeur voulue. (=sp[i])

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

Modifier la valeur a l'indice i

@in : i, l'indice de la valeur voulue. (=sp[i])

@in : v, la nouvelle valeur

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

Enlève la valeur en tête de pile

```
def pop
```

Vider la pile

```
def clear
```

Retourne la taille de la pile

```
def size
```

Retourne la pile sous forme de chaine

```
override def toString
```

}

ThreadState

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

```
class ThreadState {

    Le pointeur de pile d'exception
    var trapsp
    La pile
    var stack = new ThreadStack
    L'accumulateur
    var accu : Value = new Zamentier(0) //unit
    L'environnement
    var env = new Arraybuffer[Value]
    Le pointeur de code
    var pc = 0
    L'extra arg
    var extra_args = 0

    Gestion des exceptions (trap)
    Modificateur, place le nouveau pointeur d'exception
    @in : sp, pointeur de l'exception dans la pile
    def setTrapsp(sp : Int)
    Accesseur du pointeur d'exception
    def getTrapsp

    Gestion de l'environnement
    Accesseur, retourne une valeur stockée à l'indice i
    @in : i, l'indice de la valeur
    def getenv(i : Int)
    Modificateur, 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)
    Retourne la taille de l'environnement
    def sizeEnv
    Vide l'environnement
    def clearenv
```

Gestion du pointeur de code

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

```
def getpc
```

Modifier le pointeur de code.

@in : newpc, le nouveau pointeur

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

Gestion de l'accumulateur

Accesseur, retourne la Valeur de l'accumulateur

```
def getaccu
```

Modifier la 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
```

Modifier l'extra arg

@in : v, nouvelle valeur

```
def setextra(v : Int)
```

Retourne sous forme de chaine l'état de la thread

```
override def toString()
}
```

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 Zamint: Cette classe représente un entier

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

Classe Zamuint: Cette classe représente un entier non signé (non terminé)

```
class Zamint (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 (son tableau)
        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 chaine
        override def toString
}
```

Bevents

Classe BEvents: Elle rattrape les évènements des boutons de contrôle. Chaque bouton est fille de cette classe, le type de fonction à exécuter se fait sur le nom de celui-ci. name est donc le nom du bouton, base est le lien vers le connector et it l'indice de la thread affichée. Beaucoup de classe de cette partie sont organisée de comme cela.

```
Abstract class BEvents (name: String) extends Jbutton (name) with ActionListener {

    Fonction appelée lors d'une réception d'évènement. Typiquement lors d'un clic sur le
    bouton.
    @in : e, l'évènement
    def actionPerformed(e : ActionEvent)

    Fonction utilisée pour utiliser le case class. L'action selon le bouton est traitée ici.
    @in : b, le bouton appelé
    @in : e, l'évènement
    def action(b : Bevents, e : ActionEvent)
}
```

Grâce au case class, tous les boutons sont définis ici. Ils sont reliés au connecteur et possède l'indice de la thread. Ils étendent Bevents et lui passent leur nom.

```
case class ExecBtn(name: String, base : Connector, it : Int)
case class FowardBtn(name: String, base : Connector, it : Int)
case class BackBtn(name: String, base : Connector, it : Int)
case class ToBrkBtn(name: String, base : Connector, it : Int)
case class RestartBtn(name: String, base : Connector, it : Int)
case class OptBtn(name: String, base : Connector, it : Int)
```

ButtonTabComponent

Classe ButtonTabComponent : Elle est responsable des évènements sur la fermeture d'un onglet. Si une thread est fermée elle recrée sa fenêtre, si c'est la thread main, le programme se termine. Elle est liée au connecteur et possède son indice mais modifie aussi le gestionnaire d'onglet (pane). Elle stocke aussi son label pour retrouver son indice dans l'onglet.

```
class ButtonTabComponent(base : Connector, it : Int, pane : JtabbedPane, mylabel : JPanel ) extends Jbutton(« x ») with ActionListener {
```

Initialise le bouton du composant en stockant son indice et en créant un texte à afficher lors d'un survol

```
def init
```

Fonction appelée lors d'une réception d'évènement. Lors d'un clic sur la croix.

@in : e, l'évènement

```
def actionPerformed(e : ActionEvent)
}
```

CodeView

Classe CodeView : Cette classe affiche un onglet dans lequel se trouve la vue du code Ocaml et zam. Elle est liée au connecteur et à son indice (servant à retrouver le pc courant dans le code ZAM).

