

# Compilation : Boucles et MVaP

[L3 Informatique] Théorie des langages et compilation

---

Gaétan Richard

2020-2021



## Instructions de bases MVaP

---

Code	Pile	sp	pc	Condition
PUSHI $n$	$P[sp] := n$	$sp+1$	$pc+2$	$n$ entier
POP		$sp-1$	$pc+1$	

- La commande **PUSHI** attend un argument  $n$  qui doit être un entier (sinon erreur d'exécution).
- Si cette exécution a lieu alors que la pile vaut  $P$ , que le registre de sommet de pile vaut  $sp$  et le compteur de programme vaut  $pc$ , alors après l'exécution du programme, la pile est modifiée (valeur  $n$  à l'adresse  $sp$ ); les registres  $sp$  et  $pc$  sont respectivement incrémentés de 1 et 2.

Code	Pile	sp	pc	Condition
ADD	$P[sp-2] := P[sp-2] + P[sp-1]$	sp-1	pc+1	
SUB	$P[sp-2] := P[sp-2] - P[sp-1]$	sp-1	pc+1	
MUL	$P[sp-2] := P[sp-2] * P[sp-1]$	sp-1	pc+1	
DIV	$P[sp-2] := P[sp-2] / P[sp-1]$	sp-1	pc+1	

- La division produit une erreur si le dividende est nul.

Code	Pile	sp	pc	Condition
READ	P[sp] := entier lu	sp+1	pc+1	un entier sur l'entrée standard
WRITE		sp	pc+1	

- WRITE affiche la valeur sur le haut de la pile sans y toucher.

## Fin de programme

Code	Pile	sp	pc	Condition
HALT				

## Les variables

---

- En général, une variable globale est déclarée en début du fichier source et est valable jusqu'à la fin de l'exécution;
- Dans certains langages, il est possible de définir des variables à l'intérieur de blocs. Leur portée est limitée à ce bloc.

Cas du C :

```
#include <stdio.h>
int main (void) {
    int j;
    j = 0;
    for (int i = 0; i < 10; i++) {
        j = i * j + 2;
    }
    printf("%d,%d", j, j);
}
```

- Empiler la valeur d'une variable globale qui est contenue dans la pile
- Recopier la valeur en sommet de pile dans la variable globale

## Instructions MVaP

Code	Pile	sp	pc	Condition
PUSHG $n$	$P[sp] := P[n]$	$sp+1$	$pc+2$	$n < sp$
STOREG $n$	$P[n] := P[sp-1]$	$sp-1$	$pc+2$	$n < sp$



## Besoins

- **Déclaration** : réservation de l'espace mémoire et mémorisation du liens entre nom et emplacement mémoire.
- **Assignation / Utilisation** : retrouver l'emplacement mémoire et l'utiliser pour stocker ou charger la valeur.

## Mémorisation

Pour stocker ces informations, on utilise une **table des symboles** qui contient l'ensemble des symboles présent dans le source et permet en cas de besoin d'associer entre autre une adresse.

En antlr, il est possible de déclarer des structures utilisées sur l'ensemble de l'analyse

```
@members {  
    HashMap<String, Integer> memory = new HashMap<String, Integer>();  
}
```

et des headers java

```
@header {  
    import java.util.HashMap;  
}
```

## Opcodes MVàP pour les branchements

---

**Changer** : pour changer l'exécution, il suffit de changer la valeur du `pc`.

### Instruction MVaP

Code	Pile	sp	pc	Condition
<b>JUMP</b> Label		sp	adresse(label)	dans le source
JUMP n		sp	n	en binaire

Pour déterminer où aller, on indique des **étiquettes** dans le code MVaP :

**LABEL** Label

### Coût

Dans un processeur réel, une telle opération est coûteuse car elle nécessite de *casser* le pipeline.

**Principe** : L'instruction de branchement est exécuté suivant une condition.

### Instruction MVaP

Code	Pile	sp	pc	Condition
JUMPF label		sp-1	pc+1 <b>si</b> $P[sp-1] \neq 0$ adresse(label) <b>sinon</b>	dans le source

Dans le code assemblé, on a directement l'adresse du saut.

Code	Pile	sp	pc	Condition
INF	$P[\text{sp}-2] := 1$ <b>si</b> $P[\text{sp}-2] < P[\text{sp}-1]$ , 0 <b>sinon</b>	sp-1	pc+1	
INFEQ	$P[\text{sp}-2] := 1$ <b>si</b> $P[\text{sp}-2] \leq P[\text{sp}-1]$ , 0 <b>sinon</b>	sp-1	pc+1	
SUP	$P[\text{sp}-2] := 1$ <b>si</b> $P[\text{sp}-2] > P[\text{sp}-1]$ , 0 <b>sinon</b>	sp-1	pc+1	
SUPEQ	$P[\text{sp}-2] := 1$ <b>si</b> $P[\text{sp}-2] \geq P[\text{sp}-1]$ , 0 <b>sinon</b>	sp-1	pc+1	
EQUAL	$P[\text{sp}-2] := 1$ <b>si</b> $P[\text{sp}-2] = P[\text{sp}-1]$ , 0 <b>sinon</b>	sp-1	pc+1	
NEQ	$P[\text{sp}-2] := 1$ <b>si</b> $P[\text{sp}-2] \neq P[\text{sp}-1]$ , 0 <b>sinon</b>	sp-1	pc+1	

