Implémentation d'un Langage de Programmation

INF 441 : Programmation Avancée

Xavier Rival

15 mai 2017

Quelques éléments administratifs

Tutorat :

mardi 30 mai, 17h45-19h15, salle 30, confirmé

- format similaire à la séance qui a eu lieu le 16 mai...
- merci de me dire si vous êtes intéressés

Projets :

deadline le 31 mai (ça approche...)

- soumettre par mail un archive (`.zip`, `.tgz`...)
- doit contenir un fichier `readme.txt`
  liste des sources, procédure de compilation / exécution, jeux de tests, et toutes les explications nécessaires...
- le 31 mai AoE (Anywhere on Earth) :-)
- sountenance orale : Catherine Bensoussan va vous contacter ;
  lui fournir vos disponibilités rapidement
- format : 15 min de présentation / démo + 10 min de questions

Cours précédents

Nous avons étudié plusieurs types de notions.

Éléments d'un langage de programmation :

- **typage** et **garanties statiques** (absence d'erreurs. . .)
- **fonctions**, ordre d'évaluation, coût (pile)
- **modules** et unités de compilation
- **classes**, **objets** (héritage, liaison dynamique)

Techniques de programmation :

- **structuration** du code pour une meilleure lisibilité et confiance dans son fonctionnement (fonctions, modules, foncteurs)
- **itération** en respectant un certain niveau d'abstraction (itérateurs, réducteurs, flots. . .)
- et un peu de **programmation concurrente** via les threads légers

Quelques questions d'implémentation

Comment se comporte le compilateur `ocamlc` ?
Comment se comporte l'interprète `ocaml` ?

- nature des **erreurs**
- signification des **messages** (warnings, informations supplémentaires)
- nature du **code produit**

Le code généré est il efficace ?

- combien de **mémoire** utilise-t'il ?
- peut-on **récupérer cette mémoire plus tôt**, et si oui, comment ?
- comment s'exécute le **code généré** par le compilateur / interprète ?

Pour répondre à ces questions, il nous faut **regarder du côté de l'implémentation du langage**...

Contenu du cours

Trois grandes parties :

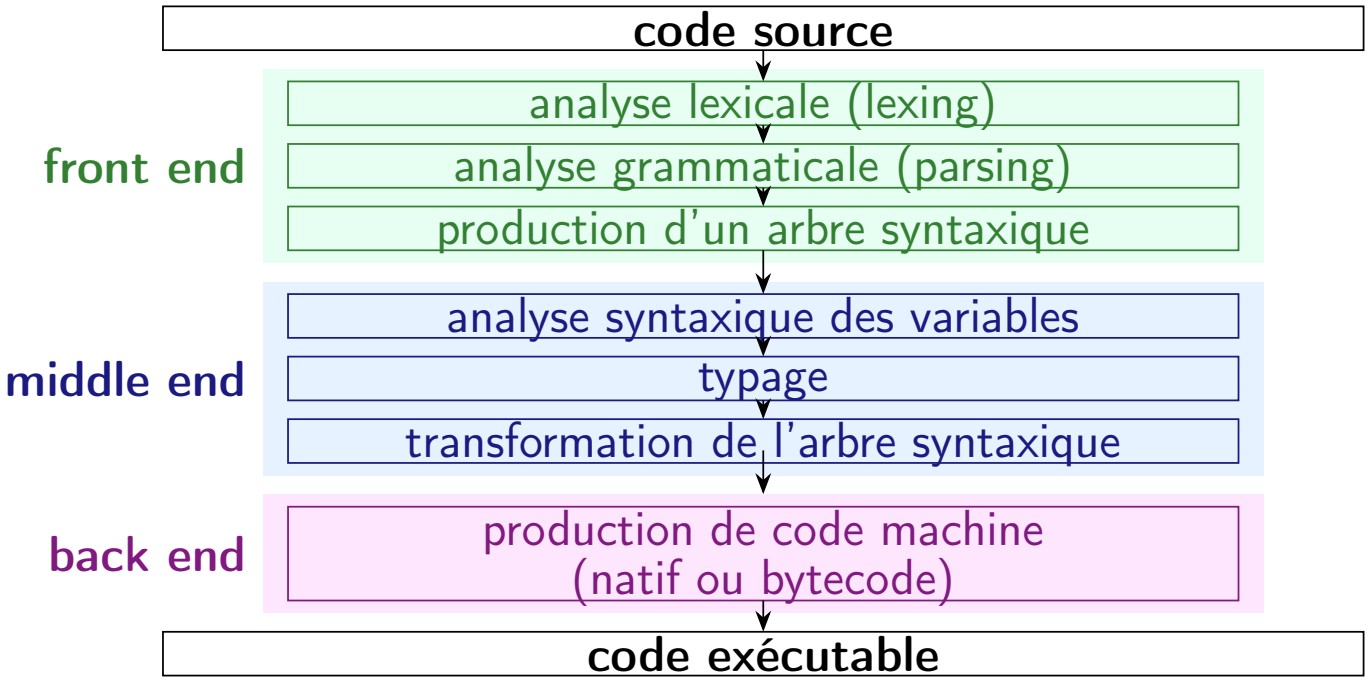
1. **le fonctionnement du compilateur** :
   phases, principes sous-jacents, méthodes utilisées
2. **la gestion mémoire** :
   représentation des données, allocation manuelle / automatique, GC
3. **génération de code** :
   machine abstraite, programmes traduits et exécution