```
class CodeView(base : Connector, it : Int) extends JPanel {
```

Le gestionnaire d'onglet

```
val CodePane
}
```

ComponentListening

Classe ComponentListening : Elle s'occupe de récupérer les évènements souris permettant de changer les vues d'ordre. Chaque vue l'ajoute en observateur. Il s'occupe seulement d'indiquer le changement au manager de vue de la thread.

```
class ComponentListening(manager : ViewsManager) extends MouseListener with
MouseMotionListener {
```

Les coordonnées du premier clic, ils permettent de détecter le mouvement voulu et la direction dans laquelle l'utilisateur veut envoyer la vue.

```
var x
var y
```

Sauvegarde les coordonnées de la source

```
override def mousePressed(arg : MouseEvent)
```

Calcule et envoie une demande de modification au gestionnaire de vue

```
override def mouseReleased(arg : MouseEvent)
```

```
}
```

Connector

Classe Connector : La classe connector possède toutes les fenêtres et toutes les threads existantes. C'est la classe englobant la partie Swing.

class Connector {

Le controler sert à passer les demandes nécessaires au modèle de donnée.

Val Controler

threadFen stocke toutes les fenêtres, une thread garde son indice toutes sa vie. Etant créée simultanément que dans le simulateur elle à le même indice que dans l'environnement global.

Val threadFen

threadMin stocke pour chaque thread un booléen indiquant si elle est minimisée.

Val threadMin

Renvoie le nom de la it thread
@in : it, l'indice de la thread

def getThreadName(it : Int)

Fait avancer une thread d'un pas
@in : it, l'indice de la thread

def step(it : Int)

Fait reculer une thread d'un pas
@in : it, l'indice de la thread

def backstep(it : Int)

Fait avancer une thread jusqu'au prochain break point
@in : it, l'indice de la thread

def avance(it : Int)

Lance la préparation du simulateur

def prepare

Renvoie le nombre de thread

def getnbthread

Renvoi le chemin du fichier

```
def getPath
```

Renvoie le code d'Ocaml pointé par le chemin courant

```
def getOcmlCode
```

Renvoi le contexte de la thread it dans le simulateur
@in : it, l'indice de la thread

```
def getThread(it : Int)
```

Renvoie l'environnement global du simulateur

```
def getEnv
```

Réinitialise le programme (non terminée)

```
def restart
```

Maximise la thread it
@in : it, l'indice de la thread
@in : v, le tableau d'id correspondants aux vues de l'onglet

```
def maxThread(it : Int, v : ArrayBuffer[Intl])
```

Réduit la thread it
@in : it, l'indice de la thread

```
def reduce(it : Int)
```

Renvoie tous les tableaux présents dans les données (non terminée)
@in : it, l'indice de la thread

```
def getTables(it : Int)
```

Termine le programme

```
def Fin
}
```

ConsoleView

Classe ConsoleView : Cette classe gère l'affichage de la console. Elle n'a besoin que du lien vers le connecteur, la console étant une chaine comportant les sorties (impressions ou exception) stockés dans l'environnement.

```
class ConsoleView(base : Connector) extends JPanel {
```

Un champs de text avec son scroller

```
val textArea
val ScrollPane
}
```

ContextView

Classe ContextView : La vue du contexte affiche sous un tableau les données présent dans la pile et l'accumulateur de la thread. On utilise un model pour créer plus facilement le tableau.

```
class ContextView(base : Connector, it : Int) extends JPanel {
```

La thread concernée et le tableau affiché.

```
Val thread
var table
```

Le model permettant d'afficher le tableau plus facilement. Les méthodes garantissent la modification et d'initialisation ses données. Il prend en entrée la taille du tableau en nombre de ligne. (Le nombre de colonnes étant fixe)

```
class MyModel(size : Int) extends AbstractTableModel  {...}
}
```

DataManager

Classe DataManager : Cette classe gère les évènements correspondant à la vue DataView. Un checkbox et combobox sont présents et désignent la donnée à remplir par la vue.

