

# Compilation : Fonctions

[L3 Informatique] Théorie des langages et compilation

---

Gaétan Richard

2024-2025



Vers les fonctions

---

### Idée

- On peut isoler une portion de code entre un `label` et un `jump`;
- on pourrait vouloir utiliser ce code depuis deux endroits différents de notre programme.

### Problème

on ne peut pas revenir à deux endroits différents.

### Solution

on va sauvegarder dans un endroit connu l'adresse de l'endroit d'où l'on part.

### Endroit connu pour nous

sur le haut de la pile.

## Problèmes

- on peut arriver dans le code depuis des endroits ayant des tailles et contenus de pile différents.
- comment faire si on veut parler de variables locales (paramètres de la fonction).

## Solution

voir une variable qui donne la position de la pile au moment de l'arrivée : **fp**

## Travail supplémentaire

sauvegarder l'ancienne valeur avant d'appeler la fonction, calculer la nouvelle valeur, restaurer la valeur à la sortie.

## Où ça

toujours sur la pile.

## Question

comment passer des arguments ?

## Solution

les positionner en haut de la pile juste avant de passer au code de la fonction.

### Question

et pour la valeur de retour?

### Réponse

La positionner en haut de la pile (réserver de la place) juste avant de passer au code de la fonction.

## En résumé : que doit-on sauvegarder/restaurer ?

- Lors du retour normal de la procédure la suite de l'exécution doit se poursuivre à l'instruction suivant l'instruction d'appel. → Le compteur de programme **pc** doit donc être sauvegardé à chaque appel
- Les données locales à la procédure s'organisent dans la pile à partir d'une adresse appelée **frame pointer** qui est conservée dans un registre spécial **fp**

Lorsqu'une nouvelle procédure est appelée, cette valeur change, il est donc nécessaire de sauvegarder la valeur courante qui devra être restaurée à la fin de la procédure.

- Les opérations à effectuer lors d'un appel de procédure se partagent entre l'appelant et l'appelé.

**L'appelant et l'appelé doivent avoir une vision cohérente de l'organisation de la mémoire**

# Fonctions

---

## Principe

des instructions assembleur dédiées à l'appel de fonction.

## Opcodes

Code	Pile	sp	pc	Condition
CALL label	...	...	adresse(label)	
RETURN		...	...	

## Action

- L'instruction **CALL** prend comme argument une adresse dans le code d'instruction;
- Le compteur d'instructions `pc` se place alors à cette adresse, son ancienne valeur est sauvegardée;
- **CALL** sauvegarde la valeur de `fp`;
- **CALL** positionne le registre `fp` (adresse du bloc d'activation) à la valeur courante de `sp`.

## Action

- L'instruction **RETURN** retrouve l'ancienne valeur du compteur de programme `pc` et se place à l'instruction suivante.
- Elle repositionne `sp` à la valeur courante de `fp` et restaure `fp` à son ancienne valeur

## Exemple d'utilisation

Appel de  $f(1, 2)$ , l'instruction suivante de l'appel étant à l'adresse 15 (valeur courante de  $pc$ ).  
On suppose que  $fp$  vaut 0.

Qui	Quoi	Pile
Appelant	Empile les valeurs <b>des arguments</b> appelle <b>CALL</b> en donnant l'adresse de la fonction	[] [1, 2]
Machine	Le <b>CALL</b> empile l' <b>adresse de retour</b> et la valeur du <b>frame pointer</b>	[1, 2, 15, 0]
Appellé	Exécute son code qui se termine par <b>RETURN</b>	[1, 2, 15, 0]
Machine	Le <b>RETURN</b> dépile tout ce que la procédure a empilé et n'a pas dépilé jusqu'à dépiler le frame pointeur et le compteur ordinal qu'elle restaure	[1, 2]
Appelant	Dépille les arguments qu'il avait empilés	[]

## Exemple avec valeur de retour

C'est similaire mais il faut garder de la place sur la pile avant l'appel de la fonction.

