CSC4251_4252 : Génération de code

Pascal Hennequin, J. Paul Gibson, Denis Conan

Novembre 2024

Sommaire

Comme on s'y attend, ce cours est plus pratique
(MIPS et projet MiniJAVA).

1 Différentes fonctions

1.1 Allocation de la Mémoire et des Registres

- Définir les « adresses » pour implanter les variables (ou les méthodes avec liaison dynamique)
- Identifier la vivacité des variables :
  - Utilisation des registres ?
  - Localité et besoin de save/restore pour les appels de fonctions ?
  - ...
- Optimisation des registres
  - Nombre d'Ershov
  - Algorithmes de coloriage de graphe

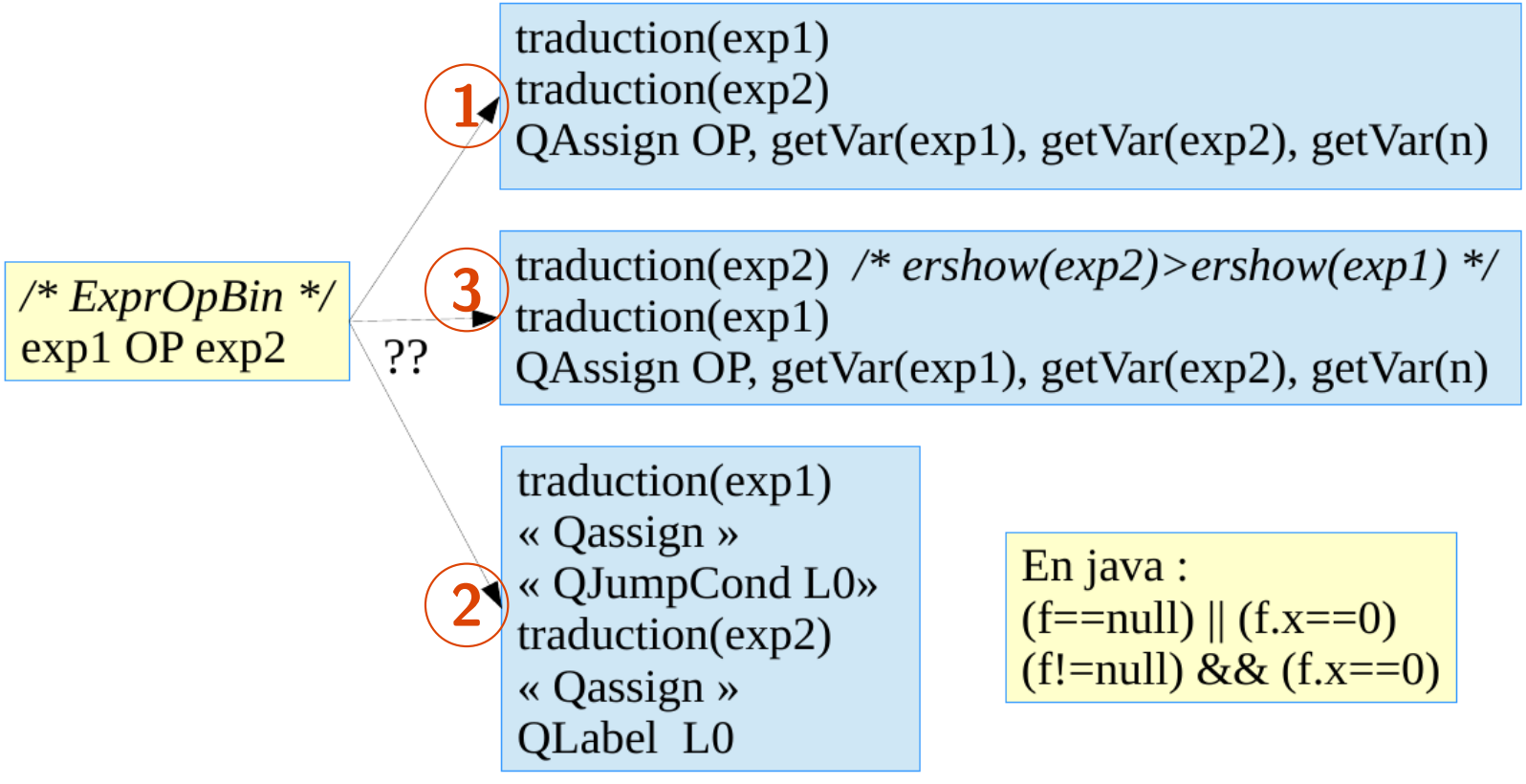
1.1.1 Nombre d'Ershov

- Nombre d'Ershov = nombre qui, associé à un nœud d'un arbre d'expression, indique le nombre de registres nécessaires pour évaluer ce nœud sans avoir besoin de variable temporaire en mémoire, mais uniquement des registres
- Algorithme de calcul du nombre d'Ershov
  1. Étiqueter toutes les feuilles avec 1
  2. Un nœud interne ayant un seul fils à le même nombre d'Ershov que ce fils