En même temps, nous allons **réviser certains points**
(évaluation, récursivité terminale, données persistantes / mutables. . .)
et nous allons **mieux comprendre comment bien programmer**

Outline

Structure générale d'un compilateur

Un compilateur est généralement **découpé** en une **suite** de **phases distinctes** :



- la limite entre **front-end** et **middle-end** est un peu arbitraire
- un interprète a une structure assez similaire

Analyse lexicale (Lexing)

Première phase :

Analyse lexicale

Lecture du fichier source (texte) et **traduction en suite de symbole**.
Un **symbole** : mot clé, un identificateur, un opérateur, une parenthèse...

Exemple :

```
let x = 7 in x * (x + fact 2)
```

produit (mots clés en rouge, identificateurs en bleu, symboles en violet, constantes en vert) :

- lecture d'un **flot de caractères**, production d'un **flot de symboles**
- utilise des **automates finis** (peut être vus dans un autre cours)
- en cas d'**échec** : `Illegal character`, `Syntax error`

Analyse grammaticale (Parsing)

Seconde phase :

Analyse grammaticale

Reconnaissance d'une suite de symboles à l'aide d'une **grammaire**, et **production d'un arbre syntaxique abstrait**.

Nous avons déjà présenté des **grammaires** au cours de l'amphi 2 :

```
e  ::=  0  |  1  |  ...
     |  x                          (où x est un identificateur)
     |  e + e
     |  if e then e else e
     |  let x = e in e
     |  ...
```

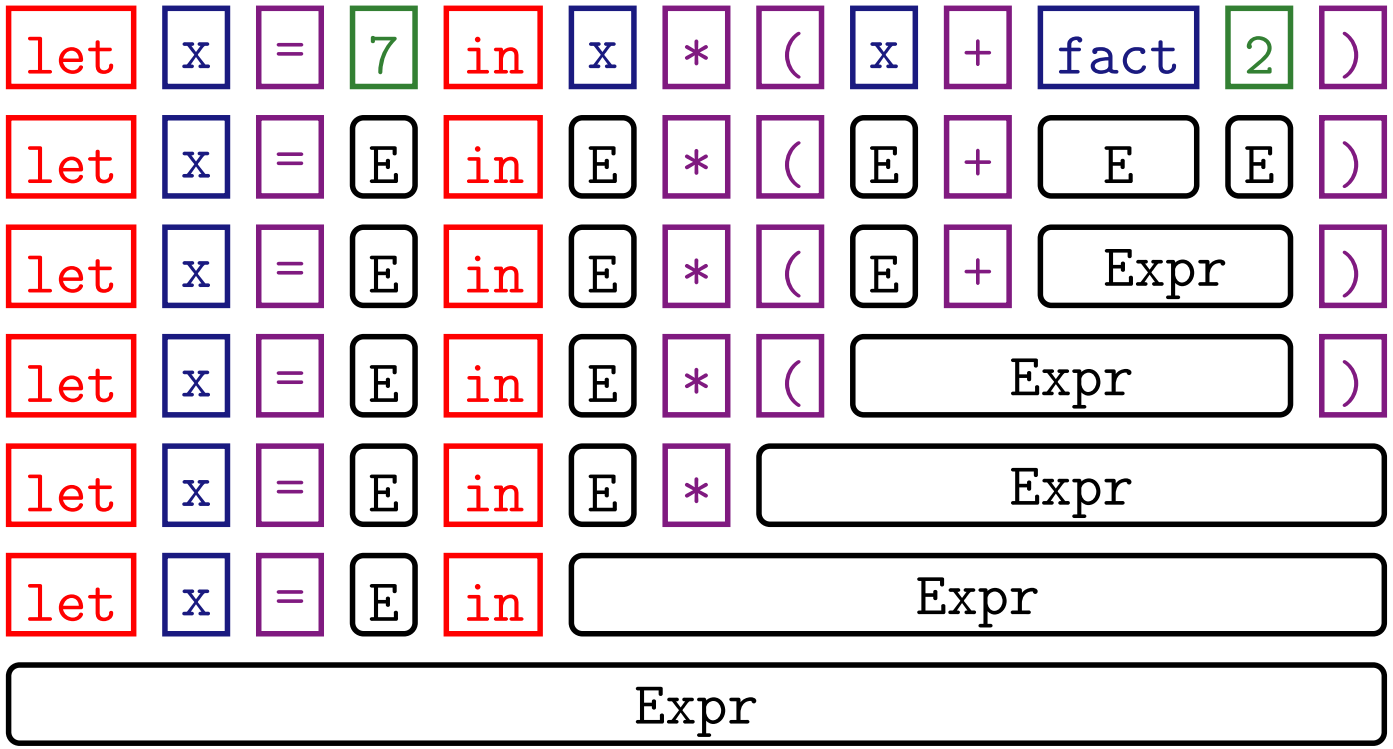
- utilise un **automate** à pile
  i.e., un automate qui conserve un état interne
- en cas d'**échec** : `Syntax error...`