Qui	Quoi	Pile
Appelant	Laisse de la place pour la valeur de retour et empile les valeurs des arguments appelle CALL en donnant l'adresse de la fonction	[] [0, 1, 2]
Machine	Le CALL empile l'adresse de retour et la valeur du frame pointer	[0, 1, 2, 15, 0]
Appelé	Exécute son code qui se termine par RETURN met à jour la valeur de retour	[42, 1, 2, 15, 0]
Machine	Le RETURN dépile tout ce que la procédure a empilé et n'a pas dépilé jusqu'à dépiler le frame pointeur et le compteur ordinal qu'elle restaure	[42, 1, 2]
Appelant	Dépille les arguments qu'il avait empilé et utilise la valeur de retour (ou pas)	[42]

## Exemple si appel depuis une autre fonction

Appel de  $f(1, 2)$ , l'instruction suivante de l'appel étant à l'adresse 15 (valeur courante de  $pc$ ).  
On suppose que  $fp$  vaut 51 (valeur courante de  $fp$  pour la fonction appelante).

Qui	Quoi	Pile
Appelant	Laisse de la place pour la valeur de retour et empile les valeurs des arguments appelle <b>CALL</b> en donnant l'adresse de la fonction	[] [0, 1, 2]
Machine	Le <b>CALL</b> empile l'adresse de retour et la valeur du frame pointer	[0, 1, 2, 15, 51]
Appelé	Exécute son code qui se termine par <b>RETURN</b> met à jour la valeur de retour	[42, 1, 2, 15, 51]
Machine	Le <b>RETURN</b> dépile tout ce que la procédure a empilé et n'a pas dépilé jusqu'à dépiler le frame pointer et le compteur ordinal qu'elle restaure	[42, 1, 2]
Appelant	Dépille les arguments qu'il avait empilé et utilise la valeur de retour (ou pas)	[42]

## Paramètres et variables locales

---

Le registre `fp` est à jour au début de l'appel de procédure et permet de référencer les valeurs locales.

Aux instructions `STOREG`, `PUSHG` correspondent les instructions `STOREL`, `PUSHL` qui ont le même comportement mais relatif à `fp`.

### Opcodes

Code	Pile	sp	pc	Condition
<code>PUSHL n</code>	$P[sp] := P[fp+n]$	$sp+1$	$pc+2$	$fp + n < sp$
<code>STOREL n</code>	$P[fp+n] := P[sp-1]$	$sp-1$	$pc+2$	$fp + n < sp$

## Un exemple complet

---

- Les opérations à effectuer lors d'un appel de procédure se partagent entre l'appelant et l'appelé.
- L'appelant effectue la réservation pour la valeur de retour dans le cas d'une fonction et évalue ces paramètres effectifs
- L'appelé initialise ses données locales et commence l'exécution du corps de la procédure. Au moment du retour, l'appelé place éventuellement le résultat de l'évaluation à l'endroit réservé par l'appelant et restaure les registres.

sur la pile le bloc d'activation de la fonction ressemble à

... `<out>` `<args>` `<back adr>` `<old fp>`

Le registre `fp` contient l'adresse de la pile de la dernière case de ce bloc d'activation, ce qui permet d'accéder aux arguments et à la valeur de retour (adresse négative par rapport à `fp`).

## Exemple

```
fun f(int x, int y) -> int
  { return 2 * x + y; }
```

```
f(20, 2);
```

L'appelant empilera donc trois valeurs :

- une pour réserver la **place pour le résultat**
- une deuxième pour la valeur du **premier paramètre**
- une troisième pour la valeur du **second paramètre**

## Exemple (code MVàP généré)

```
On saute dans le «main»  
Début fonction f  
2  
x  
2 * x  
y  
2 * x + y  
stocké dans la pile comme valeur de retour  
Fin fonction f
```

```
Début programme principal  
On réserve la place pour la valeur de retour  
On empile l'argument entier 20  
On empile l'argument entier 2  
On appelle f  
On dépile le 2e argument  
On dépile le 1er argument  
On écrit le résultat  
On dépile le résultat  
Arrêt de la machine
```

```
JUMP Start  
LABEL f  
PUSHI 2  
PUSHL -4  
MUL  
PUSHL -3  
ADD  
STOREL -5  
RETURN  
RETURN  
LABEL Start  
PUSHI 0  
PUSHI 20  
PUSHI 2  
CALL f  
POP  
POP  
WRITE  
POP  
HALT
```

