

# Concept et outils

## < les piles >

Une pile est une structure de mémoire à accès restrictifs : on peut empiler, dépiler, savoir si la pile est vide. (ex : do, undo, redo)

### - fonction -

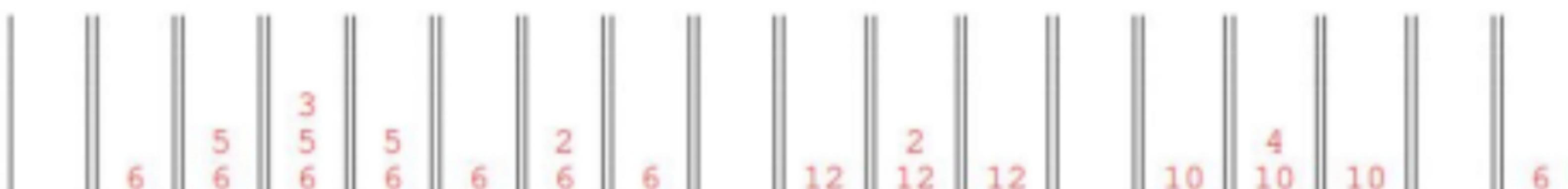
- p.empty() : vérifie si la pile est vide
- p.pop() : dépile
- p.push(x) : empile

## Analyse syntaxique

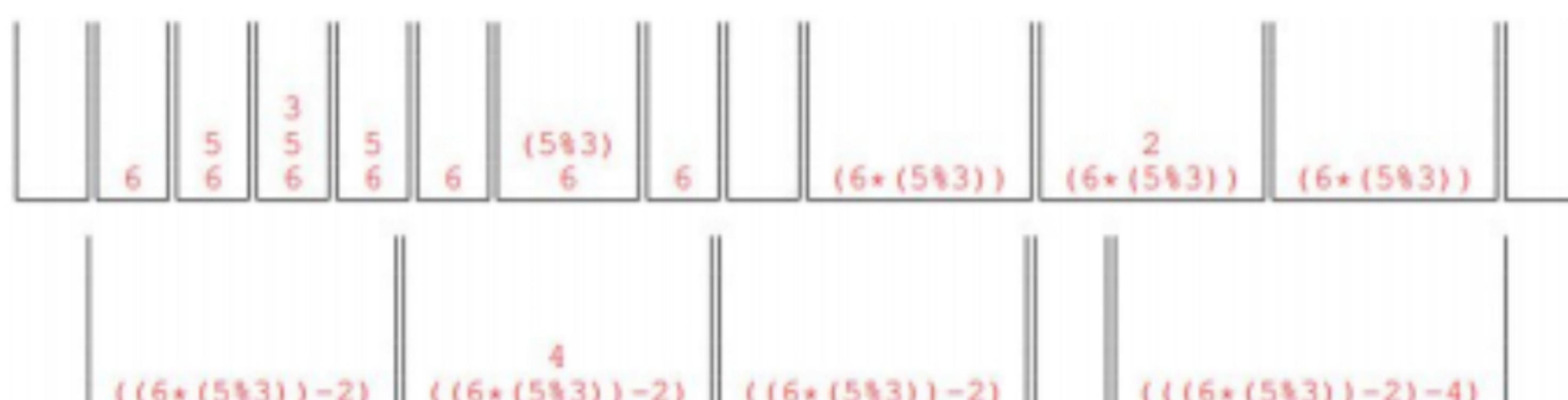
### - postfixe -

- on lit l'expression de gauche à droite
  - si le symbole est un opérande, on l'empile
  - sinon on dépile 2 éléments
    - on exécute l'opération et on empile son résultat
- le résultat est l'unique élément de la pile

ex : 6 5 3 % \* 2 - 4 -



postfixe > préfixe	postfixe > infixe
<p>on lit l'expression de gauche à droite</p> <ul style="list-style-type: none"><li>→ si c'est un opérande, on l'empile</li><li>→ sinon on dépile 2 éléments, forme le préfixe qu'on empile</li></ul> <p>le résultat est l'unique élément de la pile</p>	<p>on lit l'expression de gauche à droite</p> <ul style="list-style-type: none"><li>→ si c'est un opérande, on l'empile</li><li>→ sinon on dépile 2 éléments et on construit l'infixe avec ( )</li></ul> <p>le résultat est l'unique élément de la pile</p>