  3. Si un nœud interne possède deux fils, son nombre d'Ershov est :
     - 3.1 nombres d'Ershov des enfants différents $\Longrightarrow$ prendre max
     - 3.2 nombres d'Ershov identiques $\Longrightarrow$ prendre ce nombre $+1$
- Traduction en utilisant le nombre d'Ershov
  1. Si nbre de registres disponibles $\geq$ nbre d'Ershov à la racine de l'expr.
     - 1.1 Évaluer d'abord l'enfant avec le plus gros nombre d'Ershov
     - 1.2 Stocker le résultat dans un registre
     - 1.3 Évaluer l'enfant avec le plus petit nombre d'Ershov

. Rappel : le projet MiniJAVA cible l'assembleur MIPS, qui est un assembleur pour architecture RISC (*Reduced Instruction Set Computer*), dont la philosophie est d'avoir beaucoup de registres.

1.1.2 Traduction des expressions

1. Représentation Intermédiaire linéaire « classique », sans optimisation
2. Meilleure Représentation Intermédiaire pour les opérateurs logiques
3. Calcul du nombre d'Ershov pour minimiser le nombre de registres



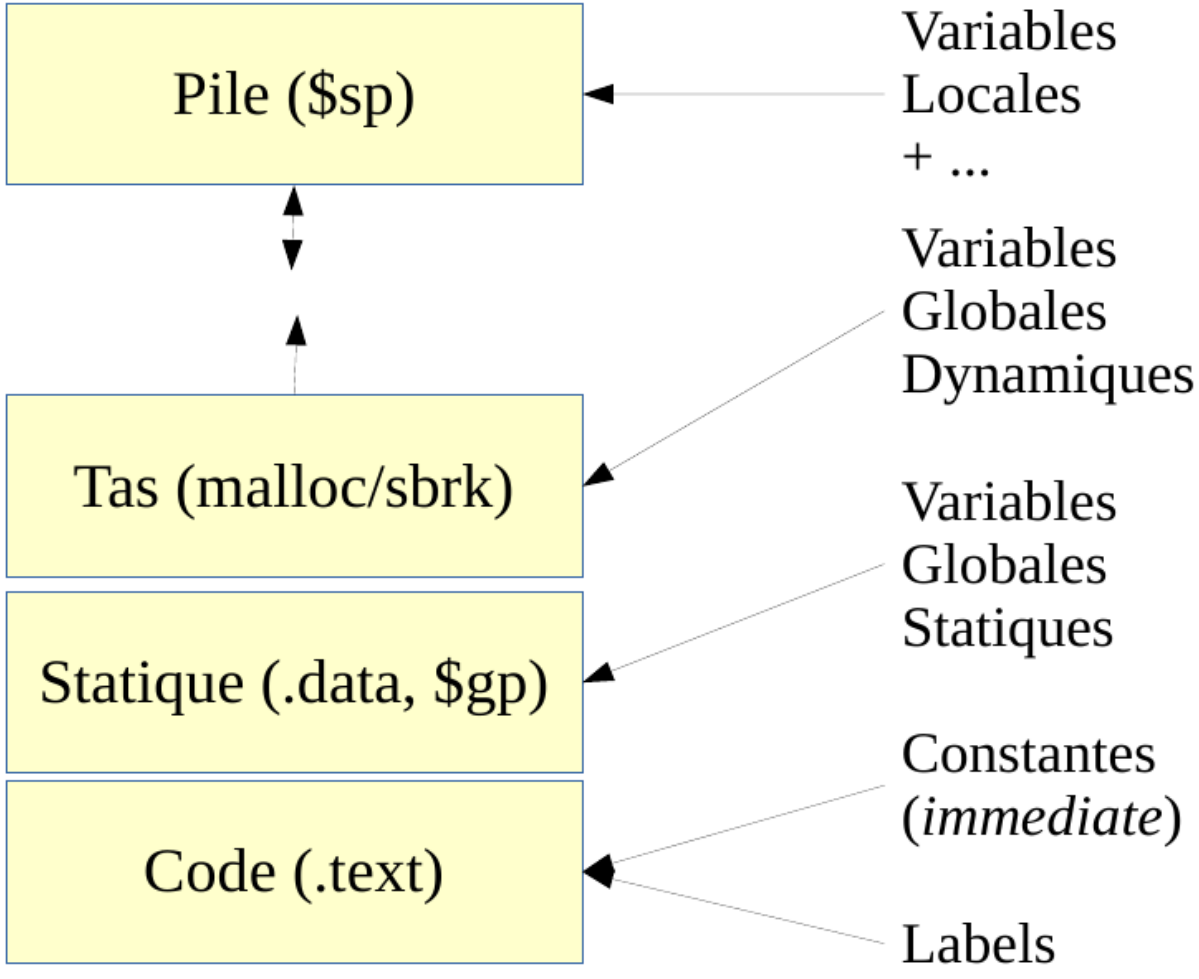
1.2 Traduction de la Représentation Intermédiaire vers le langage machine