## Branchements et assemblages

---

PUSHI 12

PUSHI 24

INFEQ

JUMPF suite

PUSHI 13

JUMP fin

LABEL suite

PUSHI 2

LABEL fin

PUSHI 3

ADD

WRITE



```
LBB0_1:                                ## =>This Inner Loop Header: Depth=1
    cmpl    $20, -8(%rbp)
    jge     LBB0_4

## BB#2:                                ##   in Loop: Header=LBB0_1 Depth=1

    movl    -12(%rbp), %eax
    imull    $3, -8(%rbp), %ecx
    addl     %ecx, %eax
    addl     $5, %eax
    movl     %eax, -12(%rbp)

## BB#3:                                ##   in Loop: Header=LBB0_1 Depth=1

    movl     -8(%rbp), %eax
    addl     $1, %eax
    movl     %eax, -8(%rbp)
    jmp     LBB0_1

LBB0_4:
```

```
0:  iconst_2
1:  istore_1
2:  iload_1
3:  sipush 1000
6:  if_icmpge      44
9:  iconst_2
10: istore_2
11: iload_2
12: iload_1
13: if_icmpge      31
16: iload_1
17: iload_2
18: irem
19: ifne      25
22: goto      38
25: iinc      2, 1
28: goto      11
31: getstatic    #84;// Field java/lang/System.out:Ljava/io/PrintStream;
34: iload_1
35: invokevirtual #85;// Method java/io/PrintStream.println:(I)V
38: iinc      1, 1
41: goto      2
44: return
```

## Label vers adresse :

- L'assemblage transforme les labels en adresses;
- On peut ajouter deux codes sources mvap à la suite mais pas deux codes assemblés;
- On peut *relativiser* chaque adresse par rapport à un registre global (code PIC).

## Deux actions :

- Un assembleur `java -cp " :/usr/share/java/* :MVaP.jar" MVaPAssembler fichier.mvap`
- Une machine virtuelle `java -jar MVaP.jar -d fichier.mvap.cbab`

## Les sources :

- Disponibles sur le site du cours en tgz;
- Contient un fichier **ALIRE.md** et un **Makefile**;
- Produit l'archive **MVaP.jar**

## Exemple

```
int i
i = 6
while (i < 10) i = i + 1
print(i)
```

```
PUSHI 0
JUMP Start
LABEL Start
PUSHI 6
STOREG 0
LABEL DebutB
PUSHG 0
PUSHI 10
INF
JUMPF FinB
PUSHG 0
PUSHI 1
ADD
STOREG 0
JUMP DebutB
LABEL FinB
PUSHG 0
WRITE
POP
HALT
```

Adr		Instruction
+---+-----		
0		PUSHI 0
2		JUMP 4
4		PUSHI 6
6		STOREG 0
8		PUSHG 0
10		PUSHI 10
12		INF
13		JUMPF 24
15		PUSHG 0
17		PUSHI 1
19		ADD
20		STOREG 0
22		JUMP 8
24		PUSHG 0
26		WRITE
27		POP
28		HALT

## Boucles

---

## Structures basiques

```
| 'if' '(' c=condition ')' blocthen=instruction ('else' blocelse=instruction)?  
| 'while' '(' c=condition ')' i=instruction  
| 'for' '(' init=assignation? ';' c=condition? ';' incr=assignation? ')' i=instruction  
| 'do' i=instruction 'until' '(' c=condition? ')' finInstruction
```

## Compilation

- Évaluer les conditions;
- Mettre les étiquettes;
- Ajouter les sauts conditionnels;
- Compiler le code.

## Syntaxe

```
'while' '(' c=condition ')' i=instruction
```

## Notes

- Dans tous les langages de programmation;
- Utile pour le cas de boucles dont on ne connaît pas le nombre d'itérations;
- Attention à bien faire changer la condition pour éviter une boucle infinie.



## Syntaxe

```
'do' i=instruction 'until' '(' c=condition ')' finInstruction
```

## Notes

- Assez peu utilisée;
- Très pratique pour un bloc devant au moins être effectué une fois;
- Permet parfois une meilleure compréhension.

## Syntaxe

```
'for' '(' init=assignation? ';' c=condition? ';' incr=assignation? ')' i=ins
```

## Notes

- Une boucle présente partout;
- Certains langages utilisent un 'for' IDENTIFIANT 'in' expression
- **Attention** : mieux vaut ne pas modifier la variable sur laquelle porte la boucle.

## Altération

De part son caractère très générique, il existe de nombreux abus de la boucle `for`.

- Transformer un `'while'` en `'for'`;
- Mettre un corps vide;
- Mélanger condition et action;
- ...

## Utilisation

En dehors des idiomes sur les chaînes, il est préférable d'éviter ces utilisations ou d'indiquer leur principe sur un commentaire.

## Condition et évaluation

---

### Exemple

```
false and (y == 4 + x or x = y+3*y^2/4x+45)
```

On sait dès le départ qu'il est inutile d'évaluer la partie droite.

### Et en C?

```
int x = 4;  
if ( 1 && x++) {}  
printf("%d\n",x);
```

**Attention** : Certains comportements sont détaillés dans la norme mais d'autres sont laissés au choix du compilateur.