### - préfixe -

→ même processus que l'évaluation postfixe mais de droite à gauche

ex : -- \* 6 % 5 3 2 4

préfixe > postfixe	préfixe > infixé
<p>on lit l'expression de droite à gauche</p> <ul style="list-style-type: none"><li>→ si c'est un opérande, on empile</li><li>→ sinon on dépile 2 éléments, forme la forme postfixe qu'on empile</li></ul> <p>Le résultat est l'unique élément de la pile</p>	<p>on lit l'expression de droite à gauche</p> <ul style="list-style-type: none"><li>→ si c'est un opérande, on empile</li><li>→ sinon on dépile 2 éléments, forme la forme infixé qu'on empile</li></ul> <p>Le résultat est l'unique élément de la pile</p>

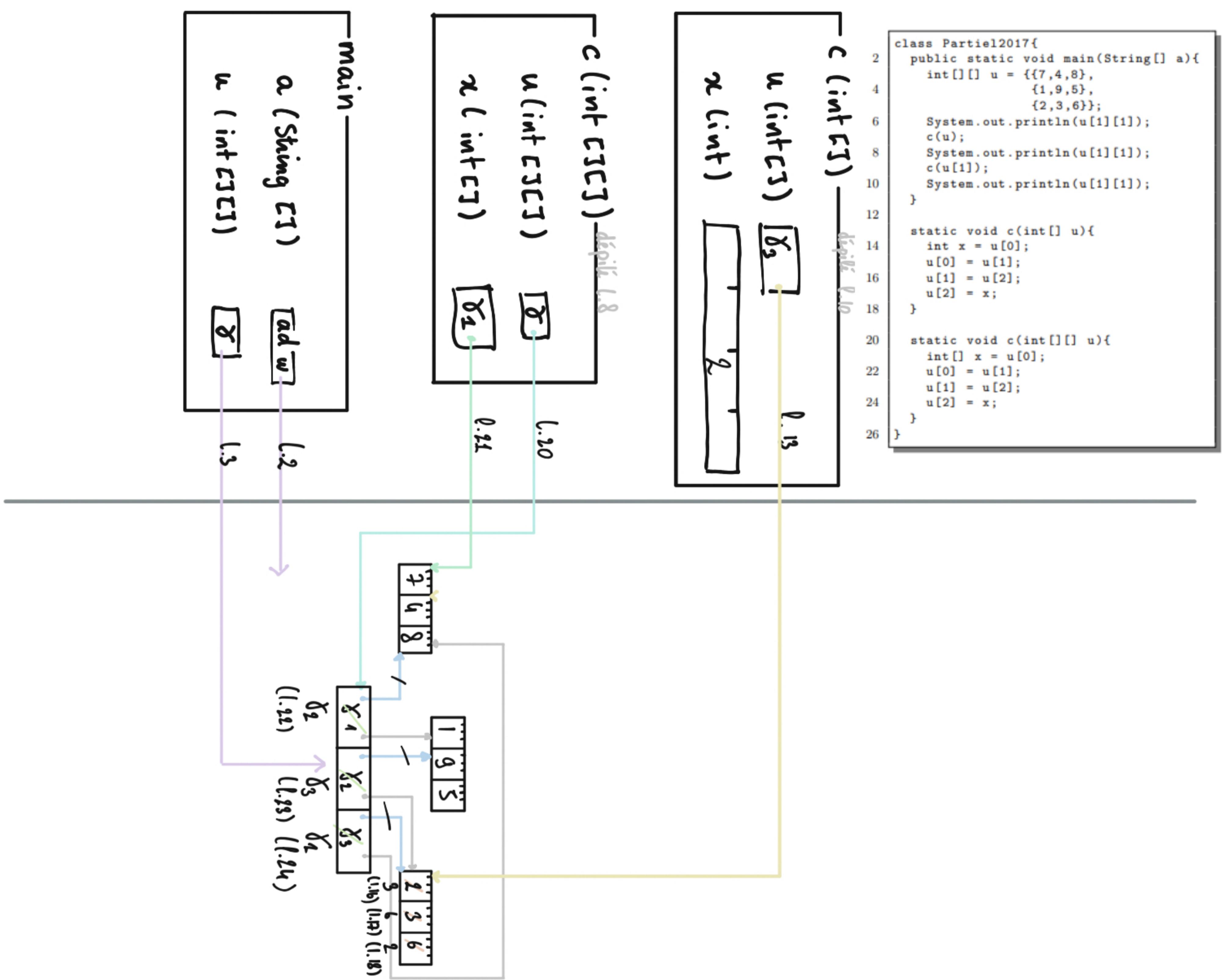
### - infixé -

ex : ( ( 6 \* ( 5 % 3 ) ) - 2 ) - 4 )