- Réalisation d'un schéma d'appel pour les fonctions :
  - Passage d'argument, valeur de retour, adresse de retour
  - Sauvegarde de registres
  - Convention d'appel entre appelante et appelée
  - Utilisation de la pile et/ou des registres
- Sélection des instructions assembleur
- Génération du code assembleur,
  Assemblage du code assembleur pour obtenir du code machine
  Édition des liens pour prendre en compte un environnement d'exécution [1]

1. Le *runtime* MiniJAVA contient par exemple une réalisation en assembleur de la classe `Object`, de l'opérateur `new` (uniquement l'allocation mémoire), ou encore des méthodes `System.out.println` et `System.exit`.

2 Mémoire et Variables

2.1 Principes



2.2 Jalon 5 : source et RI

```
class Test105 {
  public static void main(String[] args) {
    System.out.println(new Test3().Start(6, 7));
  }
}

class Test2 {
  int a;

  public int Start(int i, int j) {
    int k;
    return i * j;
  }

  public int un() { return 1; }

  int b;
}


class Test3 extends Test2 {
  public int zero() {
    return 0;
  }
}
```

```
=== new IntermediateRepresentation
"main"                          | QLabel[null, main, null, n
t_0 = new Test3                 | QNew[null, Test3, null, t_
param t_0                       | QParam[null, t_0, null, nu
param 6                         | QParam[null, c_6, null, nu
param 7                         | QParam[null, c_7, null, nu
t_1 = call Start<3>             | QCall[null, Start, c_3, t_
param t_1                       | QParam[null, t_1, null, nu
call _println<1>                | QCallStatic[null, _println
param 0                         | QParam[null, c_0, null, nu
call _exit<1>                   | QCallStatic[null, _exit, c
"Start"<3>                      | QLabelMeth[null, Start, c_
t_2 = i * j                     | QAssign[*, int i, int j, t
return t_2                      | QReturn[null, t_2, null, n
"un"<1>                         | QLabelMeth[null, un, c_1,
return 1                        | QReturn[null, c_1, null, n
"zero"<1>                       | QLabelMeth[null, zero, c_1
return 0                        | QReturn[null, c_0, null, n
```

2.3 Classe phase.e_codegen.Allocator avec ex. Jalon 5

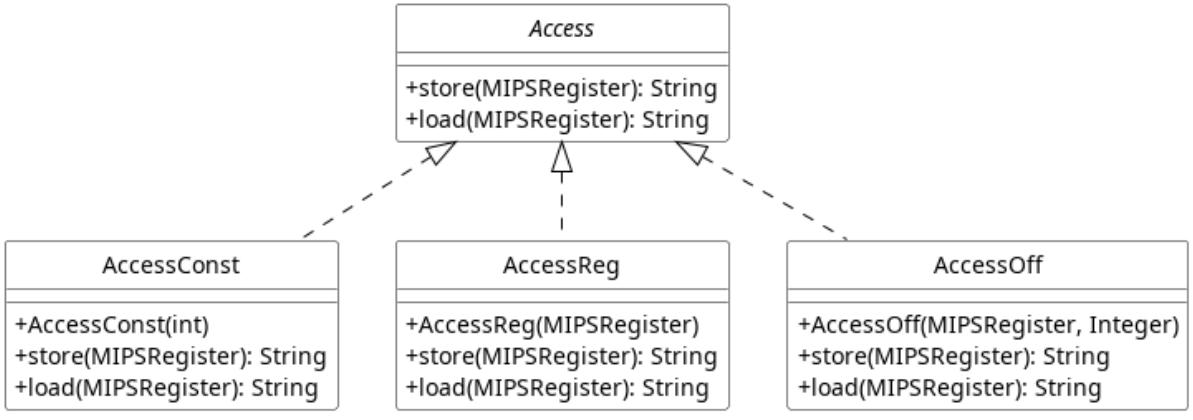
```
1  class Test105 {
2    public static void main(String[] args) {
3      System.out.println(new Test3().Start(6, 7));
4    }
5  }
6
7  class Test2 {
8    int a;
9
10   public int Start(int i, int j) {
11     int k;
12     return i * j;
13   }
14
15   public int un() { return 1; }
16
17   int b;
18 }
19
20
21 class Test3 extends Test2 {
22   public int zero() {
23     return 0;
24   }
25 }
```

```
globalSize (main) 8
classSize {Object=0, Test105=0,
     Test3=8, Test2=8}
frameSizeMin 36
frameSize {zero=36, Start=52,
     equals=40, un=36}
```