```
Class DataManager(comp : DataView) extends ItemListener with ActionListener {
```

Fonction appelée lors d'un click sur la checkBox
@in : e, l'évènement

```
def itemStateChanged(e : ItemEvent)
```

Fonction appelée lors d'un changement de valeur de la comboBox
@in: e, l'évènement

```
def actionPerformed(e : ActionEvent)
}
```

DataView

Classe DataView : Elle affiche la vue des données. Il est possible de choisir n'importe quel tableau de donnée (dans un tableau compris). (Non terminée)
Elle possède un checkbox, une combobox et un tableau.

```
class DataView(base : Connector, it : Int) extends JPanel {

}
```

EnvView

Classe EnvView: Cette classe gère l'affichage de l'environnement global. Elle ressemble beaucoup à ContextView

```
Class EnvView(base : Connector) extends JPanel{
```

L'environnement et le tableau affiché.

```
Val Env
var table
```

Le model permettant d'afficher le tableau plus facilement. Les méthodes permettent de modifier et d'initialiser ses données. Il prend en entrée la taille du tableau en nombre de ligne. (Le nombre de colonnes étant fixe)

```
class MyModel(size : Int) extends AbstractTableModel  {...}
}
```

Fenetre

Classe Fenêtre: On est dans la fenêtre principale. Elle gère la présentation du tout et la différence entre la première et les autres threads. Elle peut être initialisée avec des vues et est bien sûr liée à une thread

```
Class Fenetre(v : ArrayBuffer[Int], base : Connector, it : Int) extends JFrame{
```

Le panel principal

```
Val conteneurP
```

Accesseur aux vues courantes

```
def getviews
```

Ajouter une thread minimisée au gestionnaire d'onglet
@in : it, l'indice de la thread
@in : v, les vues désirées

```
def addTab(it : Int, v : ArrayBuffer[Int])
```

Mise à jour des vues (Lors d'une maximisation)
@in : v, les vues désirées

```
def bindViews(v : ArrayBuffer[Int])
}
```

MyFileChooser

Classe MyFileChooser: Il est nécessaire pour la gestion de recherche de fichier. (Lors d'un chargement ou d'une sauvegarde de fichier)

Class MyFileChooser extends Jpanel with ActionListener {}

MyMenuListener

Classe MyMenuListener : Elle s'occupe des évènements arrivant sur le menu déroulant. La différentiation des options se fait avec le nom du menu. (Non terminée)

Class MyMenuListener(name : String, base : Connector) extends ActionListener }

Fonction appelée lors d'un événement sur le menu

def actionPerformed(e : ActionEvent)

}

MyViewPanel

Classe MyViewPanel : Cette classe s'occupe d'afficher les vues dans l'ordre du gestionnaire de vue.

Class MyViewPanel(base : Connector, it : Int, choice : ViewsManager) extends JPanel{

La fonction permettant d'afficher les vues grâce au gestionnaire de vue

def afficher

Réaffiche toutes les vues

def modif

Retourne dans l'ordre les vues courantes

def getviews

}

OcamlView

Classe OcamlView : Elle affiche le code Ocaml

Class OcamlView(base : Connector, it : Int) extends Jpanel {

}

ViewsManager

Classe ViewsManager : C'est la classe qui s'occupe et gère l'ordre des vues. Elle est directement reliée aux boutons du menu.

Class ViewsManager(Views : ArrayBuffer[Int]) extends ItemListener {

Ajoute une nouvelle vue en fin de tableau
@in v, l'id de la nouvelle vue
def add(v : Int)

Supprime la vue v
@in v, l'id de la nouvelle vue
def sub(v : Int)

Renvoie vrai si la vue est déjà présente
@in v, l'id de la vue
def contains(v : Int)

Renvoie le nombre de vues
def size

Renvoie la vue à l'indice i
def get(i : Int)

Fait le lien entre la checkbox et la vue *bind* de la thread. Bind étant incrémenté à chaque fois, la première checkbox correspond donc à la vue 0, la deuxième à la vue 1 …
@in cbMenuItem, la checkbox à lier
def bindBox(cbMenuItem : JcheckBoxMenuItem)

Lie le gestionnaire de vue à son panel
@in p, le panel (MyViewPanel)
def bindPan(p : MyViewPanel)

Fonction appelée lors d'un click sur une checkbox
@in e, l'évènement
def itemStateChanged(e : IntemEvent)

Retourne les Views Courante
def getviews

Swap deux vues

@in : str, le nom de la vue à changer (la source)

@in : change, le changement d'it à effectuer dans le tableau (la cible)

```
def change(str : String, change : Int)
```

}

WMainEvents

Classe WmainEvents : Cette classe s'occupe de récupérer les évènements de fenêtre. Elle contient la fenêtre swing et passe quelques ordres reçus par le connecteur.