## Exemple (code assemblé et exécution)

### Code assemblé

Adr		Instruction	
0		JUMP	14
2		PUSHI	2
4		PUSHL	-4
6		MUL	
7		PUSHL	-3
9		ADD	
10		STOREL	-5
12		RETURN	
13		RETURN	
14		PUSHI	0
16		PUSHI	20
18		PUSHI	2
20		CALL	2
22		POP	
23		POP	
24		WRITE	
25		POP	
26		HALT	

### Exécution

pc				fp	pile
0		JUMP	14		0 [ ] 0
14		PUSHI	0		0 [ ] 0
16		PUSHI	20		0 [ 0 ] 1
18		PUSHI	2		0 [ 0 20 ] 2
20		CALL	2		0 [ 0 20 2 ] 3
2		PUSHI	2		5 [ 0 20 2 22 0 ] 5
4		PUSHL	-4		5 [ 0 20 2 22 0 2 ] 6
6		MUL			5 [ 0 20 2 22 0 2 20 ] 7
7		PUSHL	-3		5 [ 0 20 2 22 0 40 ] 6
9		ADD			5 [ 0 20 2 22 0 40 2 ] 7
10		STOREL	-5		5 [ 0 20 2 22 0 42 ] 6
12		RETURN			5 [ 42 20 2 22 0 ] 5
22		POP			0 [ 42 20 2 ] 3
23		POP			0 [ 42 20 ] 2
24		WRITE			0 [ 42 ] 1
42					
25		POP			0 [ 42 ] 1
26		HALT			0 [ ] 0

## Récurtivité

---

## Principe

faire appel à une fonction à l'intérieur d'elle même.

## Mise en place

cela marche tout seul.

```
fun fact(int n) -> int {  
    if(n <= 1) return 1;  
    return n*fact(n-1);  
}  
println(fact(3));
```

## Exemple (code généré)

On saute à la «première» instruction

Début fonction fact

si (n <= 1)

retour 1

stocké dans la pile

fin alors

sinon rien

finsi

n

valeur retour

n

1

n - 1

appel fact(n-1)

on dépile (n-1)

n \* fact(n-1)

stocké dans la pile

Fin fonction fact

Début programme principal

valeur retour

paramètre 3

appel fact(3)

on dépile 3

on dépile fact(3)

