

# Paradigmes de programmation

Cours 3 : Pile et tas

---

Benoît Montagu — `benoit.montagu@inria.fr`

Préparation à l'agrégation d'informatique — Automne 2022



Au programme aujourd'hui :

- ▶ Pile et tas
- ▶ Appels et retours de fonction
- ▶ Passages de paramètres
- ▶ Quelques éléments d'allocation mémoire
- ▶ Appels terminaux

Pour introduire simplement ces concepts :

- ▶ Un petit langage de programmation, très simple : MINI
- ▶ Une machine abstraite avec pile et tas explicite

# MINI : un petit langage de programmation

---

## Le langage MINI : un petit langage impératif

Un petit langage « *while* » avec références et appels de fonctions :

$b ::= \mathbf{true} \mid \mathbf{false}$	(Booléens)
$n ::= \dots \mid -1 \mid 0 \mid 1 \mid \dots$	(Entiers)
$\ell ::= \ell_1 \mid \ell_2 \mid \dots$	(Adresses)
$v ::= () \mid b \mid n \mid \ell$	(Valeurs)

# Le langage MINI : un petit langage impératif

Un petit langage « *while* » avec références et appels de fonctions :

$b ::= \mathbf{true} \mid \mathbf{false}$	(Booléens)
$n ::= \dots \mid -1 \mid 0 \mid 1 \mid \dots$	(Entiers)
$\ell ::= \ell_1 \mid \ell_2 \mid \dots$	(Adresses)
$v ::= () \mid b \mid n \mid \ell$	(Valeurs)
$i ::= x \mid v$	(Immédiats)
$a ::= i \mid i \boxplus i$	(Termes atomiques)
$\mathbf{ref} \ i \mid !i \mid i := i$	
$f(i, \dots, i)$	

# Le langage MINI : un petit langage impératif

Un petit langage « *while* » avec références et appels de fonctions :

$b ::= \mathbf{true} \mid \mathbf{false}$	(Booléens)
$n ::= \dots \mid -1 \mid 0 \mid 1 \mid \dots$	(Entiers)
$\ell ::= \ell_1 \mid \ell_2 \mid \dots$	(Adresses)
$v ::= () \mid b \mid n \mid \ell$	(Valeurs)
$i ::= x \mid v$	(Immédiats)
$a ::= i \mid i \boxplus i$	(Termes atomiques)
$\mathbf{ref} \ i \mid !i \mid i := i$	
$f(i, \dots, i)$	
$cmd ::= a$	(Commandes)
$x = a$	
$\mathbf{if} \ i \ \mathbf{then} \ \{ \mathit{cmds} \} \ \mathbf{else} \ \{ \mathit{cmds} \}$	
$\mathbf{while} \ i \ \{ \mathit{cmds} \}$	
$cmds ::= cmd^+$	(Séquences de commandes)

# Le langage MINI : un petit langage impératif

Un petit langage « *while* » avec références et appels de fonctions :

$b$	$::=$	<b>true</b>   <b>false</b>	(Booléens)
$n$	$::=$	...   -1   0   1   ...	(Entiers)
$\ell$	$::=$	$\ell_1$   $\ell_2$   ...	(Adresses)
$v$	$::=$	()   $b$   $n$   $\ell$	(Valeurs)
$i$	$::=$	$x$   $v$	(Immédiats)
$a$	$::=$	$i$   $i \boxplus i$   <b>ref</b> $i$   <b>!</b> $i$   $i := i$   $f(i, \dots, i)$	(Termes atomiques)
$cmd$	$::=$	$a$   $x = a$   <b>if</b> $i$ <b>then</b> { $cmds$ } <b>else</b> { $cmds$ }   <b>while</b> $i$ { $cmds$ }	(Commandes)
$cmds$	$::=$	$cmd^+$	(Séquences de commandes)
$decl$	$::=$	<b>fun</b> $f(x, \dots, x) = \{ cmds \}$	(Déclarations de fonction)
$p$	$::=$	$decl^+$	(Programmes)

## Exemples de programme MINI (1)

```
fun fact_while(n) = {  
  i = 1;  
  r = 1;  
  b = i < n;  
  while b {  
    i = i + 1;  
    r = r * i;  
    b = i < n  
  };  
  r  
}
```

```
fun main() = {  
  fact_while(5)  
}
```

- ▶ Un programme *impératif* classique
- ▶ Les variables sont mutables



## Exemples de programme MINI (2)

```
fun fact(n) = {  
  b = n <= 0;  
  if b  
  then 1  
  else {  
    n1 = n - 1;  
    p = fact(n1);  
    n * p  
  }  
}
```

```
fun main () = {  
  fact(5)  
}
```

- ▶ Un programme *récurif* classique
- ▶ Ici aussi : les variables sont mutables mais elles ne sont écrites qu'une fois
- ▶ Car de nouvelles variables sont créées à chaque nouvel appel

## Exemples de programme MINI (3)

```
fun swap(r1, r2) = {  
  v1 = !r1;  
  v2 = !r2;  
  r1 := v2;  
  r2 := v1  
}
```

```
fun main() = {  
  x1 = ref 1;  
  x2 = ref 2;  
  swap(x1, x2);  
  !x1  
}
```

- ▶ Un autre programme impératif classique
- ▶ Un programme qui *alloue* des références...  
... les modifie  
... et les déréférence

# De quoi a-t-on besoin pour évaluer les programmes MINI?

Pour MINI, on choisit *expressément* un modèle simple

## ❓ Quel est le code courant?

▶ Quel code est-on en train d'exécuter au sein de la fonction courante?

👉 Le code est une **liste de commandes** *cmds*

# De quoi a-t-on besoin pour évaluer les programmes MINI?

Pour MINI, on choisit *expressément* un modèle simple

## ❓ Quel est le code courant?

▶ Quel code est-on en train d'exécuter au sein de la fonction courante?

👉 Le code est une **liste de commandes** *cmds*

## ❓ Quelles sont les valeurs des variables courantes?

▶ On parle des variables relatives à l'exécution de la fonction courante

👉 On utilise un **environnement** local : c-à-d une map finie  $E : \text{variables} \rightarrow \text{valeurs}$

# De quoi a-t-on besoin pour évaluer les programmes MINI?

Pour MINI, on choisit *expressément* un modèle simple

## ❓ Quel est le code courant?

▶ Quel code est-on en train d'exécuter au sein de la fonction courante?