```
Class WMainEvents(v : ArrayBuffer[Int], base : Connector, it : Int) extends WindowsListener {
```

Sa fenêtre

```
val MyWindows
```

Fonction appelée lors de la fermeture de fenêtre

```
override def windowClosing(event : WindowEvent)
```

'Ferme' la fenêtre lors d'une minimisation

```
def  close
```

Renvoie les vues de cette fenêtre

```
def  getviews
```

Ajoute une fenêtre minimisée à l'onglet (thread main)

@in : it, l'indice de la thread ajoutée

@in : v, les vues désirées

```
def  addTab( it : Int, v : ArrayBuffer[Int])
```

Maximiser la fenêtre

@in : Les vues désirées

```
def max(v : ArrayBuffer[Int])
```

}

ZamView

Classe ZamView : C'est la classe qui affiche le code zam ainsi que les breaks points. Ces breaks Points sont gérés comme les tableaux précédents (avec un model), celui ci par contre est observateur et gère aussi le changement de ses champs. (Non terminée)

```
Class ZamView(base : Connector, it : Int) extends JPanel{

    Sa fenetre
    val MyWindows

    Fonction appelée lors de la fermeture de fenêtre
    override def windowClosing(event : WindowEvent)

    'Ferme' la fenêtre lors d'une minimisation
    def  close

    Renvoie les vues de cette fenêtre
    def  getviews

    Ajoute une fenêtre minimisée à l'onglet (thread main)
    @in : it, l'it de la thread ajoutée
    @in : v, les vues désirées
    def  addTab( it : Int, v : ArrayBuffer[Int])

    Maximiser la fenêtre
    @in : Les vues désirées
    def max(v : ArrayBuffer[Int])
}
```

Package Control

Controler

La classe contrôler est simplement une délégation et un relayeur qui transmet les ordres des vues au modèles de donnée.

```
class Controler {
```

Il possède donc la porte d'entrée du modèle de donnée, à savoir la classe simulateur

```
val Simulateur
```

Retourne le chemin courant

```
def getPath
```

Retourne le code Ocaml

```
def getOcmlCode
```

Retourne la i° thread de l'environnement global
@in: it, l'indice de la thread

```
def getThread(it : Int
```

Retourne l'environnement global

```
def getEnv
```

Prépare le Simulateur avec le path actuel

```
def prepare s
```

Retourne le pointeur de code de la thread it
@in : it, l'indice de la thread

```
def getpc(it : Int)
```

Retourne la taille de l'AST

```
def getASTSize
```

Avance la thread it de n pas
@in : it, l'indice de la thread
@in :n, le nombre d'instruction à exécuter

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

Recule la thread it de n pas

@in : it, l'indice de la thread

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

```
def Reculer(it : Int, n : Int)
```

Redémarre le simulateur

```
def restart
}
```

Package Test

Les tests sont implémentés sous la forme de JUnit, ils ont pour but de valider et d'assurer la qualité du logiciel. (Non terminée)

EvaluateurTest

La classe EvaluateurTest est chargée de vérifier la bonne exécution de chaque instruction. Chaque instruction se voit donc attribuer une méthode test par test la concernant.

```
Class EvaluateurTest {
```

Selon le déroulement de JUnit, cette méthode est appelée avant chaque test

```
@before
def beforeTests
```

Selon le déroulement de JUnit, cette méthode est appelée après chaque test

```
@After
def afterTests
```

Les méthodes test sont annotées de @test. Ici elle test la validité de l'instruction Acc.

```
@Test
def testexecAcc

}
```

SimulateurTest

La classe SimulateurTest est chargée de vérifier la bonne exécution du simulateur. Elle doit vérifier chacune de ses méthodes, et s'organise comme la classe de test précédente.