infixé > postfixe	infixé > préfixe
<p>on lit l'expression de gauche à droite</p> <ul style="list-style-type: none"><li>→ si c'est un opérande on l'affiche</li><li>→ si c'est ( on l'empile (avec priorité 0)</li><li>→ si c'est ) on dépile et affiche jusqu'à (</li><li>→ si c'est un opérateur : tant que (priorité du sommet de pile <math>\geq</math> priorité de op), on dépile le sommet de pile en l'affichant et on empile op</li></ul> <p>on vide la pile de ses derniers opérateurs en affichant</p>	<p>on lit l'expression de droite à gauche</p> <ul style="list-style-type: none"><li>→ même processus que infixé &gt; postfixe</li></ul> <p>le résultat est inversé</p>



## < la mémoire dans un programme >



## < traduction de programmes >

### Rappels sur la traduction de tableaux

#### Déclaration

Un tableau n'est rien d'autre qu'une suite de cases mémoire. Allouer un tableau, via par exemple `int[] t = new int[20]`, revient à « réservé » un bloc de 20 cases dans la mémoire pour y stocker des entiers.

Tout comme une déclaration sans affectation d'une variable, c'est une décision que vous prenez en annotant le programme à traduire, et qui ne correspond pas à une instruction dans le programme traduit. En effet, la mémoire est déjà allouée au début de votre programme via `int[] mem = new int[100000]`.

À titre d'exemple, considérez l'extrait à traduire suivant :

```
int a;  
int[] t = new int[10];  
int b = 4;
```

Cet extrait contient trois choses distinctes.

1. une déclaration de variable `int a` sans affectation, qui ne correspondra donc à aucune instruction dans le programme mais qui doit être annotée afin de choisir un emplacement mémoire pour la variable `a`.
2. une déclaration de tableau `int[] t = new int[10]`, qui, comme pour l'instruction précédente, ne correspondra à aucune instruction dans le programme. Cependant, vous devez aussi l'annoter afin de choisir 10 emplacements mémoire pour le tableau `t`.
3. enfin, une déclaration avec affectation de variable `int b = 4`. Le plus simple est de voir cette instruction comme deux instructions en séparant la déclaration de l'affectation comme ceci :

```
int b;  
b = 4;
```

La déclaration peut être traitée comme le premier point, et l'affectation correspond à une instruction.

En résumé, le programme peut être annoté comme ceci :

```
int a; // a = mem[0]  
int[] t = new int[10]; // t = mem[1..10]  
int b = 4; // b = mem[11], affectation case 0  
//instruction  
case 0: mem[11] = 4; break;
```

## Accès

La traduction des accès aux tableaux peut-être faite de manière très mécanique en ayant bien annoté le programme.

Reprenez l'exemple précédent : Comment traduire un accès à  $t[0]$  ? Et bien, un tel accès revient à accéder à la première case de notre tableau, donc celle que nous avons décidé être  $\text{mem}[1]$ . Un accès à  $t[1]$  est un accès à la deuxième case, donc  $\text{mem}[2]$ .

Si vous continuez un peu comme ça, vous remarquerez qu'un accès à la case  $i$  via  $t[i]$  est traduit par un accès à la case mémoire  $\text{mem}[1+i]$ . Vous pouvez donc traduire mécaniquement tous les accès à  $t$  dans votre programme. Par exemple, si votre programme contient l'extrait suivant (annoté un peu au pif) :

```
t[0] = 4; // case 5
t[3] = 5; // case 6
t[b] = 2; // case 7, n'oubliez pas que b est mem[11]
```

alors vous pouvez traduire ça directement en :

```
case 5: mem[1+0] = 4; break;
case 6: mem[1+3] = 5; break;
case 7: mem[1+mem[11]] = 2; break;
```

Et si le tableau ne commence pas en  $\text{mem}[1]$  ?

La traduction  $t[i] \rightarrow \text{mem}[1+i]$  ne fonctionne que parce que le tableau commence en  $\text{mem}[1]$ . En effet, si vous refaites le raisonnement précédent avec un tableau commençant, par exemple, en  $\text{mem}[3]$ , vous remarquerez que l'accès  $t[0]$  est traduit par  $\text{mem}[3]$ . Plus généralement, l'accès à  $t[i]$  est traduit par  $\text{mem}[3+i]$ .