JUMP Start

LABEL fact

PUSHL -3

PUSHI 1

INFEQ

JUMPF Sinon

PUSHI 1

STOREL -4

RETURN

JUMP FinSi

LABEL Sinon

LABEL FinSi

PUSHL -3

PUSHI 0

PUSHL -3

PUSHI 1

SUB

CALL fact

POP

MUL

STOREL -4

RETURN

RETURN

LABEL Start

PUSHI 0

PUSHI 3

CALL fact

POP

WRITE

POP

HALT

# Exemple (exécution)

Adr	Instruction	pc	fp	pile
0	JUMP 33	0	0	[ ] 0
2	PUSHL -3	33	0	[ ] 0
4	PUSHI 1	35	0	[ 0 ] 1
6	INFEQ	37	0	[ 0 3 ] 2
7	JUMPF 16	2	4	[ 0 3 39 0 ] 4
9	PUSHI 1	4	4	[ 0 3 39 0 3 ] 5
11	STOREL -4	6	4	[ 0 3 39 0 3 1 ] 6
13	RETURN	7	4	[ 0 3 39 0 0 ] 5
14	JUMP 16	16	4	[ 0 3 39 0 ] 4
16	PUSHL -3	18	4	[ 0 3 39 0 3 ] 5
18	PUSHI 0	20	4	[ 0 3 39 0 3 0 ] 6
20	PUSHL -3	22	4	[ 0 3 39 0 3 0 3 ] 7
22	PUSHI 1	24	4	[ 0 3 39 0 3 0 3 1 ] 8
24	SUB	25	4	[ 0 3 39 0 3 0 2 ] 7
25	CALL 2	2	9	[ 0 3 39 0 3 0 2 27 4 ] 9
27	POP	4	9	[ 0 3 39 0 3 0 2 27 4 2 ] 10
28	MUL	6	9	[ 0 3 39 0 3 0 2 27 4 2 1 ] 11
29	STOREL -4	7	9	[ 0 3 39 0 3 0 2 27 4 0 ] 10
31	RETURN	16	9	[ 0 3 39 0 3 0 2 27 4 ] 9
32	RETURN	18	9	[ 0 3 39 0 3 0 2 27 4 2 ] 10
33	PUSHI 0	20	9	[ 0 3 39 0 3 0 2 27 4 2 0 ] 11
35	PUSHI 3	22	9	[ 0 3 39 0 3 0 2 27 4 2 0 2 ] 12
37	CALL 2	24	9	[ 0 3 39 0 3 0 2 27 4 2 0 2 1 ] 13
39	POP	25	9	[ 0 3 39 0 3 0 2 27 4 2 0 1 ] 12
40	WRITE	2	14	[ 0 3 39 0 3 0 2 27 4 2 0 1 27 9 ] 14
41	POP	4	14	[ 0 3 39 0 3 0 2 27 4 2 0 1 27 9 1 ] 15
42	HALT	6	14	[ 0 3 39 0 3 0 2 27 4 2 0 1 27 9 1 1 ] 16
		7	14	[ 0 3 39 0 3 0 2 27 4 2 0 1 27 9 1 ] 15
		9	14	[ 0 3 39 0 3 0 2 27 4 2 0 1 27 9 ] 14

## Question

Que se passe-t-il si on peut définir une fonction à l'intérieur d'une fonction.

## Problème

Accès aux variables locale de la fonction englobante.

## Solution possible

À l'aide des `fp` (et `pc`) sauvegardés dans la pile.

## Problèmes et fonctions

---

Il existe deux façon de passer des arguments :

- le **passage par valeur**, qui recopie l'argument; et,
- le **passage par référence**, qui utilise juste un pointeur.

Dans le passage de paramètre par valeur,  $x$  est une nouvelle variable allouée localement par la procédure dont la valeur est le résultat de l'évaluation de  $e$  (la valeur du paramètre).

- après la fin de la procédure, la place mémoire allouée à la variable  $x$  est libérée. Les modifications apportées à  $x$  ne sont donc plus visibles.
- En l'absence de pointeurs, les seules variables modifiées sont les variables non locales à la procédure explicitement nommées dans les instructions du programme.
- Il est nécessaire de réserver une place proportionnelle à la taille du paramètre ce qui peut être coûteux dans le cas de tableaux.

- On calcule l'adresse de  $e$  (la valeur gauche de l'expression)
- La fonction alloue une variable  $x$  qui est initialisée par la valeur gauche de  $e$ .
- Toute référence à  $x$  dans le corps de la fonction est interprétée comme une opération sur l'objet situé à l'adresse stockée en  $x$ .
- Ce mode de passage occupe une place indépendante de la taille du paramètre (une adresse).

### Notes

- En C, le passage par référence est explicitement programmé par le passage d'un pointeur (adresse mémoire) par valeur.
- En Java, le passage se fait par valeur mais les objets ont pour valeur une référence

## Problème

L'appelant positionne les variables et l'appelé les utilise. Il faut donc que les deux soient d'accord sur l'organisation de la mémoire.

## Incompatibilités

il faut donc que les conventions soient les mêmes pour pouvoir utiliser une fonction venant de l'extérieur.

### Remarque

Lors de la compilation d'une bibliothèque, le compilateur utilisé n'est pas le même que celui qui sera utilisé pour la compilation du programme.

### Problème

Cette différence peut engendrer des différences de compilation lors de l'appel de fonction (optimisations, ...) qui rendent les codes incompatibles.