Analyse grammaticale : exemple

Revenons sur **notre exemple** :

```
let x = 7 in x * (x + fact 2)
```

Étapes de l'analyse grammaticale, avec les blocs reconnus comme expressions décrits par les blocs noirs :

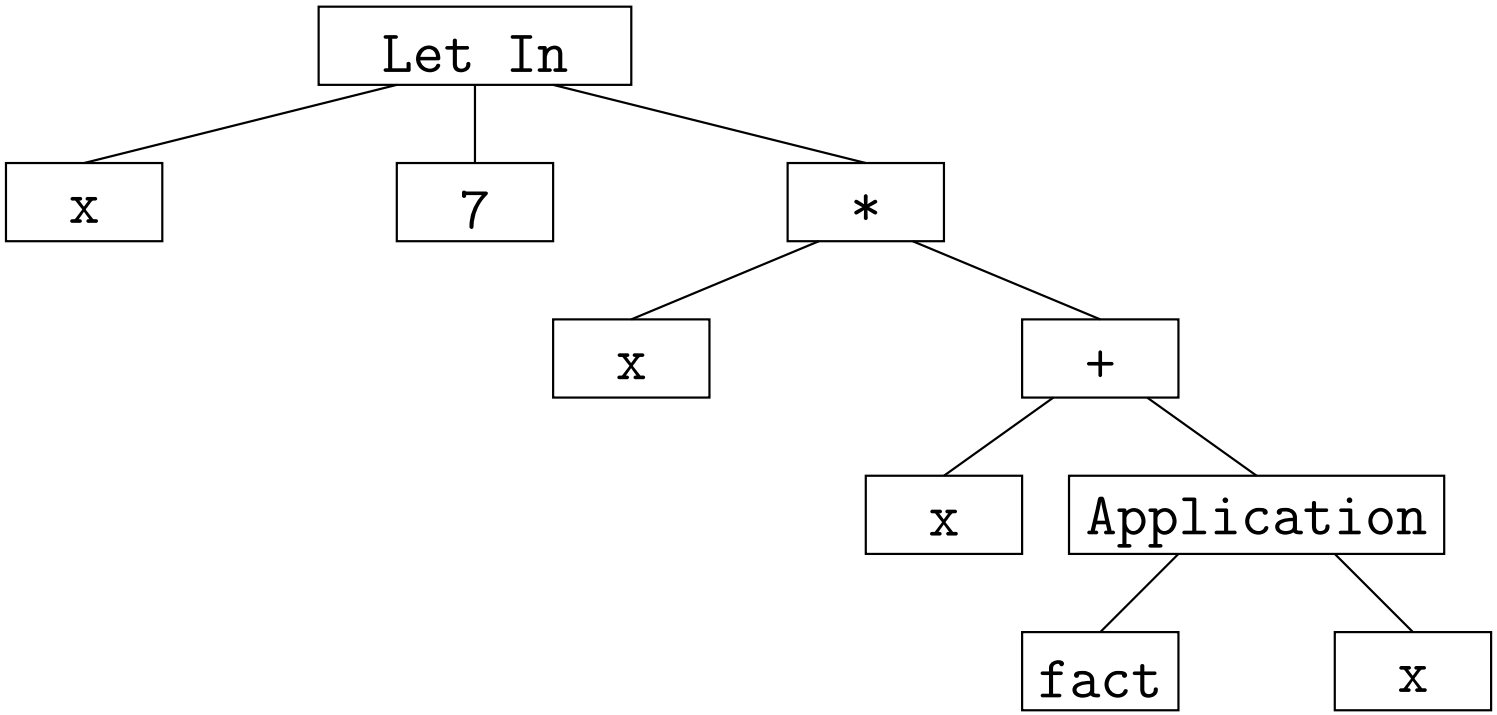


Analyse grammaticale : exemple

Revenons sur **notre exemple** :

```
let x = 7 in x * (x + fact 2)
```

Résultat de l'analyse grammaticale : **un arbre syntaxique**, représenté par un **type somme récursif**



Résolution des variables

Troisième phase :

Analyse syntaxique des variables

La **résolution des variables** a pour but de s'assurer que toute occurrence de variable est **valide**, du point de vue de son **domaine de visibilité**.

Exemple :

```
let x = 7 in x * (x + fact 2)
```

- si `fact` a été déclaré avant, l'expression est **acceptée**