👉 Le code est une **liste de commandes** *cmds*

## ❓ Quelles sont les valeurs des variables courantes?

▶ On parle des variables relatives à l'exécution de la fonction courante

👉 On utilise un **environnement** local : c-à-d une map finie  $E : \text{variables} \rightarrow \text{valeurs}$

## ❓ Que doit-on faire une fois que la fonction courante a terminé?

▶ Quelles fonctions étaient « en cours d'exécution » jusqu'ici? (appels imbriqués)

▶ Que doit-on faire du résultat calculé par la fonction courante?

👉 On utilise une **pile**

C-à-d une liste  $\pi$  des actions à effectuer *par la suite*

C'est ce qui est usuellement appelé la « pile d'appels »

# De quoi a-t-on besoin pour évaluer les programmes MINI ?

Pour MINI, on choisit *expressément* un modèle simple

## ❓ Quel est le code courant ?

▶ Quel code est-on en train d'exécuter au sein de la fonction courante ?

👉 Le code est une **liste de commandes** *cmds*

## ❓ Quelles sont les valeurs des variables courantes ?

▶ On parle des variables relatives à l'exécution de la fonction courante

👉 On utilise un **environnement** local : c-à-d une map finie  $E : \text{variables} \rightarrow \text{valeurs}$

## ❓ Que doit-on faire une fois que la fonction courante a terminé ?

▶ Quelles fonctions étaient « en cours d'exécution » jusqu'ici ? (appels imbriqués)

▶ Que doit-on faire du résultat calculé par la fonction courante ?

👉 On utilise une **pile**

C-à-d une liste  $\pi$  des actions à effectuer *par la suite*

C'est ce qui est usuellement appelé la « pile d'appels »

## ❓ Vers quelles valeurs pointent les références ?

👉 On utilise une **mémoire** globale

C-à-d une map finie  $\mu : \text{adresses} \rightarrow \text{valeurs}$

C'est ce qui est usuellement appelé « tas »

# Une machine abstraite pour évaluer les programmes MINI

- Un programme transforme une configuration en une autre :

$conf$	$::=$	$(cmds, E, \pi, \mu)$	(Configuration)
$E$	$::=$	$\cdot \mid E[x \mapsto v]$	(Environnement)
$\pi$	$::=$	$\cdot \mid f : \pi$	(Pile d'appels)
$f$	$::=$	$(x, cmds, E) \mid (\_, cmds, E)$	(Trame de pile)
$\mu$	$::=$	$\cdot \mid \mu[\ell \mapsto v]$	(Mémoire)

# Une machine abstraite pour évaluer les programmes MINI

- ▶ Un programme transforme une configuration en une autre :

$conf$	$::=$	$(cmds, E, \pi, \mu)$	(Configuration)
$E$	$::=$	$\cdot \mid E[x \mapsto v]$	(Environnement)
$\pi$	$::=$	$\cdot \mid f : \pi$	(Pile d'appels)
$f$	$::=$	$(x, cmds, E) \mid (\_, cmds, E)$	(Trame de pile)
$\mu$	$::=$	$\cdot \mid \mu[\ell \mapsto v]$	(Mémoire)

- ▶ On va définir une relation d'évaluation *pas à pas* entre configurations :

$conf_1 \rightarrow conf_2$  se lit : «  $conf_1$  se réduit en  $conf_2$  »

**i** Vocabulaire : sémantique opérationnelle à petits pas

- ▶ Configuration initiale :  $([main()], \cdot, \cdot, \cdot)$  (appel à la fonction *main*)
- ▶ Configuration finale :  $([v], E, \cdot, \mu)$  (valeur obtenue, plus rien à faire ensuite)



# Une machine abstraite pour évaluer les programmes MINI

- ▶ Un programme transforme une configuration en une autre :

$conf$	$::=$	$(cmds, E, \pi, \mu)$	(Configuration)
$E$	$::=$	$\cdot \mid E[x \mapsto v]$	(Environnement)
$\pi$	$::=$	$\cdot \mid f : \pi$	(Pile d'appels)
$f$	$::=$	$(x, cmds, E) \mid (\_, cmds, E)$	(Trame de pile)
$\mu$	$::=$	$\cdot \mid \mu[\ell \mapsto v]$	(Mémoire)

- ▶ On va définir une relation d'évaluation *pas à pas* entre configurations :

$conf_1 \rightarrow conf_2$  se lit : «  $conf_1$  se réduit en  $conf_2$  »

**i** Vocabulaire : sémantique opérationnelle à petits pas

- ▶ Configuration initiale :  $([main()], \cdot, \cdot, \cdot)$  (appel à la fonction *main*)
- ▶ Configuration finale :  $([v], E, \cdot, \mu)$  (valeur obtenue, plus rien à faire ensuite)
- ▶ Un modèle simple :

- ▶ Pas de pointeur vers la pile
- ▶ Pile et tas sont disjoints
- ▶ Références allouées dans le tas
- ▶ Pas de registres
- ▶ Pile et tas sont structurés
- ▶ Variables locales allouées dans la pile

## Évaluation « pas à pas » de la machine abstraite (1)

Évaluation des immédiats :

$$\llbracket i \rrbracket_E = \begin{cases} v & \text{si } i = v \\ E(x) & \text{si } i = x \end{cases}$$

# Évaluation « pas à pas » de la machine abstraite (1)

Évaluation des immédiats : 
$$\llbracket i \rrbracket_E = \begin{cases} v & \text{si } i = v \\ E(x) & \text{si } i = x \end{cases}$$

Conditionnelles :

$(\text{if } i \text{ then } \{ cmds_1 \} \text{ else } \{ cmds_2 \} :: cmds, E, \pi, \mu) \rightarrow (cmds_1 ++ cmds, E, \pi, \mu)$   
si  $\llbracket i \rrbracket_E = \mathbf{true}$

$(\text{if } i \text{ then } \{ cmds_1 \} \text{ else } \{ cmds_2 \} :: cmds, E, \pi, \mu) \rightarrow (cmds_2 ++ cmds, E, \pi, \mu)$   
si  $\llbracket i \rrbracket_E = \mathbf{false}$

# Évaluation « pas à pas » de la machine abstraite (1)

Évaluation des immédiats : 
$$\llbracket i \rrbracket_E = \begin{cases} v & \text{si } i = v \\ E(x) & \text{si } i = x \end{cases}$$

Conditionnelles :

$(\mathbf{if } i \mathbf{ then } \{ cmds_1 \} \mathbf{ else } \{ cmds_2 \} :: cmds, E, \pi, \mu) \rightarrow (cmds_1 ++ cmds, E, \pi, \mu)$   
si  $\llbracket i \rrbracket_E = \mathbf{true}$