- `globalSize` : 2 variables temporaires pour l'instanciation `new Test3()` et l'appel `Start(...)`
- Classes `Object` sans attribut, `Test105` sans attribut, `Test2` avec `a` et `b`, `Test3` avec les attributs « parents »
- Cadre d'appel minimum = `$ra` et `$s0`–`$s7`
- Cadres d'appel de `zero` et `un` sans variable locale, donc 36 octets,
  de `Start` avec paramètres `i` et `j` + variable locale `k` + variable temporaire pour `return`,
  de `equals` avec paramètre `o`

2.4 Interface `Access` et ses classes concrètes

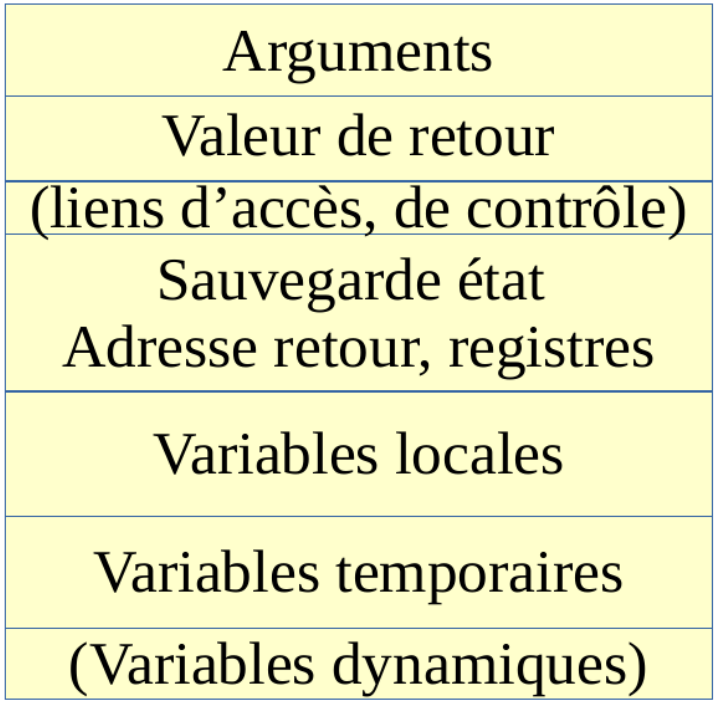
- Interface `Access` : Instancier un objet pour chaque variable de la RI
  - `store(MIPSRegister reg)` : enregistre `reg` dans la variable
  - `load(MIPSRegister reg)` : charge la variable dans `reg`
- `AccessConst` :
  - `store` : exception
  - `load` : p.ex. `li reg, 6`
- `AccessReg` :
  - Attribut `register`
  - `store(reg)` : p.ex. `move this.register, reg`
  - `load(reg)` : p.ex. `move reg, this.register`
- `AccessOff` :
  - Attributs `register` et `offset`
  - `store(reg)` : `sw reg, offset(this.register)`
  - `load(reg)` : `lw reg, offset(this.register)`



3 Cadre et convention d'appel pour MiniJAVA

3.1 Cadre d'appel

- **Cadre d'appel = Bloc d'activation = Trame = *Frame***
  - Utilisation de la pile pour les appels de fonctions
  - Des choix possibles :
    - Pile ou Registre (args, etc.)
    - Définition de l'état (quels registres ?)
  - Séquence d'appel
    - Allocation dans la pile
    - Remplissage de champs
  - Séquence de retour
    - Restauration état (pile et registres)