- sinon, le programme est **rejeté** :

```
Error: Unbound value fact
```

- cette phase consiste en un **parcours d'arbre** (cf TD 5)

Inférence de types

Quatrième phase :

Inférence et vérification de types (amphi 2)

La **vérification de types** s'assure que chaque composant d'un programme peut être bien typé, à l'aide d'un ensemble prédéfini de règles.
L'**inférence de types** calcule des information de types non fournies par le programmeur, en vue d'une vérification de types.

L'enjeu fondamental est double :

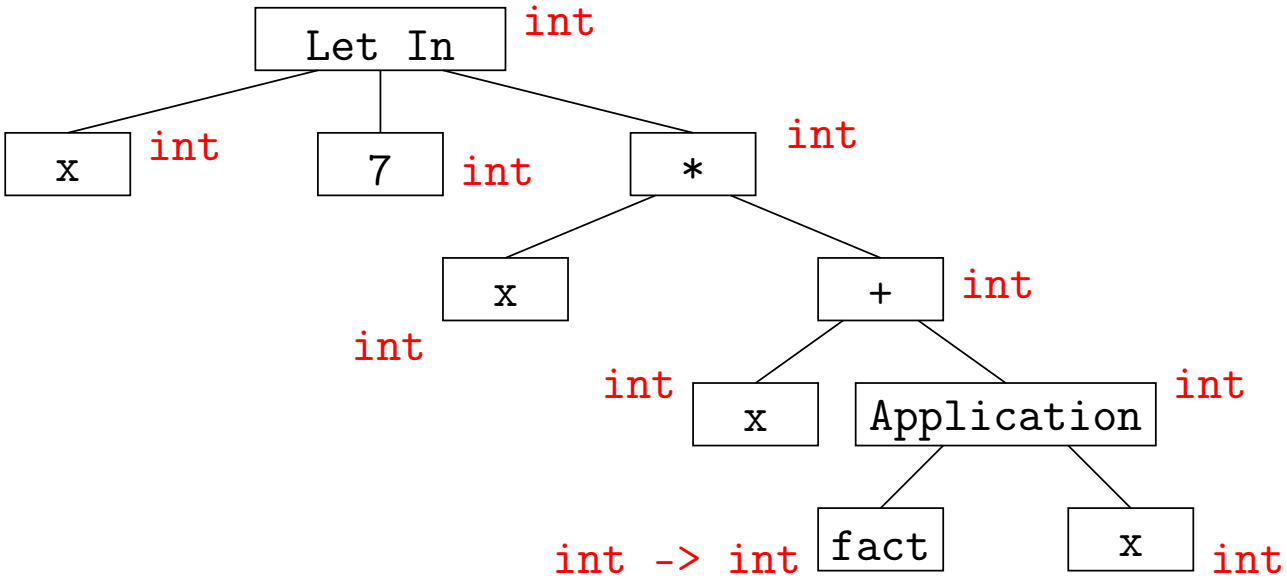
1. **éliminer des programmes invalides**
   "*well typed programs do not go wrong*"
2. **fournir des informations au compilateur**
   quelle **représentation** doit-on **utiliser** pour telle expression ?
   quelle **représentation** doit-on **supposer** pour un argument de fonction ?