$(\mathbf{if } i \mathbf{ then } \{ cmds_1 \} \mathbf{ else } \{ cmds_2 \} :: cmds, E, \pi, \mu) \rightarrow (cmds_2 ++ cmds, E, \pi, \mu)$   
si  $\llbracket i \rrbracket_E = \mathbf{false}$

Boucles :

$(\mathbf{while } i \{ cmds' \} :: cmds, E, \pi, \mu) \rightarrow (cmds' ++ \mathbf{while } i \{ cmds' \} :: cmds, E, \pi, \mu)$   
si  $\llbracket i \rrbracket_E = \mathbf{true}$

$(\mathbf{while } i \{ cmds' \} :: cmds, E, \pi, \mu) \rightarrow ( () :: cmds, E, \pi, \mu )$   
si  $\llbracket i \rrbracket_E = \mathbf{false}$

Production de valeurs :

$$\begin{aligned}(i :: cmds, E, \pi, \mu) &\longrightarrow (cmds, E, \pi, \mu) && \text{si } \llbracket i \rrbracket_E = () \text{ et } cmds \neq [] \\(i_1 \boxplus i_2 :: cmds, E, \pi, \mu) &\longrightarrow (\llbracket i_1 \rrbracket_E \boxplus \llbracket i_2 \rrbracket_E :: cmds, E, \pi, \mu) \\(\mathbf{ref} \ i :: cmds, E, \pi, \mu) &\longrightarrow (\ell :: cmds, E, \pi, \mu[\ell \mapsto \llbracket i \rrbracket_E]) && \text{avec } \ell \notin \text{dom } \mu \\(!i :: cmds, E, \pi, \mu) &\longrightarrow (\mu(\ell) :: cmds, E, \pi, \mu) && \text{avec } \llbracket i \rrbracket_E = \ell \\(i_1 := i_2 :: cmds, E, \pi, \mu) &\longrightarrow (() :: cmds, E, \pi, \mu[\ell \mapsto \llbracket i_2 \rrbracket_E]) && \text{avec } \llbracket i_1 \rrbracket_E = \ell\end{aligned}$$

## Assignations :

$$(x = i :: cmds, E, \pi, \mu) \longrightarrow ( () :: cmds, E[x \mapsto \llbracket i \rrbracket_E], \pi, \mu)$$

$$(x = i_1 \boxplus i_2 :: cmds, E, \pi, \mu) \longrightarrow ( () :: cmds, E[x \mapsto \llbracket i_1 \rrbracket_E \boxplus \llbracket i_2 \rrbracket_E], \pi, \mu)$$

$$(x = \mathbf{ref} \ i :: cmds, E, \pi, \mu) \longrightarrow ( () :: cmds, E[x \mapsto \ell], \pi, \mu[\ell \mapsto \llbracket i \rrbracket_E]) \quad \text{avec } \ell \notin \text{dom } \mu$$

$$(x = !i :: cmds, E, \pi, \mu) \longrightarrow ( () :: cmds, E[x \mapsto \mu(\ell)], \pi, \mu) \quad \text{avec } \llbracket i \rrbracket_E = \ell$$

$$(x = i_1 := i_2 :: cmds, E, \pi, \mu) \longrightarrow ( () :: cmds, E[x \mapsto ()], \pi, \mu[\ell \mapsto \llbracket i_2 \rrbracket_E]) \quad \text{avec } \llbracket i_1 \rrbracket_E = \ell$$

## Évaluation « pas à pas » de la machine abstraite (4)

Appels de fonctions :

Étant donné une déclaration : **fun**  $f(x_1, \dots, x_n) = \{ \text{cmds}' \}$

$$(x = f(i_1, \dots, i_n) :: \text{cmds}, E, \pi, \mu) \longrightarrow (\text{cmds}', E', (x, \text{cmds}, E) : \pi, \mu)$$

$$(f(i_1, \dots, i_n) :: \text{cmds}, E, \pi, \mu) \longrightarrow (\text{cmds}', E', (\_, \text{cmds}, E) : \pi, \mu)$$

$$\text{où } E' = [x_1 \mapsto \llbracket i_1 \rrbracket_E, \dots, x_n \mapsto \llbracket i_n \rrbracket_E]$$

**i** Vocabulaire :

- ▶ Les  $x_k$  sont les **paramètres formels** de  $f$  (*formal parameters*)
- ▶ Les  $i_k$  sont les **paramètres réels** de  $f$  (*actual parameters*)

## Évaluation « pas à pas » de la machine abstraite (4)

Appels de fonctions :

Étant donné une déclaration : **fun**  $f(x_1, \dots, x_n) = \{ \text{cmds}' \}$

$$(x = f(i_1, \dots, i_n) :: \text{cmds}, E, \pi, \mu) \longrightarrow (\text{cmds}', E', (x, \text{cmds}, E) : \pi, \mu)$$

$$(f(i_1, \dots, i_n) :: \text{cmds}, E, \pi, \mu) \longrightarrow (\text{cmds}', E', (\_, \text{cmds}, E) : \pi, \mu)$$

$$\text{où } E' = [x_1 \mapsto \llbracket i_1 \rrbracket_E, \dots, x_n \mapsto \llbracket i_n \rrbracket_E]$$

**i** Vocabulaire :

- ▶ Les  $x_k$  sont les **paramètres formels** de  $f$  (*formal parameters*)
- ▶ Les  $i_k$  sont les **paramètres réels** de  $f$  (*actual parameters*)
- ▶ **Appel par valeur** : les paramètres sont évalués avant l'appel
- ▶ **Passage de paramètres par copie** : les valeurs des paramètres sont copiées pour utilisation par l'appelé



# Évaluation « pas à pas » de la machine abstraite (4)

## Appels de fonctions :

Étant donné une déclaration : **fun**  $f(x_1, \dots, x_n) = \{ \text{cmds}' \}$

$$(x = f(i_1, \dots, i_n) :: \text{cmds}, E, \pi, \mu) \longrightarrow (\text{cmds}', E', (x, \text{cmds}, E) : \pi, \mu)$$

$$(f(i_1, \dots, i_n) :: \text{cmds}, E, \pi, \mu) \longrightarrow (\text{cmds}', E', (\_, \text{cmds}, E) : \pi, \mu)$$

$$\text{où } E' = [x_1 \mapsto \llbracket i_1 \rrbracket_E, \dots, x_n \mapsto \llbracket i_n \rrbracket_E]$$