$\Longrightarrow$ Convention d'appel
- Définition détaillée du cadre et répartition entre appelante et appelée

3.2 Convention MIPS

- Registres et appels de fonctions
  - Valeur d'un registre préservée par l'appel : la fonction appelée est responsable de ne pas écraser ce registre ou doit faire une sauvegarde/restauration autour des appels si elle veut utiliser ce registre
  - Valeur d'un registre non préservée par l'appel : l'appelée fait ce qu'il veut de ce registre et l'appelante fait une sauvegarde/restauration autour des appels de fonction s'il veut utiliser ce registre
- Registres MIPS et appels de méthodes :
  - `$a0`–`$a3` : 4 premiers paramètres lors d'un appel de fonction et peuvent être écrasés par l'appelée
  - `$t0`–`$t9` : peuvent être écrasés par l'appelée lors d'un appel de fonction
  - `$s0`–`$s7` : ne doivent pas être écrasés par l'appelée,

3.3 Convention d'appel

(0) début de l'appel par l'appelante

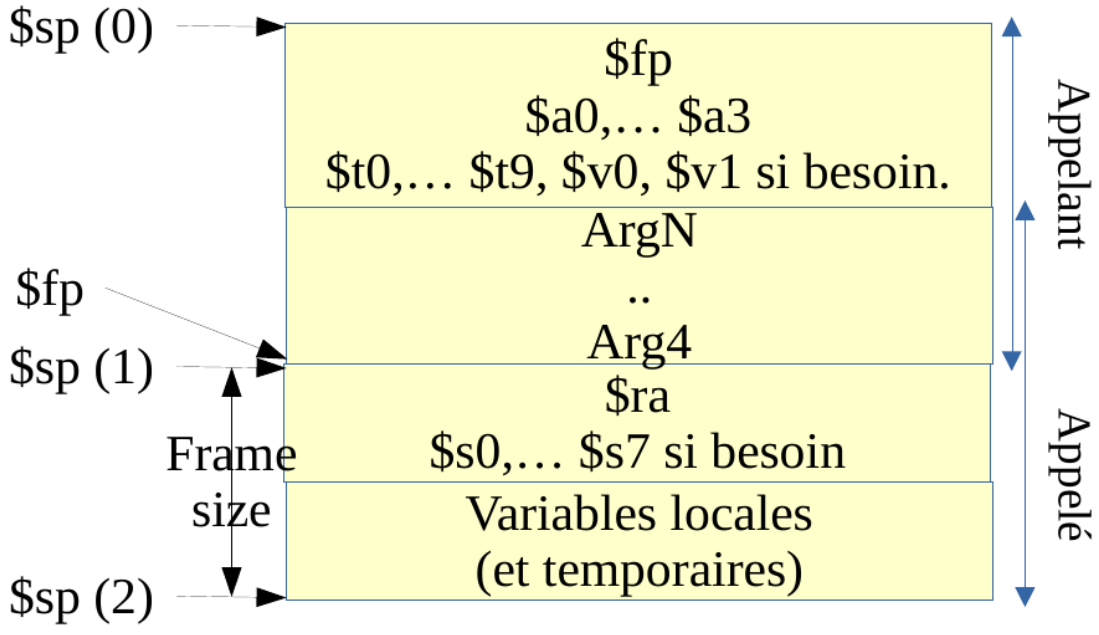
(1) fin de l'appel chez l'appelante

(2) début de l'appel chez l'appelée

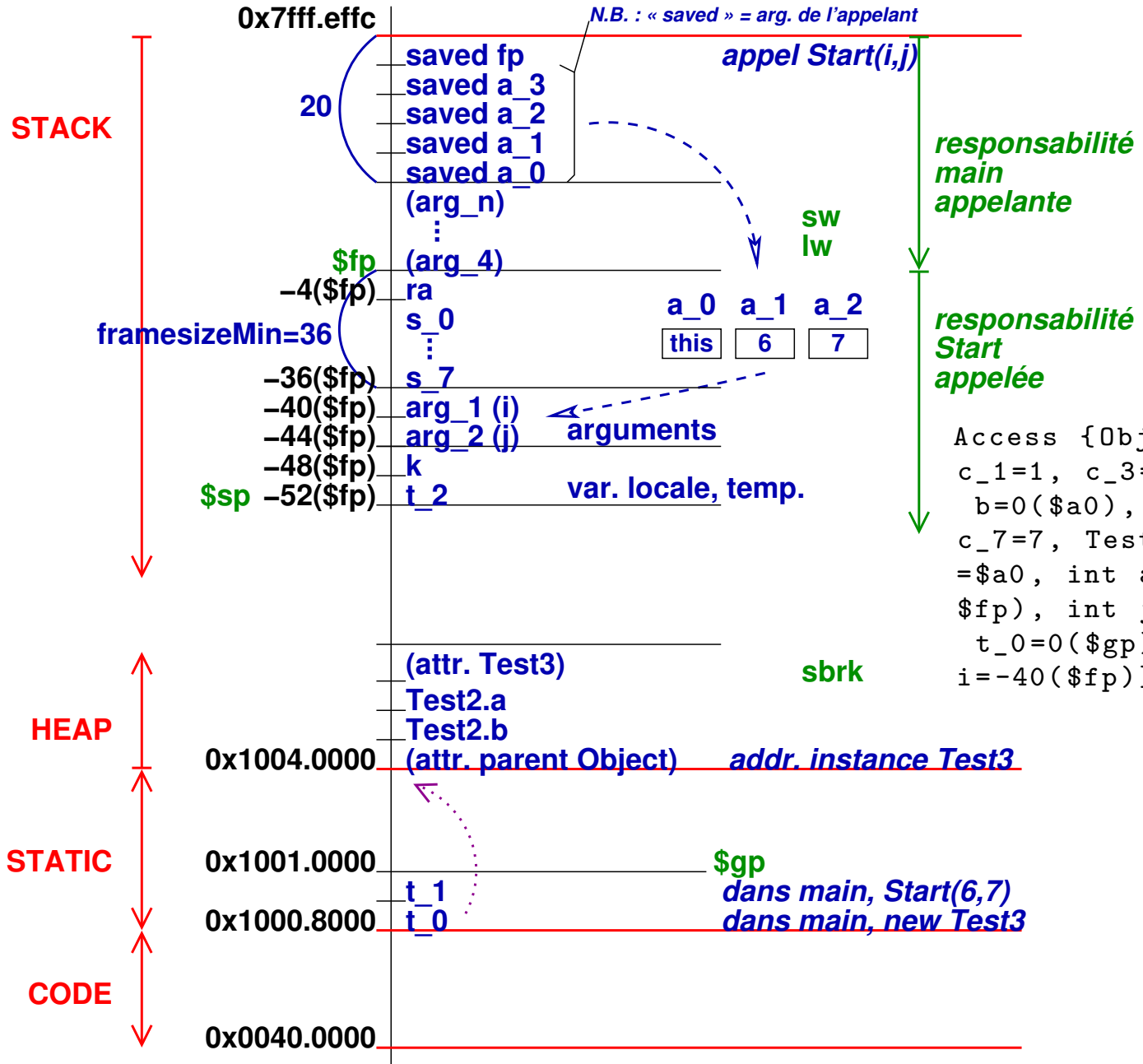
$sp = *stack pointer*
$fp = *frame pointer*

Arguments 0 à 3
$a0,... $a3

Valeur de retour
$v0