Inférence de types : exemple

Reprenons notre **exemple** :

```
let x = 7 in x * (x + fact 2)
```

Résultat : production d'un AST **décoré par les informations de types**



- l'inférence repose sur des **règles de types** vues au cours 2
- en cas d'**echec** :

  `Error: This expression has type t but an expression was expected of type t'.`

Transformation de l'arbre syntaxique et production de code

Cinquième phase :

Transformation d'arbre syntaxique

Une phase de **transformation d'arbre syntaxique** permet de passer d'une représentation interne au compilateur à l'autre, plus bas niveau.

En pratique :

- un compilateur effectue **plusieurs** phases de ce type
- chaque **transformation** permet de passer d'un arbre proche du langage source à un arbre plus proche du langage vers lequel on compile

Exemples de transformations :

- traduction de `t.(e)` en un accès direct en mémoire (**partie 2**)
- traduction d'un appel de fonction en une suite d'opérations sur la pile (**partie 3**)

Optimisation

Sixième phase :

Optimisation du code

Une **optimisation** est une transformation d'arbre syntaxique, qui a pour but de rendre le code compilé **plus rapide**, **moins gourmand en mémoire**, ou généralement **plus efficace**.

Exemple : **optimisation des appels récursifs terminaux**

- réduit l'espace en pile nécessaire, quand c'est possible
- vu en amphi 2...

Exemple : **propagation de constantes**

- on calcule tout ce qui peut l'être
- `let x = 8 in fun y -> (x + 2)* y` devient `fun y -> 10 * y`

Assemblage

Septième phase :

Assemblage

L'**assemblage** a pour but de traduire un arbre syntaxique bas niveau vers une **représentation du code final** :

- du **bytecode** (un genre d'assembleur spécifique à OCaml)
  dans le cas d'`ocamlc`
- du **code assembleur machine, en binaire**
  dans le cas d'`ocamlopt`

- le **bytecode** est un langage bas niveau interprété par un runtime spécifique
- la traduction est généralement **assez directe**

Note : un compilateur réel comporte bien sûr plus de phases,
des phases plus complexes, et plusieurs **représentations intermédiaires**

Outline

Questions liées à la gestion de la mémoire

La mémoire est une ressource précieuse. Comment est-elle gérée ?

- comment un programme peut-il **obtenir plus de mémoire** ?
  pour stocker de nouvelles structures de données, tableaux, listes...
- comment un programme peut-il **rendre de la mémoire inutilisée ?**
  lorsque des structures volumineuses deviennent inutiles...

Comment les données y sont-elles stockées ?

- à quoi ressemble **la représentation physique** des types abstraits auxquels nous sommes habitués ?
  enregistrements, tableaux, types sommes, etc...

Comprendre ces questions est utile pour **mieux programmer en OCaml**
mais aussi pour mieux **appréhender d'autres langages**
et pour relier du code OCaml avec d'autres langages

Gestion manuelle de la mémoire

Le programmeur est **responsable** de **toutes les requêtes** pour **solliciter** de la mémoire ou **rendre** de la mémoire inutilisée.

Langages où la **mémoire doit être gérée ainsi** :

- **C** : très utilisé pour le code bas niveau (e.g., noyau Linux)
  gestion complètement manuelle...
- **C++** : une extension de C orientée objet
  offre à la fois gestion manuelle et gestion automatique

Ce mode est **difficile à maîtriser** et peut être source de **nombreux bugs** (plus à ce sujet dans quelques minutes)...
La plupart des langages modernes procèdent par **gestion automatique** (OCaml, Java).

Gestion manuelle de la mémoire

Primitives de gestion mémoire en C, par **blocs** :

- **allocation** à l'aide de `malloc` :
  `void *malloc(size_t size);` prend un argument décrivant la taille du bloc que l'on souhaite allouer et renvoie un pointeur
- **libération** à l'aide de `free` :
  `void free(void *ptr);` prend en argument un pointeur vers un bloc

Exemple : **allocation** d'une zone pour stocker deux entiers, **utilisation** et **libération**

```
#include<malloc.h>
int main( ){
  int * p = malloc( 2 * sizeof( int ) );
  *p = 8;
  *(p + 1) = 12;
  free( p );
  return 0;
}
```

Gestion manuelle de la mémoire et erreurs (1)

La gestion manuelle de la mémoire est **difficile** et **demande beaucoup de rigueur**... La moindre erreur peut **provoquer un crash**.