## **i** Vocabulaire :

- ▶ Les  $x_k$  sont les **paramètres formels** de  $f$  (*formal parameters*)
- ▶ Les  $i_k$  sont les **paramètres réels** de  $f$  (*actual parameters*)
- ▶ **Appel par valeur** : les paramètres sont évalués avant l'appel
- ▶ **Passage de paramètres par copie** : les valeurs des paramètres sont copiées pour utilisation par l'appelé

## Retours de fonctions :

$$(\llbracket v \rrbracket, E, (x, \text{cmds}, E') : \pi, \mu) \longrightarrow ((\_) :: \text{cmds}, E' [x \mapsto v], \pi, \mu)$$

$$(\llbracket v \rrbracket, E, (\_, \text{cmds}, E') : \pi, \mu) \longrightarrow (v :: \text{cmds}, E', \pi, \mu)$$

- ▶ Implémentation de MINI en OCaml : lexer/parser + machine abstraite
- ▶ Code source disponible, bientôt sur **#prog**
- ▶ Programme `swap.mini`  
\$ `dune exec -- ./cli.exe programs/swap.mini --interactive`

- ▶ Implémentation de MINI en OCaml : lexer/parser + machine abstraite
- ▶ Code source disponible, bientôt sur **#prog**
- ▶ Programme `swap.mini`  
\$ dune exec -- ./cli.exe programs/swap.mini --interactive
- ▶ Programme `mc91.mini`  
\$ dune exec -- ./cli.exe programs/mc91.mini --interactive

- ▶ Implémentation de MINI en OCaml : lexer/parser + machine abstraite
- ▶ Code source disponible, bientôt sur `#prog`
- ▶ Programme `swap.mini`  
`$ dune exec -- ./cli.exe programs/swap.mini --interactive`
- ▶ Programme `mc91.mini`  
`$ dune exec -- ./cli.exe programs/mc91.mini --interactive`
- ▶ Exercice pour chez vous :
  - ▶ Quelle serait une spécification pour `mc91`?
  - ▶ Preuve de correction?
  - ▶ Preuve de terminaison?

## Passage de paramètres

---

## Passages de paramètres (1)

En MINI, comme dans les 3 langages au programme (C, OCaml, Python) :

- ▶ Les paramètres des fonctions sont évalués *avant* l'appel de fonction
  - ❗ C'est ce qu'on appelle « l'appel par valeur »
- ▶ Les paramètres des fonctions sont passés par copie
  - ⚠ La copie est superficielle (on ne traverse pas les pointeurs)

## Passages de paramètres (1)

En MINI, comme dans les 3 langages au programme (C, OCaml, Python) :

- ▶ Les paramètres des fonctions sont évalués *avant* l'appel de fonction
  - ❗ C'est ce qu'on appelle « l'appel par valeur »
- ▶ Les paramètres des fonctions sont passés par copie
  - ⚠ La copie est superficielle (on ne traverse pas les pointeurs)

En C :

- ▶ Si vous passez des **struct** en paramètres ils seront *copiés*  
Cela peut coûter cher si le **struct** est de grande taille  
Le style idiomatique en C est de passer un *pointeur* vers un **struct**

## Passages de paramètres (1)

En MINI, comme dans les 3 langages au programme (C, OCaml, Python) :

- ▶ Les paramètres des fonctions sont évalués *avant* l'appel de fonction
  - ❗ C'est ce qu'on appelle « l'appel par valeur »
- ▶ Les paramètres des fonctions sont passés par copie
  - ⚠ La copie est superficielle (on ne traverse pas les pointeurs)

En C :

- ▶ Si vous passez des **struct** en paramètres ils seront *copiés*  
Cela peut coûter cher si le **struct** est de grande taille  
Le style idiomatique en C est de passer un *pointeur* vers un **struct**
- ▶ Les pointeurs vers **struct** sont tellement courants en C qu'une syntaxe spéciale est introduite :  
pour **struct** { **int** f } \*p, on peut écrire p->f à la place de (\*p).f



# Passages de paramètres (1)

En MINI, comme dans les 3 langages au programme (C, OCaml, Python) :

- ▶ Les paramètres des fonctions sont évalués *avant* l'appel de fonction
  - ❗ C'est ce qu'on appelle « l'appel par valeur »
- ▶ Les paramètres des fonctions sont passés par copie
  - ⚠ La copie est superficielle (on ne traverse pas les pointeurs)

En C :

- ▶ Si vous passez des **struct** en paramètres ils seront *copiés*  
Cela peut coûter cher si le **struct** est de grande taille  
Le style idiomatique en C est de passer un *pointeur* vers un **struct**
- ▶ Les pointeurs vers **struct** sont tellement courants en C qu'une syntaxe spéciale est introduite :  
pour **struct** { **int** f } \*p, on peut écrire p->f à la place de (\*p).f
- ❓ Que fait ce code ?

```
void swap(int a, int b) {  
    int tmp = b; b = a; a = tmp;  
}
```

## Passages de paramètres (2)

En Python :

- ▶ Ici encore : passage par copie
- ▶ Un objet est une « référence » : le « contenu » de l'objet n'est pas copié

```
1 class Stack:
2     def __init__(self):
3         self.value = []
4
5     def push(self, v):
6         self.value.append(v)
7
8     def get(self):
9         return self.value
10
11
12 def push_twice(stack, v):
13     stack.push(v)
14     stack.push(v)
15
16
17 s = Stack()
18 s.push(42)
19 push_twice(s, 0)
20 print(s.get())
21 # quelle liste est affichée ?
```

## Passages de paramètres (3)

En OCaml :

- ▶ Ici encore, passage par copie
- ▶ En pratique, le compilateur utilise une représentation uniforme des données : une valeur est représentée soit par un entier machine soit par un pointeur vers un bloc mémoire (vers le tas)

```
1  type stack = {
2     value: int list ref
3 }
4
5  let init () = { value = ref [] }
6
7  let get s = !(s.value)
8
9  let push stack v =
10     stack.value :=
11     v :: !(stack.value)
12
13  let push_twice stack v =
14     push stack v;
15     push stack v
16
17  let l =
18     let s = init () in
19     push s 42;
20     push_twice s 0;
21     get s
22     (* quel est le résultat ? *)
```

## MINI : une simplification des machines réelles

---

## Code

- ▶ MINI : liste de commandes
- ▶ Machine réelle :
  - ▶ un pointeur vers l'instruction courante
  - ▶ code enregistré en mémoire, dans le segment `text` (*read-only*)
  - ▶ le pointeur de code est enregistré dans un *registre* (`pc`, ou `ip` suivant les architectures)