3.4 Image mémoire au début de Start(6, 7)




```
Access {Object this=$a0, c_0=0,
c_1=1, c_3=3, int k=-48($fp), int
 b=0($a0), t_2=-52($fp), c_6=6,
c_7=7, Test3 this=$a0, Test2 this
=$a0, int a=4($a0), Object o=-40(
$fp), int j=-44($fp), t_1=4($gp),
 t_0=0($gp), Test2 this=$a0, int
i=-40($fp)}
```

4 Traduction de Représentation Intermédiaire vers Code Assembleur

4.1 Sélection des instructions I

- Sélection des instructions de la Représentation Intermédiaire
- ≡ Sélection des instructions Assembleur
- Dans MiniJAVA, visiteur `ToMips` et classe utilitaire `MipsWriter`
  - Quadruplets de la RI $\rightsquigarrow$ Instructions MIPS en passant par des méthodes utilitaires

    — `Quadruplet` / `ToMips` / `MipsWriter` / `instruction MIPS`

4.1 Sélection des instructions II

N.B. : les tableaux qui suivent ne considèrent pas les boni du projet MiniJAVA et sont donnés à titre indicatif.

QParam	params.add		
QLabel	—	label	indent. + « : »
QCallStatic	push regLoad jumpIn pull	plus storeOffset Access::load storeOffset inst loadOffset plus	addi sw li, lw, move sw jal lw addi
QJump		jump	j
QJumpCond	regLoad —	... jumpIfNot	... beq
QNew	push — — regStore pull	... load jumpIn	... li ... ...