```
Class SimulateurTest {
```

Selon le déroulement de JUnit, cette méthode est appelée avant chaque test

```
@before
def beforeTests
```

Selon le déroulement de JUnit, cette méthode est appelée après chaque test

```
@After
def afterTests
```

Les méthodes test sont annotées de @test. Ici elle test la validité de l'instruction Avancer.

```
@Test
def testAvancer

}
```

Exemple :

Le parseur n'étant pas implémenté, quelques exemples ont été créé afin de faire tourner le simulateur seulement 'physiquement'. Ces exemples servent à montrer l'utilisation et la validité du simulateur et de ses instructions.

Amélioration :

Un dernier exemple devrait montrer à la fois la partie physique et vue.

Voyons un peu les exemples et ses formulations :

Ces exemples sont à placer dans le main après avoir instancié un simulateur de la manière suivante :

```
def main(args: Array[String]) { ...

        val Simulateur = new ZAM.Simulator
```

Premier test:

Ce premier test construit un code mettant l'entier 2 puis 4 dans la pile pour ensuite mettre un entier 50 en variable globale. Il est exécuté par une thread qui avance et recule dans son exécution.

```
/**
 * Test implantation
 * 1) Un thread et instruction simple
 */

Simulateur.AST.add(new ZAM.Const(2))
Simulateur.AST.add(new ZAM.Push)
Simulateur.AST.add(new ZAM.Const(4))
Simulateur.AST.add(new ZAM.Push)
Simulateur.AST.add(new ZAM.Constint(50))
Simulateur.AST.add(new ZAM.Setglobal(3))

Simulateur.Preparer
//print
print(Simulateur.toString)

Simulateur.Env.pushthread(new ZAM.ThreadState)
//execution
Simulateur.Avancer(0, 5)

println(Simulateur.printthread(0))

Simulateur.Revenir(0, 2)

 //print
println(Simulateur.printthread(0))
println(Simulateur.printenv())

Simulateur.Avancer(0, 3)

  //print
println(Simulateur.printthread(0))
println(Simulateur.printenv)
```

Deuxième test :

Dans ce deuxième test les entiers 10 et 55 sont aussi ajoutés dans l'environnement global. L'entier 55 étant la même variable globale (elle écrase donc le 50). Ce code est ensuite exécuté par deux threads avançant ou reculant indépendamment.

```
/**
  * Test implantation
  * 2) Deux threads et instruction simple
  */
Simulateur.AST.add(new ZAM.Const(2))
Simulateur.AST.add(new ZAM.Push)
Simulateur.AST.add(new ZAM.Const(4))
Simulateur.AST.add(new ZAM.Push)
Simulateur.AST.add(new ZAM.Constint(50))
Simulateur.AST.add(new ZAM.Setglobal(3))
Simulateur.AST.add(new ZAM.Constint(10))
Simulateur.AST.add(new ZAM.Setglobal(2))
Simulateur.AST.add(new ZAM.Constint(55))
Simulateur.AST.add(new ZAM.Setglobal(3))

Simulateur.Preparer
//print
print(Simulateur.toString)

//deux threads
Simulateur.Env.pushthread(new ZAM.ThreadState)
Simulateur.Env.pushthread(new ZAM.ThreadState)

//execution
Simulateur.Avancer(0, 8)
println("**********¥n")
println(Simulateur.printthread(0))
println(Simulateur.printenv)

 //execution
Simulateur.Avancer(1, 10)
println("**********¥n")
println(Simulateur.printthread(1))
println(Simulateur.printenv)

Simulateur.Revenir(1, 4)

 //print
println("**********¥n")
println(Simulateur.printthread(0))
println(Simulateur.printthread(1))
println(Simulateur.printenv)
```

```
Simulateur.Avancer(0, 2)

  //print
println("**********¥n")
println(Simulateur.printthread(0))
println(Simulateur.printenv)
```

Troisième test :

Ce troisième test utilise les blocs, et ses méthodes de gestions. Ici on crée un string et l'on change la variable globale stockant cette chaine par sa 6° lettre (le W). Trois threads exécutent ce code.