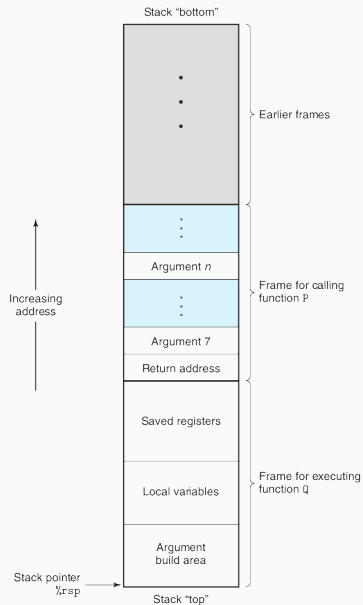
### Variables locales :

- ▶ En MINI : environnement
- ▶ Machine réelle :
  - ▶ Données en registres
  - ▶ Données enregistrées dans la trame de pile courante (si besoin)  
(c'est la trame de pile située au sommet de la pile)


# Comparaison d'un état de MINI avec une machine réelle (3)

## Pile :

- ▶ MINI : liste de trames contenant variable de retour, environnement à restaurer, et code à restaurer
- ▶ Machine réelle :
  - ▶ La liste de trames est en mémoire : les trames les unes à côté des autres
  - ▶ Chaque trame contient :
    - ▶ les variables et données locales
    - ▶ les registres à restaurer
    - ▶ les arguments passés à la fonction courante
    - ▶ le pointeur de code à restaurer
    - ▶ et parfois : un « *canari* » (contre-mesure de sécurité)
  - ▶ Deux registres délimitent la trame courante (frame pointer et stack pointer)
  - ▶ Le contenu exact dépend de la *convention d'appel* de l'architecture, et du compilateur utilisé




### Tas :

- ▶ MINI : mémoire (map adresses → valeurs)
- ▶ Machine réelle :
  - ▶ Les adresses sont des entiers
  - ▶ Valeurs enregistrées en mémoire sous forme binaire
  - ▶ Tas et pile sont dans la même mémoireAdresses basses : tas                      Adresses hautes : pile
- ▶  Il faut se prémunir contre les chevauchements entre tas et pile
- ▶ En général : l'OS préalloue l'espace pour la pile, et ajoute une « page de garde » juste avant



# Comparaison d'un état de MINI avec une machine réelle (4)

## Tas :

- ▶ MINI : mémoire (map adresses → valeurs)
- ▶ Machine réelle :
  - ▶ Les adresses sont des entiers
  - ▶ Valeurs enregistrées en mémoire sous forme binaire
  - ▶ Tas et pile sont dans la même mémoireAdresses basses : tas                      Adresses hautes : pile
- ▶  Il faut se prémunir contre les chevauchements entre tas et pile
- ▶ En général : l'OS préalloue l'espace pour la pile, et ajoute une « page de garde » juste avant

## Point culturel : « *capability machines* »

- ▶ pointeur  $\neq$  entier
- ▶ Impossibilité de « forger » des pointeurs
- ▶ Encore un sujet de recherche actif
- ▶ Architecture bientôt disponible ? (exemple : ARM-Morello/CHERI)

## Pile : quelques informations supplémentaires

- ▶ Les constituants de la pile sont appelés :  
« stack frame » ou « activation record » ou « trame de pile »

## Pile : quelques informations supplémentaires

- ▶ Les constituants de la pile sont appelés :  
« stack frame » ou « activation record » ou « trame de pile »
- ▶ La trame la plus récente est appelée trame « courante »
  - ▶ Elle est au sommet de la pile
  - ▶ Elle est utilisée par la fonction qui est *en train* de s'exécuter

## Pile : quelques informations supplémentaires

- ▶ Les constituants de la pile sont appelés :  
« stack frame » ou « activation record » ou « trame de pile »
- ▶ La trame la plus récente est appelée trame « courante »
  - ▶ Elle est au sommet de la pile
  - ▶ Elle est utilisée par la fonction qui est *en train* de s'exécuter
- ▶ Dans beaucoup de langages de programmation :
  - ▶ La fonction courante n'a accès qu'à la trame courante
  - ▶ Elle ne peut ni lire ni écrire dans les trames parentes

## Pile : quelques informations supplémentaires

- ▶ Les constituants de la pile sont appelés :  
« stack frame » ou « activation record » ou « trame de pile »
- ▶ La trame la plus récente est appelée trame « courante »
  - ▶ Elle est au sommet de la pile
  - ▶ Elle est utilisée par la fonction qui est *en train* de s'exécuter
- ▶ Dans beaucoup de langages de programmation :
  - ▶ La fonction courante n'a accès qu'à la trame courante
  - ▶ Elle ne peut ni lire ni écrire dans les trames parentes
- ▶ En C : c'est possible, via un pointeur vers la pile parente

```
1 void f(int *p) {  
2     *p = 0;  
3 }  
4  
5 int main() {  
6     int x = 42; // x est alloué dans la pile de `main`  
7     f(&x);     // puis modifié par l'exécution de `f`  
8     return x;  
9 }
```

# Allocation de mémoire

---

## Allocation : pile ou tas ?

Que contiennent pile et tas, pour les langages de programmation au programme ?

---

	Pile <sup>1</sup>	Tas
MINI	variable locales, paramètres	références

---

- 
1. Ou dans les registres, si la place le permet
  - 2.
  - 3.

## Allocation : pile ou tas ?

Que contiennent pile et tas, pour les langages de programmation au programme ?

---

	Pile <sup>1</sup>	Tas
MINI	variable locales, paramètres	références
Python	variables locales, paramètres <sup>2</sup>	strings, objets, listes...

---

1. Ou dans les registres, si la place le permet

2. Généralement : ce qui peut tenir dans un registre (entiers, booléens, pointeurs...)

3.



## Allocation : pile ou tas ?

Que contiennent pile et tas, pour les langages de programmation au programme ?

	Pile <sup>1</sup>	Tas
MINI	variable locales, paramètres	références
Python	variables locales, paramètres <sup>2</sup>	strings, objets, listes...
OCaml	variables locales, paramètres <sup>2</sup>	strings, clôtures, listes, objets, enregistrements, références...

1. Ou dans les registres, si la place le permet

2. Généralement : ce qui peut tenir dans un registre (entiers, booléens, pointeurs...)

3.

## Allocation : pile ou tas ?

Que contiennent pile et tas, pour les langages de programmation au programme ?