4.1 Sélection des instructions III

QCopy	regLoad regStore	... Access::store	... sw, move
QAssign	regLoad — — — — regStore	... plus moins fois et inferieur ...	... add sub mult, mflo and slt ...
QAssignUnary	regLoad — regStore	... not ...	... seq ...
QCall	callerSave regLoad — callerRestore regStore	push ... jumpIn pull ...	... ...

4.1 Sélection des instructions IV

QLabelMeth	—	label	...
	—	move	move
	—	plus	...
	calleeSave	storeOffset	...
	—	storeOffset	...
QReturn	regLoad	...	...
	—	move	...
	—	jumpOut	jr
QAssignArrayFrom	regLoad	...	...
	—	loadOffset	...
	—	inRange	sltu
	—	load	...
	push	...	...
	pull	...	...
	—	fois4	sll
	—	plus	...
	regStore	...	...

4.1 Sélection des instructions V

QAssignArrayTo	regLoad	...	...
	—	loadOffset	...
	—	inRange	sltu
	—	load	...
	push	...	...
	pull	...	...
	—	fois4	...
	—	plus	...
	regLoad	...	...
	regStore	...	...
QLength	regLoad	...	...
	—	loadOffset	...
	regStore	...	...

4.1 Sélection des instructions VI

QNewArray	push	...	...
	regLoad	...	...
	—	move	...
	—	inferieur	...
	—	not	...
	push	...	...
	pull	...	...
	—	fois4	...
	—	plus	...
	—	jumpIn	...

4.2 Traduction vers le code assembleur I

- Dans MiniJAVA, `CodeGen::execute` appelle :
  1. `Allocator::execute` : l'allocateur mémoire
  2. `ToMips::execute` : le visiteur de traduction du code assembleur
  3. `LinkRuntime::execute` : l'éditeur de lien avec l'environnement d'exécution
- Donc, dans `ToMips::execute`
  - Les objets `Access` des variables de la RI sont créés
  - Les tailles mémoire des classes sont calculées
  - Les tailles mémoire des trames des méthodes sont calculées
- Conventions d'écriture
  - `alloc` pour la classe `Allocator`
  - `mw` pour la classe `MipsWriter`
  - « variable `arg1`/.../`result` du quadruplet »
    pour « utiliser l'objet `Access` de la variable `arg1`/.../`result` du quadruplet »

4.2 Traduction vers le code assembleur II

- QJumpCond :
  - charger la variable `arg2` du quadruplet, qui contient la valeur de l'expression de la condition du saut, dans le registre `$v0`
  - utiliser `mw::jumpIfNot` pour aller à l'adresse du label, qui est dans la variable `arg1` du quadruplet
- QNew :
  - utiliser `alloc` pour obtenir la taille mémoire de la classe
  - pousser sur la pile avec `push` la valeur du registre `$a0`
  - mettre dans `$a0` avec `mw::load` la taille mémoire de la classe
  - appeler la fonction `_new_object` du *runtime* avec `mw::jumpIn`
  - récupérer avec `regStore` le résultat, qui est dans le registre `$v0`, pour le mettre dans la variable `result` du quadruplet
  - restaurer avec `pull` la valeur du registre `$a0`

4.2 Traduction vers le code assembleur III

- QCopy :
  - avec `regLoad`, mettre dans le registre `$v0` la valeur de la variable `arg1` du quadruplet
  - avec `regStore`, mettre cette valeur dans la variable `result` du quadruplet
- QAssign :
  - avec `regLoad`, mettre dans le registre `$v0` la valeur de la variable `arg1` du quadruplet
  - avec `regLoad`, mettre dans le registre `$v1` la valeur de la variable `arg2`
  - utiliser `mw::plus`, etc. selon l'opérateur
  - avec `regStore`, mettre cette valeur dans la variable `result` du quadruplet

4.2 Traduction vers le code assembleur IV

- QParam :
  - ajouter la variable `arg1` dans la liste des paramètres, qui est l'attribut `params` (le retrait intervient à la fin du traitement de `QCall` ou `QCallStatic`)
- QCall : maximum $1 + 3$ paramètres, le premier étant `this`
  - vérifier que le nombre de quadruplets `QParam` visités (attribut `params`) est égal à la valeur de la constante `arg2` du quadruplet
    - N.B. : `arg2` est une constante et on a mis le nombre de paramètres dans `arg2` lors de traduction vers la RI pour faire cette vérification ici
  - utiliser `callerSave` pour sauvegarder les registres non préservés par l'appelée
  - avec `regLoad`, mettre la valeur des paramètres dans les registres `$a1`–`$a3`
    - penser à utiliser le tableau `AREGS`
  - avec `regLoad`, mettre la valeur `this` (première entrée de `params`) dans `$a0`
  - utiliser `mw::jumpIn` pour le saut dans l'appelée
  - utiliser `callerRestore` pour restaurer les registres
  - vider la liste des paramètres `params`