```
/**
  * Test implantation
  * 3) Trois threads et blocs (string)
  */
Simulateur.AST.add(new ZAM.Const('d'))
Simulateur.AST.add(new ZAM.Pushconst('l'))
Simulateur.AST.add(new ZAM.Pushconst('r'))
Simulateur.AST.add(new ZAM.Pushconst('o'))
Simulateur.AST.add(new ZAM.Pushconst('W'))
Simulateur.AST.add(new ZAM.Pushconst('o'))
Simulateur.AST.add(new ZAM.Pushconst('l'))
Simulateur.AST.add(new ZAM.Pushconst('l'))
Simulateur.AST.add(new ZAM.Pushconst('e'))

Simulateur.AST.add(new ZAM.Pushconst('h'))

Simulateur.AST.add(new ZAM.Makeblock(10, ZAM.BlockT.string_t.id))

Simulateur.AST.add(new ZAM.Setglobal(3))

Simulateur.AST.add(new ZAM.Getglobalfield(3,5))
Simulateur.AST.add(new ZAM.Setglobal(3))


Simulateur.Preparer
//print
print(Simulateur.toString)

//trois threads
Simulateur.Env.pushthread(new ZAM.ThreadState)
Simulateur.Env.pushthread(new ZAM.ThreadState)
Simulateur.Env.pushthread(new ZAM.ThreadState)
```

```
//execution
Simulateur.Avancer(0, 11)
println("**********¥n")
println(Simulateur.printthread(0))
println(Simulateur.printenv)

 //execution
Simulateur.Avancer(1, 12)
println("**********¥n")
println(Simulateur.printthread(1))
println(Simulateur.printenv)

Simulateur.Revenir(1, 4)

 //print
println("**********¥n")
println(Simulateur.printthread(0))
println(Simulateur.printthread(1))
println(Simulateur.printenv)


Simulateur.Avancer(0, 2)
Simulateur.Avancer(2, 14)

  //print
println("**********¥n")
println(Simulateur.printthread(0))
 println(Simulateur.printthread(2))
println(Simulateur.printenv)
```

Quatrième test :

Dans ce test on crée un tableau de double. Un double étant lui même un bloc cela test une grande partie des instructions concernant la gestion de bloc. Ce code exécuté par un seul thread crée 5 blocs de double comportant chacun deux entiers, puis un tableau de double avec ces 5 blocs

```
/**
        * Test implantation
        * 4) thread et blocs (Double Array)
        */

       Simulateur.AST.add(new ZAM.Const(156))
       Simulateur.AST.add(new ZAM.Pushconst(123))

       Simulateur.AST.add(new ZAM.Makeblock(2, ZAM.BlockT.double_t.id))
       Simulateur.AST.add(new ZAM.Setglobal(0))

       Simulateur.AST.add(new ZAM.Const(256))
       Simulateur.AST.add(new ZAM.Pushconst(223))
       Simulateur.AST.add(new ZAM.Makeblock(2, ZAM.BlockT.double_t.id))

       Simulateur.AST.add(new ZAM.Setglobal(1))
       Simulateur.AST.add(new ZAM.Const(356))
       Simulateur.AST.add(new ZAM.Pushconst(323))
       Simulateur.AST.add(new ZAM.Makeblock(2, ZAM.BlockT.double_t.id))

       Simulateur.AST.add(new ZAM.Setglobal(2))
       Simulateur.AST.add(new ZAM.Const(456))
       Simulateur.AST.add(new ZAM.Pushconst(423))
       Simulateur.AST.add(new ZAM.Makeblock(2, ZAM.BlockT.double_t.id))

       Simulateur.AST.add(new ZAM.Setglobal(3))
       Simulateur.AST.add(new ZAM.Const(556))
       Simulateur.AST.add(new ZAM.Pushconst(523))
       Simulateur.AST.add(new ZAM.Makeblock(2, ZAM.BlockT.double_t.id))


       Simulateur.AST.add(new ZAM.Pushgetglobal(3))
       Simulateur.AST.add(new ZAM.Pushgetglobal(2))
       Simulateur.AST.add(new ZAM.Pushgetglobal(1))
       Simulateur.AST.add(new ZAM.Pushgetglobal(0))

       Simulateur.AST.add(new ZAM.Makefloatblock(5))

       Simulateur.AST.add(new ZAM.Setglobal(4))

       Simulateur.AST.add(new ZAM.Getglobal(4))

       Simulateur.AST.add(new ZAM.Getfloatfield(2))
```

```
Simulateur.AST.add(new ZAM.Pushgetglobal(4))