Donc, de manière générale, si votre tableau commence en  $\text{mem}[n]$  alors l'accès à  $t[i]$  est traduit par  $\text{mem}[n+i]$ .

## Application (TD 3, exercice 4)

Tout d'abord, voici comment il est possible d'annoter le programme. Le tableau est déclaré avant tout le reste, donc il sera placé au tout début de la mémoire. L'accès à  $t[i]$  sera simplement traduit par un accès  $\text{mem}[0+i]$ , ou encore par  $\text{mem}[i]$

```
public static void main(String[] arg) {
    int[] t = new int[20];                                // t = mem[0..19]
    t[0] = 1;                                            // case 0
    t[1] = 1;                                            // case 1
    for(int i = 2;                                       // i = mem[20], affectation case 2
        i < 20;                                         // case 3
        i++)                                              // case 5, annoté après avoir
    annoté le corps de la boucle
    t[i] = 2 * t[i-1] + t[i-2];                          // case 4
                                                        // case 6, n'oubliez pas
    l'accolade qui devrait se trouver là
                                                        // pour fermer la boucle
    System.out.println("Nombre(19) =" + t[19]); // case 7
}
                                                // case 8, exit

class ProgTrad {

    public static void main(String [] args) {

        // Déclaration du compteur d'instruction et de la mémoire (pas de pile ici)
        int ic = 0;
        int mem[] = new int[100000];

        while (true) {
            switch (ic++) {
                case 0: mem[0] = 1; break;
                case 1: mem[1] = 1; break;
                case 2: mem[20] = 2; break; // i est en mem[20]
                case 3: if (!(mem[20]<20)) ic += 3; break;
                case 4: mem[mem[20]] = 2 * mem[mem[20]-1] + mem[mem[20]-2]; break;
                case 5: mem[20]++; break;
                case 6: ic -= 4; break;
                case 7: System.out.println("Nombre(19) =" + mem[19]);
                case 8: System.exit(0);
            }
        }
    }
}
```

## Application (TD 4, exercice 1)

Le main commence en 0, la fonction f en 100, g en 200 et h en 300. Vous êtes bien entendu libres de choisir d'autres valeurs, tant que les indices au sein d'une même fonction sont successifs.

```
class Basique {
    static void h(){
        System.out.println("Entrée h"); // case 300
        g(); // case 301
        System.out.println("Sortie h"); // case 302
    } // case 303

    static void g(){
        System.out.println("Entrée g"); // case 200
        f(); // case 201
        System.out.println("Sortie g"); // case 202
    } // case 203

    static void f(){
        System.out.println("Milieu f"); // case 100
    } // case 101

    public static void main(String[] args){
        System.out.println("Entrée main"); // case 0
        f(); // case 1
        h(); // case 2
        System.out.println("Sortie main"); // case 3
    } // case 4
}
```

La traduction du programme est alors la suivante.

```
class BasiqueTrad {

    public static void main(String [] args) {

        int ic = 0;
        int mem[] = new int[100000]; // En fait la mémoire n'est pas nécessaire ici
        Stack<Integer> stack = new Stack<Integer>();
        while (true) {
            switch (ic++) {

                // Fonction main
                case 0: System.out.println("Entrée main"); break;
                case 1: stack.push(ic); ic = 100; break;
                case 2: stack.push(ic); ic = 300; break;
                case 3: System.out.println("Sortie main"); break;
                case 4: System.exit(0);

                // Fonction f
                case 100: System.out.println("Milieu f"); break;
                case 101: ic = stack.pop(); break;

                // Fonction g
                case 200: System.out.println("Entrée g"); break;
                case 201: stack.push(ic); ic = 100; break;
                case 202: System.out.println("Sortie g"); break;
                case 203: ic = stack.pop(); break;

                // Fonction h
                case 300: System.out.println("Entrée h"); break;
                case 301: stack.push(ic); ic = 200; break;
                case 302: System.out.println("Sortie h"); break;
                case 303: ic = stack.pop(); break;
            }
        }
    }
}
```