	Pile <sup>1</sup>	Tas
MINI	variable locales, paramètres	références
Python	variables locales, paramètres <sup>2</sup>	strings, objets, listes...
OCaml	variables locales, paramètres <sup>2</sup>	strings, clôtures, listes, objets, enregistrements, références...
C	variables locales, paramètres <sup>3</sup>	tout ce qui est créé par des appels à <code>malloc/calloc</code>

1. Ou dans les registres, si la place le permet

2. Généralement : ce qui peut tenir dans un registre (entiers, booléens, pointeurs...)

3. Tout ce qui est déclaré comme variable locale (y compris **struct**, **union**, tableaux...)

# Allocation : pile ou tas ?

Que contiennent pile et tas, pour les langages de programmation au programme ?

	Pile <sup>1</sup>	Tas
MINI	variable locales, paramètres	références
Python	variables locales, paramètres <sup>2</sup>	strings, objets, listes...
OCaml	variables locales, paramètres <sup>2</sup>	strings, clôtures, listes, objets, enregistrements, références...
C	variables locales, paramètres <sup>3</sup>	tout ce qui est créé par des appels à <code>malloc/calloc</code>

Et **toujours**, dans la pile : infos sur quoi faire au retour de la fonction courante (`pc/sp` de l'appelant, adresse pour la valeur à retourner, registres à restaurer...)

1. Ou dans les registres, si la place le permet

2. Généralement : ce qui peut tenir dans un registre (entiers, booléens, pointeurs...)

3. Tout ce qui est déclaré comme variable locale (y compris **struct**, **union**, tableaux...)

## Exemples

En C :

```
struct pair {
    int fst;
    int snd;
};

int f() {
    struct pair p;
    // le contenu de `p` est
    // alloué en pile
    p.fst = 17;
    p.snd = 25;
    int sum = p.fst + p.snd;
    // l'entier `sum` est
    // alloué en pile
    return sum;
}
```

(On suppose ici qu'aucune optimisation n'est effectuée)

## Exemples

En C :

```
struct pair {
    int fst;
    int snd;
};

int f() {
    struct pair p;
    // le contenu de `p` est
    // alloué en pile
    p.fst = 17;
    p.snd = 25;
    int sum = p.fst + p.snd;
    // l'entier `sum` est
    // alloué en pile
    return sum;
}
```

En OCaml :

```
type pair = {
    fst: int;
    snd: int;
}

let f () =
    let p =
        (* le contenu de `p` est
        alloué dans le tas *)
        { fst = 17;
          snd = 25 } in
    let sum = p.fst + p.snd in
    (* l'entier `sum` est
    alloué en pile *)
    sum
```

(On suppose ici qu'aucune optimisation n'est effectuée)



## Allocation : pile ou tas? Avantages et inconvénients

- ▶ En Python et OCaml, vous n'avez pas le choix  
La mémoire (allocation/libération) est gérée automatiquement
- ▶ En C, vous pouvez choisir
  - ▶ Allocation en pile : par déclaration de variables locales
    - ❗ Aussi appelée « allocation statique »
    - ⊕ Libération automatique (au retour des fonctions)
    - ⊕ Allocation/libération : peu coûteux en temps (il suffit de modifier `sp`)
    - ⊖ La taille de la pile est limitée (et **beaucoup** plus petite que le tas<sup>1</sup>)
    - ⊖ Tous les motifs de programmation ne se prêtent pas à une discipline d'allocation/libération qui soit *bien parenthésée*
  - ▶ Allocation dans le tas : par appels à `malloc/calloc`

---

1. Taille maximale de `stack` sous Linux : `ulimit -s`

Sur ma machine : 8192 kbytes

# Allocation : pile ou tas? Avantages et inconvénients

- ▶ En Python et OCaml, vous n'avez pas le choix  
La mémoire (allocation/libération) est gérée automatiquement
- ▶ En C, vous pouvez choisir
  - ▶ Allocation en pile : par déclaration de variables locales
    - ❗ Aussi appelée « allocation statique »
    - ⊕ Libération automatique (au retour des fonctions)
    - ⊕ Allocation/libération : peu coûteux en temps (il suffit de modifier `sp`)
    - ⊖ La taille de la pile est limitée (et **beaucoup** plus petite que le tas<sup>1</sup>)
    - ⊖ Tous les motifs de programmation ne se prêtent pas à une discipline d'allocation/libération qui soit *bien parenthésée*
  - ▶ Allocation dans le tas : par appels à `malloc/calloc`
    - ❗ Aussi appelée « allocation dynamique »
    - ⊕ La taille du tas est généralement grande<sup>2</sup>
    - ⊕ Offre plus de liberté dans le style de programmation
    - ⊖ Libération explicite de la mémoire (**free**) : difficile, source d'erreurs
    - ⊖ Allocation/libération : plus coûteux en temps (utilisation d'un allocateur mémoire)

---

1. Taille maximale de `stack` sous Linux : `ulimit -s`

Sur ma machine : 8192 kbytes

2. Taille maximale de `data` sous Linux : `ulimit -d`

Sur ma machine : « unlimited »



## Appels terminaux et consommation de pile

---

## Appels terminaux

Exemples en MINI :

```
fun g(z) = { z * 2 }
```

```
fun f(x) = { y = x + 1; g(y) (* appel terminal *) }
```

```
fun main() = { f(20) (* un autre appel terminal *) }
```

Exemples en MINI :

```
fun g(z) = { z * 2 }
```

```
fun f(x) = { y = x + 1; g(y) (* appel terminal *) }
```

```
fun main() = { f(20) (* un autre appel terminal *) }
```

```
$ # Exécutons pas à pas ce programme, avec la commande suivante:
```

```
$ dune exec -- ./cli.exe programs/tailcalls.mini --interactive
```

```
...
```

```
Main function: main
```

```
Result: 42
```

```
Maximum stack size: 3
```

```
Number of evaluation steps: 9
```

On ajoute un cas particulier à la relation de réduction lorsque l'appel de fonction est la « dernière chose à faire pour la trame courante »

Étant donné une déclaration : **fun**  $f(x_1, \dots, x_n) = \{ \text{cmds}' \}$

$$(\llbracket f(i_1, \dots, i_n) \rrbracket, E, \pi, \mu) \longrightarrow (\text{cmds}', E', \pi, \mu)$$

où  $E' = [x_1 \mapsto \llbracket i_1 \rrbracket_E, \dots, x_n \mapsto \llbracket i_n \rrbracket_E]$

Remarque : dans la commande  $x = f(i_1, \dots, i_n)$ , l'appel à  $f$  n'est pas terminal

## Appels terminaux, avec machine abstraite optimisée

Les mêmes exemples en MINI :

```
fun g(z) = { z * 2 }
```

```
fun f(x) = { y = x + 1; g(y) (* appel terminal *) }
```

```
fun main() = { f(20) (* un autre appel terminal *) }
```

## Appels terminaux, avec machine abstraite optimisée

Les mêmes exemples en MINI :

```
fun g(z) = { z * 2 }
```

```
fun f(x) = { y = x + 1; g(y) (* appel terminal *) }
```

```
fun main() = { f(20) (* un autre appel terminal *) }
```

```
$ # Exécutons ce programme avec l'optimisation d'appels terminaux:
```

```
$ dune exec -- ./cli.exe programs/tailcalls.mini --tailcalls
```

```
...
```

```
Main function: main
```

```
Result: 42
```

```
Maximum stack size: 0
```

```
Number of evaluation steps: 6
```

👍 Cette fois, aucun espace de pile n'est consommé

📌 Cela fonctionne aussi avec des fonctions récursives!