Simulateur.AST.add(new ZAM.Setfloatfield(1))

Simulateur.Preparer
//print
print(Simulateur.toString)

//un thread
Simulateur.Env.pushthread(new ZAM.ThreadState)

//execution
Simulateur.Avancer(0, 2)
println("**********¥n")
println(Simulateur.printthread(0))
println(Simulateur.printenv)

 Simulateur.Avancer(0, 2)
println("**********¥n")
println(Simulateur.printthread(0))
println(Simulateur.printenv)

 Simulateur.Avancer(0, 2)
println("**********¥n")
println(Simulateur.printthread(0))
println(Simulateur.printenv)

 Simulateur.Avancer(0, 23)
println("**********¥n")
println(Simulateur.printthread(0))
println(Simulateur.printenv)
```

Cinquème test :

Dans ce code on test les opérations d'entiers, booléens et les branchements. Chaque thread test ses opérations et se retrouve à l'instruction switch. Les instructions de vecteurs sont ensuite testées.

```
/**
   * Test implantation
   * 5) thread, arithmetic et sauts
   */

 Simulateur.AST.add(new ZAM.Const(4))
 Simulateur.AST.add(new ZAM.Setglobal(0))

 Simulateur.AST.add(new ZAM.Const(2))
 Simulateur.AST.add(new ZAM.Pushgetglobal(0))

 Simulateur.AST.add(new ZAM.Integer_comparision("EQ"))
 Simulateur.AST.add(new ZAM.Branchif(26)) //if varglob(0) == 2 -> pc +20

 // {
 Simulateur.AST.add(new ZAM.Const(1))
 Simulateur.AST.add(new ZAM.Pushgetglobal(0))
 Simulateur.AST.add(new ZAM.Subint) // var --
 Simulateur.AST.add(new ZAM.Setglobal(0))

 Simulateur.AST.add(new ZAM.Getglobal(0))
 Simulateur.AST.add(new ZAM.Pushconst(3))
 Simulateur.AST.add(new ZAM.Integer_comparision("LEINT"))
 Simulateur.AST.add(new ZAM.Branchifnot(9)) //if! ( 3 <=  varglob(0)) -> pc + 20

 // {
 Simulateur.AST.add(new ZAM.Const(2))
 Simulateur.AST.add(new ZAM.Pushconst(2))
 Simulateur.AST.add(new ZAM.Addint) //2+2

 Simulateur.AST.add(new ZAM.Pushconst(6))
 Simulateur.AST.add(new ZAM.Pushconst(6))
 Simulateur.AST.add(new ZAM.Mulint) //6*6

 Simulateur.AST.add(new ZAM.Pushconst(2))
 Simulateur.AST.add(new ZAM.Branch(22))
 //}
 //{ ifnot
 Simulateur.AST.add(new ZAM.Const(1))
 Simulateur.AST.add(new ZAM.Negint) //~1

 Simulateur.AST.add(new ZAM.Pushconst(3))
 Simulateur.AST.add(new ZAM.Pushconst(9))
 Simulateur.AST.add(new ZAM.Divint) // 9/3

 Simulateur.AST.add(new ZAM.Pushconst(2))
```

```
 Simulateur.AST.add(new ZAM.Pushconst(10))
 Simulateur.AST.add(new ZAM.Modint) // 10%2

 Simulateur.AST.add(new ZAM.Integer_branch_comparision(0, 13, "BEQ"))
 //}}
 // if {
Simulateur.AST.add(new ZAM.Const(2))
Simulateur.AST.add(new ZAM.Pushconst(6))
Simulateur.AST.add(new ZAM.Andint) // 6 & 2

Simulateur.AST.add(new ZAM.Pushconst(2))
Simulateur.AST.add(new ZAM.Pushconst(4))
Simulateur.AST.add(new ZAM.Orint) // 4 | 2

Simulateur.AST.add(new ZAM.Pushconst(2))
Simulateur.AST.add(new ZAM.Pushconst(6))
Simulateur.AST.add(new ZAM.Xorint) // 6 ^ 2

Simulateur.AST.add(new ZAM.Integer_comparision("GTINT")) // 4 > 6
Simulateur.AST.add(new ZAM.Boolnot) //true
Simulateur.AST.add(new ZAM.Branchifnot(205)) //not jump
//}