# Traduction des fonctions avec environnement et valeurs de retour

## Stockage des valeurs des paramètres

```
static void f(int x, int y) {  
    ...  
}
```

il nous faut un bloc contenant ic comme toujours, mais aussi les valeurs de x et y. Ainsi, au tout début de la traduction, il faut donc créer la classe suivante :

```
class BlockA {  
    int arg0; // valeur de x pour la fonction f  
    int arg1; // valeur de y pour la fonction f  
    int ric; // valeur de ic à laquelle retourner après l'appel  
  
    public BlockA(int arg0, int arg1, int ric) {  
        this.arg0 = arg0;  
        this.arg1 = arg1;  
        this.ric = ric;  
    }  
}
```

Il faut aussi changer la déclaration de la pile afin que celle-ci ne stocke plus uniquement des entiers, mais des blocs complets :

```
Stack<BlockA> stack = new Stack<BlockA>();
```

## Traduction des appels

La traduction des appels ne change pas beaucoup par rapport à avant. Jusqu'à maintenant, on empilait simplement la valeur de ic puis on faisait un saut, en modifiant ic. Le saut ne change pas, en revanche, il nous faut maintenant empiler un bloc complet.

```
static void f(int x, int y) {  
    ... // on suppose que f commence au case 100  
}  
  
public static void main(String[] args) {  
    f(3, 5); // case 0  
}  
// instruction  
case 0: stack.push(new Block(3, 5, ic)); ic = 100; break;
```

## Traduction des fonctions avec paramètres

La traduction du corps des fonctions change aussi un peu, puisque ces dernières peuvent maintenant accéder aux valeurs de leurs paramètres. Il faut donc aller chercher ces valeurs dans le bloc au sommet de la pile. Pour aller chercher une valeur dans le bloc sans le dépiler, on utilisera l'instruction `stack.peek()` qui renvoie le sommet de la pile sans le dépiler.

```
static void f(int x, int y) {
    System.out.println(x + y);    // case 100
}                                // case 101

// l'accès à x = valeur arg0 => stack.peek().arg0, et de même pour y avec arg1.
// N'oubliez pas, comme d'habitude, de dépiler le bloc et de restaurer ic (à la
// valeur de ric dans le bloc) dans le case 101.
case 100: System.out.println(stack.peek().arg0 + stack.peek().arg1); break;
case 101: ic = stack.pop().ric; break;
```

## Cas des fonctions avec valeur de retour

Nous avons vu jusqu'à maintenant le cas des fonctions sans valeur de retour, il nous reste juste à ajouter cette possibilité. Mais, encore une fois, le plus dur est fait. :-)

Lors d'un appel, l'appelant doit transmettre des informations à la fonction appelée. Ici, c'est l'inverse : la fonction doit transmettre des informations (la valeur de retour) à l'appelant. On va utiliser le même principe, c'est à dire ajouter l'information à la pile, dans laquelle l'appelant pourra lire.

Considérons donc la fonction suivante :

```
static int f(int x, int y) {
    return (x+y);
}
```

Il faut que l'appelant dépile lui-même le bloc après l'appel, et après avoir traité la valeur de retour. Ceci implique que l'appel d'une fonction prenne systématiquement deux instructions (l'appel et le dépilement), pour une seule ligne de code.

```
static int f(int x, int y) {
    return (x+y);           // case 100, affectation de ret
}                           // case 101, restoration de ic

public static void main(String[] args) {
    int a;                 // mem[0]
    a = f(3,5);            // case 0 (appel), case 1 (dépilement et
    affectation)
    System.out.println(a); // case 2
}                           // case 3
```

La traduction complète serait :

```
class BlockA {
    int arg0;
    int arg1;
    int ric;
    int ret;

    public BlockA(int arg0, int arg1, int ric) {
        this.arg0 = arg0;
        this.arg1 = arg1;
        this.ric = ric;
    }
}

class ProgTrad {

    public static void main(String [] args) {

        int ic = 0;
        int mem[] = new int[100000];
        Stack<BlockA> stack = new Stack<BlockA>();

        while (true) {
            switch (ic++) {
                // fonction main
                case 0: stack.push(new BlockA(3,5,ic)); ic = 100; break;
                case 1: mem[0] = stack.pop().ret; break; // ici on dépile le bloc !
                case 2: System.out.println(mem[0]); break;
                case 3: System.exit(0);

                // fonction f
                case 100: stack.peek().ret = stack.peek().arg0 + stack.peek().arg1;
                break;
                case 101: ic = stack.peek().ric; break;
            }
        }
    }
}
```