## Appels terminaux en Python

En Python, les appels terminaux ne sont pas optimisés : ils consomment de la pile

```
>>> def loop(x, acc):
...     if (x <= 0):
...         return acc
...     return loop(x-1, acc+1)
... 
```

```
>>> loop(997, 0)
```

```
997
```

```
>>> loop(998, 0)
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
File "<stdin>", line 4, in loop
```

```
File "<stdin>", line 4, in loop
```

```
File "<stdin>", line 4, in loop
```

```
[Previous line repeated 995 more times]
```

```
File "<stdin>", line 2, in loop
```

```
RecursionError: maximum recursion depth exceeded in comparison
```

En OCaml, les appels terminaux sont optimisés presque tout le temps

X. Leroy ([discuss.ocaml.org](https://discuss.ocaml.org), Octobre 2021) :

“  
*If there are many arguments to the call and they don't all fit in the processor registers reserved for argument passing, the remaining arguments are put on the stack, and a regular, non-tail call is performed. This limitation had been with us since day 1 of OCaml.*  
”



En OCaml, les appels terminaux sont optimisés presque tout le temps

X. Leroy ([discuss.ocaml.org](https://discuss.ocaml.org), Octobre 2021) :

“  
*If there are many arguments to the call and they don't all fit in the processor registers reserved for argument passing, the remaining arguments are put on the stack, and a regular, non-tail call is performed. This limitation had been with us since day 1 of OCaml.*

Dans le même message, décrivant une correction pour la version 4.14 d'OCaml :

“  
*Once the registers available for passing function arguments are exhausted, the next 64 arguments are passed in a memory area that is part of the domain state. This argument passing is compatible with tail calls, so we get guaranteed tail calls up to 70 arguments at least. [...] Enjoy your 70-arguments tail calls!*

## Appels terminaux en OCaml

```
(* Factorielle: version réursive, non terminale *)
```

```
let rec fact n =  
  if n <= 0  
  then 1  
  else n * fact (n-1)
```

```
(* Factorielle: version réursive, non terminale *)
```

```
let fact_tailrec =  
  let rec aux acc n = (* `acc' est un accumulateur *)  
    if n <= 0  
    then acc  
    else aux (n * acc) (n - 1)  
  in  
  aux 1
```

Exercice : démontrer que `fact` et `fact_tailrec` calculent les mêmes résultats

- ❓ Quel lemme auxiliaire est nécessaire ?
- ❓ Quelles propriétés de la multiplication sont nécessaires ?

## Appels terminaux en C

En C, les appels terminaux *peuvent* être optimisés : cela dépend du compilateur, et du niveau d'optimisation choisi

```
#include <stdio.h>
```

```
// When compiled with -O2, tail-call optimization is performed
```

```
int loop(int x, int acc) {  
    if (x <= 0) { return acc; }  
    else { return loop(x-1, acc+1); }  
}
```

```
int main() {  
    int result = loop(5000000, 0);  
    printf("result = %d\n", result);  
    return 0;  
}
```

Pour en être certain, mieux vaut regarder le code assembleur produit!

```
$ gcc -std=c99 -S -O2 tailcall.c -o tailcall.S
```

Comment déterminer la consommation de pile asymptotique dans le pire cas ?

- ▶ Python : sémantique peu claire pour ce genre d'analyse...  
De plus : la pile est celle de l'interpréteur, pas celle de la machine  
(le code est interprété : pas compilé)

Comment déterminer la consommation de pile asymptotique dans le pire cas ?

- ▶ Python : sémantique peu claire pour ce genre d'analyse...

De plus : la pile est celle de l'interpréteur, pas celle de la machine  
(le code est interprété : pas compilé)

- ▶ OCaml :

- ▶ En 1<sup>re</sup> approximation : la taille des trames est proportionnelle au nombre de variables locales d'une fonction
- ▶ Elle dépend du code et du compilateur
- ▶ Elle ne dépend pas des valeurs des arguments des fonctions
- ▶ Il suffit donc de compter le nombre d'appels de fonction qui sont imbriqués et non-terminaux pour avoir une approximation en  $O(\cdot)$

# Éléments pour raisonner sur la consommation de pile : en Python/OCaml

Comment déterminer la consommation de pile asymptotique dans le pire cas ?

- ▶ Python : sémantique peu claire pour ce genre d'analyse...  
De plus : la pile est celle de l'interpréteur, pas celle de la machine (le code est interprété : pas compilé)
- ▶ OCaml :
  - ▶ En 1<sup>re</sup> approximation : la taille des trames est proportionnelle au nombre de variables locales d'une fonction
  - ▶ Elle dépend du code et du compilateur
  - ▶ Elle ne dépend pas des valeurs des arguments des fonctions
  - ▶ Il suffit donc de compter le nombre d'appels de fonction qui sont imbriqués et non-terminaux pour avoir une approximation en  $O(\cdot)$
- ▶ Si on insiste pour Python :
  - ▶ Faire comme en OCaml, en comptant cette fois les appels terminaux

# Éléments pour raisonner sur la consommation de pile : en Python/OCaml

Comment déterminer la consommation de pile asymptotique dans le pire cas ?