Simulateur.AST.add(new ZAM.Switch(0, Array(1, 6, 11)))

Simulateur.AST.add(new ZAM.Const(2))
Simulateur.AST.add(new ZAM.Pushconst(2))
Simulateur.AST.add(new ZAM.Lslint) // 2 << 2
Simulateur.AST.add(new ZAM.Makeblock(3, ZAM.BlockT.double_t.id))
Simulateur.AST.add(new ZAM.Setglobal(0))

Simulateur.AST.add(new ZAM.Const(2))
Simulateur.AST.add(new ZAM.Pushconst(2))
Simulateur.AST.add(new ZAM.Lsrint) // 2 >> 2
Simulateur.AST.add(new ZAM.Getglobal(0))
Simulateur.AST.add(new ZAM.Vectlength)

Simulateur.AST.add(new ZAM.Const(2))
Simulateur.AST.add(new ZAM.Pushconst(2))
Simulateur.AST.add(new ZAM.Asrint) // 2 >> 2
Simulateur.AST.add(new ZAM.Const(2))
Simulateur.AST.add(new ZAM.Push)
Simulateur.AST.add(new ZAM.Getglobal(0))
Simulateur.AST.add(new ZAM.Setvectitem)

 Simulateur.Preparer
 //print
 print(Simulateur.toString)

 //un thread
 Simulateur.Env.pushthread(new ZAM.ThreadState)
 Simulateur.Env.pushthread(new ZAM.ThreadState)
 Simulateur.Env.pushthread(new ZAM.ThreadState)

 Simulateur.Avancer(0, 2)
 Simulateur.Avancer(1, 2)
```

```
Simulateur.Avancer(2, 2)

Simulateur.Avancer(0, 20)
println("**********¥n")
println(Simulateur.printthread(0))
println(Simulateur.printenv)

Simulateur.Avancer(1, 21)
println("**********¥n")
println(Simulateur.printthread(1))
println(Simulateur.printenv)

Simulateur.Avancer(2, 16)
println("**********¥n")
println(Simulateur.printthread(2))
println(Simulateur.printenv)

println("**********¥n")
println("**  apres switch  **¥n")

Simulateur.Avancer(1, 6)
println("**********¥n")
println(Simulateur.printthread(1))
println(Simulateur.printenv)

Simulateur.Avancer(2, 6)
println("**********¥n")
println(Simulateur.printthread(2))
println(Simulateur.printenv)

Simulateur.Avancer(0, 8)
println("**********¥n")
println(Simulateur.printthread(0))
println(Simulateur.printenv)
```

Journal de Bord

Sur des périodes de 15 jours, plusieurs objectifs ont été fixés et réalisés. Le journal de bord recense cette chronologie.

- Explication du sujet, compréhension et analyse (Exemple d'une Vm ZAM), proposition graphique (pencil)

- Début d'implantation de la partie Graphique & apprentissage du langage SCALA.

- Analyse des sources de Caml (Autre exemple de Vm ZAM) & idée d'implantation de la ZAM dans le programme (Instructions et représentation valeurs).

- Implantation du contexte et concrétisation de l'ensemble par des exemples.
  (1 Thread, pile et état global : Entier)

- Implantation du contexte et concrétisation de l'ensemble par des exemples. (2 Threads, pile et état global : Entier), refactoring et ajustement du code.

- Mise à jour des documents et création de l'explication des classes. Implantation du contexte et concrétisation de l'ensemble par des exemples. (3 Threads, pile, état global et blocs)

- Implantation du contexte et concrétisation de l'ensemble par des exemples.(arithmétique et comparaison)

- Implantation du reste des instructions de la ZAM (principalement les instructions concernant les fonctions), complétion de la partie vue et formation du model MVC.

- Création du rapport.

Conclusion :

Malheureusement non terminé ce projet m'a apporté une grande connaissance des machines virtuelles (principalement de la ZAM) et du langage Scala qui est très pratique et agréable. Bien que initialement prévu pour deux j'ai quand même pu avancer à la fois dans l'implémentation physique de la ZAM et visuelle, m'apportant aussi un approfondissement de mes connaissances dans Swing.

Je me suis aussi familiarisé à l'OCaml et aux difficultés liées à la représentation et aux conflits de la programmation concurrente.