4.2 Traduction vers le code assembleur V

- QLabelMeth :
  - ajouter le label contenu dans la variable `arg1` avec `mw::label`
  - positionner la trame (`$fp`) à la valeur du pointeur de pile (`$sp`) avec `mw::move`
  - allouer la trame de l'appelée en décalant `$sp` de la taille de trame calculée par `Allocator` : `plus(MIPSRegister.SP, -allocator.frameSize(nom))`
    - `nom` est contenu dans la variable `arg1` du quadruplet
  - utiliser `calleeSave` pour sauvegarder les registres `$ra` et `$s0`–`$s7`
  - avec `mw::storeOffset`, recopier comme variable locale les registres `$a1`–`$a3`
    - N.B. : recopie au cas où la méthode appelée est récursive
    - *offset* « `-allocator.frameSizeMin() - (1 * SIZEOF)` » pour `$a1`
    - *offset* « `-allocator.frameSizeMin() - (2 * SIZEOF)` » pour `$a2`
    - *offset* « `-allocator.frameSizeMin() - (3 * SIZEOF)` » pour `$a3`

4.2 Traduction vers le code assembleur VI

- QReturn :
  - utiliser `calleeRestore` pour restaurer les registres `$ra` et `$s0`–`$s7`
  - avec `regLoad`, mettre la valeur de la variable `result` dans le registre `$v0`
  - déallouer la trame de l'appelée avec `mw::move`
  - retourner dans l'appelante avec `mw::jumpOut`

4.2 Traduction vers le code assembleur VII

- QNewArray :
  - pousser sur la pile avec `push` la valeur du registre `$a0`
  - avec `regLoad`, mettre la valeur de la variable `arg2` du quadruplet dans `a0`
  - calculer la taille à allouer (un entier pour la taille) : (`$a0` + 1) × 4
  - appeler la fonction `_new_object` du *runtime* avec `mw::jumpIn`
  - avec `regLoad`, mettre à nouveau la valeur de la variable `arg2` dans `a0`
  - avec `storeOffset`, récupérer l'adresse du tableau, qui est dans `$v0`, pour la mettre dans `$a0`
  - avec `regStore`, récupérer l'adresse dans la variable `result` du quadruplet
  - récupérer depuis la pile avec `pull` le registre `$a0`

4.2 Traduction vers le code assembleur VIII

- QAssignArrayFrom :
  - avec `regLoad`, mettre la valeur de la variable `arg1`, qui est l'adresse du tableau, dans le registre `$v0`
  - avec `regLoad`, mettre la valeur de la variable `arg2`, qui est l'indice de case à lire, dans le registre `$v1`
  - calculer dans `$v0` l'adresse de la case à accéder : $\texttt{\$v1} + \Big((\texttt{\$v0} \times 4) + 4\Big)$
  - utiliser `mw::loadOffset` pour mettre dans le registre `$v0` la valeur à l'adresse calculée, qui est aussi dans `$v0`
  - avec `regStore`, mettre la valeur du registre `$v0` dans la variable `result` du quadruplet

4.2 Traduction vers le code assembleur IX

- QAssignArrayTo :
  - avec `regLoad`, mettre la valeur de la variable `arg2`, qui est l'indice de case à lire, dans le registre `$v0`
  - avec `regLoad`, mettre la valeur de la variable `result`, qui est l'adresse du tableau, dans le registre `$v1`
  - calculer dans `$v0` l'adresse de la case à accéder : $\texttt{\$v1} + \big((\texttt{\$v0} \times 4) + 4\big)$
  - avec `regLoad`, mettre la valeur de la variable `arg1`, qui est l'adresse de la valeur à affecter, dans le registre `$v0`
  - avec `mw::storeOffset`, mettre la valeur dans le registre `$v1` à l'adresse qui est dans le registre `$v1`
- QLength :
  - avec `regload`, mettre la valeur de la variable `arg1` dans le registre `$v0`
  - avec `loadOffset`, charger dans `$v0` la valeur à cette adresse
  - avec `regStore`, charger la valeur obtenue dans la variable `result`