- ▶ Python : sémantique peu claire pour ce genre d'analyse...  
De plus : la pile est celle de l'interpréteur, pas celle de la machine (le code est interprété : pas compilé)
- ▶ OCaml :
  - ▶ En 1<sup>re</sup> approximation : la taille des trames est proportionnelle au nombre de variables locales d'une fonction
  - ▶ Elle dépend du code et du compilateur
  - ▶ Elle ne dépend pas des valeurs des arguments des fonctions
  - ▶ Il suffit donc de compter le nombre d'appels de fonction qui sont imbriqués et non-terminaux pour avoir une approximation en  $O(\cdot)$
- ▶ Si on insiste pour Python :
  - ▶ Faire comme en OCaml, en comptant cette fois les appels terminaux
- ▶ Dans tous les cas : si le sujet ne le précise pas, **rappelez explicitement ce que vous comptez**

## Éléments pour raisonner sur la consommation de pile : en C

- ▶ **Problème** : à cause de l'allocation dans la pile, la taille des trames de pile peut dépendre des valeurs des arguments!

```
int f(int n) {  
    int t[n];    // un tableau de taille `n' est alloué en pile  
    ...        // `t' est un VLA: Variable-Length Array  
}
```



## Éléments pour raisonner sur la consommation de pile : en C

- ▶ **Problème** : à cause de l'allocation dans la pile, la taille des trames de pile peut dépendre des valeurs des arguments!

```
int f(int n) {  
    int t[n];    // un tableau de taille `n' est alloué en pile  
    ...        // `t' est un VLA: Variable-Length Array  
}
```

- ❗ Le support des VLAs est optionnel en C11

## Éléments pour raisonner sur la consommation de pile : en C

- ▶ **Problème** : à cause de l'allocation dans la pile, la taille des trames de pile peut dépendre des valeurs des arguments!

```
int f(int n) {  
    int t[n];    // un tableau de taille `n' est alloué en pile  
    ...        // `t' est un VLA: Variable-Length Array  
}
```

- ❗ Le support des VLAs est optionnel en C11
- ▶ Si le code n'utilise pas ce motif, alors la taille des frames ne dépend pas des arguments, et il suffit de compter les appels de fonctions imbriqués

## Éléments pour raisonner sur la consommation de pile : en C

- ▶ **Problème** : à cause de l'allocation dans la pile, la taille des trames de pile peut dépendre des valeurs des arguments!

```
int f(int n) {  
    int t[n];    // un tableau de taille `n' est alloué en pile  
    ...        // `t' est un VLA: Variable-Length Array  
}
```

- ❗ Le support des VLAs est optionnel en C11
- ▶ Si le code n'utilise pas ce motif, alors la taille des frames ne dépend pas des arguments, et il suffit de compter les appels de fonctions imbriqués
- ▶ Mon conseil : *considérez* par défaut que les appels terminaux ne sont *pas* optimisés en C

## Éléments pour raisonner sur la consommation de pile : en C

- ▶ **Problème** : à cause de l'allocation dans la pile, la taille des trames de pile peut dépendre des valeurs des arguments!

```
int f(int n) {  
    int t[n];    // un tableau de taille `n` est alloué en pile  
    ...        // `t` est un VLA: Variable-Length Array  
}
```

- ❗ Le support des VLAs est optionnel en C11
- ▶ Si le code n'utilise pas ce motif, alors la taille des frames ne dépend pas des arguments, et il suffit de compter les appels de fonctions imbriqués
- ▶ Mon conseil : *considérez* par défaut que les appels terminaux ne sont *pas* optimisés en C
- ▶ Dans tous les cas : si le sujet ne le précise pas, **rappelez explicitement ce que vous comptez**

- ▶ Python :
  - ▶ Taille des strings créées
  - ▶ Tuples, listes, ensembles, objets
  - ⚠ La taille des listes/tableaux/strings peut dépendre de la valeur des paramètres
  - ⚠ Taille des entiers calculés (ils sont en précision arbitraire!)

## ▶ Python :

- ▶ Taille des strings créées

- ▶ Tuples, listes, ensembles, objets

- ⚠ La taille des listes/tableaux/strings peut dépendre de la valeur des paramètres

- ⚠ Taille des entiers calculés (ils sont en précision arbitraire!)

## ▶ OCaml :

- ▶ Taille des strings créées

- ▶ Tuples, variants (constructeurs), records, objets, clôtures, **lazy**

- ⚠ La taille des listes/tableaux/strings peut dépendre de la valeur des paramètres

- ▶ Python :
  - ▶ Taille des strings créées
  - ▶ Tuples, listes, ensembles, objets
  - ⚠ La taille des listes/tableaux/strings peut dépendre de la valeur des paramètres
  - ⚠ Taille des entiers calculés (ils sont en précision arbitraire!)
- ▶ OCaml :
  - ▶ Taille des strings créées
  - ▶ Tuples, variants (constructeurs), records, objets, clôtures, **lazy**
  - ⚠ La taille des listes/tableaux/strings peut dépendre de la valeur des paramètres
- ▶ C :
  - ▶ Appels à `malloc/calloc`
  - ⚠ L'argument peut dépendre de la valeur des paramètres

# Conclusion

---





Randal E. BRYANT et David R. O'HALLARON (fév. 2018).  
Computer Systems: A Programmer's Perspective, Global Edition. Pear-  
son. 1120 p. ISBN : 1292101768

## Chapitre 3 : Machine-Level Representation of Programs

- ▶ Section 7 (Procedures) : explications détaillées de gestion de la pile pour architecture **x86-64**
- ▶ Autres sections : beaucoup d'autres explications sur comment représenter des types et des structures de contrôle de C en machine



Michael SCOTT (déc. 2015). Programming Language Pragmatics. Fourth Edition. San Francisco, CA : Morgan Kaufmann Pub. 992 p. ISBN : 9780124104099

### Chapitre 8 : Subroutines and Control Abstraction

- ▶ Autres modes de passage de paramètres, pour d'autres langages
- ▶ Éléments de réponse pour implémenter les exceptions, les coroutines



Sylvain CONCHON et Jean-Christophe FILLIÂTRE (sept. 2014).  
Apprendre à programmer avec Ocaml: Algorithmes et structures de données.  
ADIZES INST. 444 p. ISBN : 2212136781

### Chapitre 3 : Approfondir les concepts d'OCaml

- ▶ Section 2 (Modèle d'exécution) : en particulier, représentation des données (représentation uniforme)

- ▶ Un modèle d'exécution simple pour un petit langage (MINI)
  - ▶ Une pile pour gérer les appels de fonction
  - ▶ Un tas pour enregistrer les valeurs des références
- ▶ Lien entre ce modèle et une machine réelle
- ▶ Utilisation de la pile et du tas pour les langages au programme
- ▶ Quelques remarques pour raisonner sur la consommation de mémoire
- ▶ Retrouvez le code de l'interpréteur MINI sur **Discord**

Cet après-midi :

Programmation fonctionnelle