Exemple :

```
int main( ){
  int * p = malloc( sizeof( int ) );
  int q;
  if( q ) free( p );
  * p = 3; // ecriture via un pointeur invalide
  return 0
}
```

- si le pointeur est libéré, `* p = 3` écrit dans de la mémoire dont on ne dispose plus, au moins en théorie
- **résultat possible** : `Segmentation fault`
- **autre résultat possible** : on écrase d'autres données
  aucune garantie sur une écriture en C !

Gestion manuelle de la mémoire et erreurs (1)

Un **autre exemple** :

```
int f( int a ){
  int * p = malloc( sizeof( int ) );
  *p = a + 1;
  return *p;
}
```

Lors d'un appel à la fonction `f` :

- un bloc est **alloué**, mais n'est **pas libéré**
- il ne reste **aucun pointeur** vers ce bloc, donc il ne peut plus être alloué après le retour de `f`
- ce bloc reste donc en mémoire **jusqu'à la fin de l'exécution du programme**
- si on appelle `f` souvent, on finit par n'avoir **plus du tout de mémoire disponible** : le système peut ralentir, ce programme ou un autre peut planter...

On appelle cela une **fuite mémoire** (très difficile à corriger).

Gestion automatique de la mémoire

L'**implémentation du langage** (compilateur, librairies, interprète runtime) est **responsable** de **toutes les requêtes** pour **solliciter** de la mémoire, **rendre** de la mémoire inutilisée ou bien **réutiliser** de la mémoire inutilisée.

Le plus souvent :

- l'**allocation** est **manuelle** ou **implicite** :
  `new` en Java
  constructeur non constant, paire, tableau, enregistrement en OCaml
- la **libération** ou la **réutilisation** est **automatique**
  Java, OCaml...

La tâche la plus **difficile** est la **libération** / **réutilisation**.
Nous allons l'étudier un peu, même si elle est en pratique transparente (c'est à dire qu'un utilisateur débutant n'a pas besoin d'y penser).

Mémoire inutilisée

Voyons une **suite d'opérations** (certes artificielle) :

```
(* allocation:
 *  #lk: maillon de liste; #pk: paire *)
let l =
  let l0 = [ (3, 4) ] in      (* alloc: #p1, #l1 *)
  let l1 = (12, 7) :: l0 in   (* alloc: #p2, #l2 *)
  let l1 = (2, 9) :: l0 in    (* alloc: #p3, #l3 *)
                              (* inutile: #p2, #l2 *)
  (List.length l1, l0)        (* alloc: #p4 *)
let _ = ...                   (* inutile: #p3, #l3 *)
```

À chaque création d'une paire ou d'un maillon de liste correspond une **allocation**. Lorsqu'un élément alloué **devient inaccessible**, son stockage devient inutile :

- #p2 et #l2 deviennent inutiles lorsque `l1` devient in-atteignable
- lorsque le calcul de `l` est terminé, les éléments #p3 et #l3 deviennent aussi in-atteignables

Mémoire inutilisée

Voyons une **suite d'opérations** (certes artificielle) :

```
(* allocation:
 *  #lk: maillon de liste; #pk: paire *)
let l =
  let l0 = [ (3, 4) ] in      (* alloc: #p1, #l1 *)
  let l1 = (12, 7) :: l0 in   (* alloc: #p2, #l2 *)
  let l1 = (2, 9) :: l0 in    (* alloc: #p3, #l3 *)
                              (* inutile: #p2, #l2 *)
  (List.length l1, l0)        (* alloc: #p4 *)
let _ = ...                   (* inutile: #p3, #l3 *)
```

Note :

- en OCaml, la mémoire inutilisée n'est **jamais rendue au système**
- par contre, elle est recyclée pour **réutilisation** par le programme (plutôt que de solliciter à nouveau de la mémoire au système)

GC (Glaneur de cellules / Garbage collector)

Définition : GC

Un **GC** est un **ensemble de fonctions**, fournies par le runtime ou par une librairie, qui **recherche les cellules in-atteignables** et permet de les **recycler**

Revenons à l'**exemple précédent** :

```
let l =
  let l0 = [ (3, 4) ] in
```

Comment fonctionne un GC ?

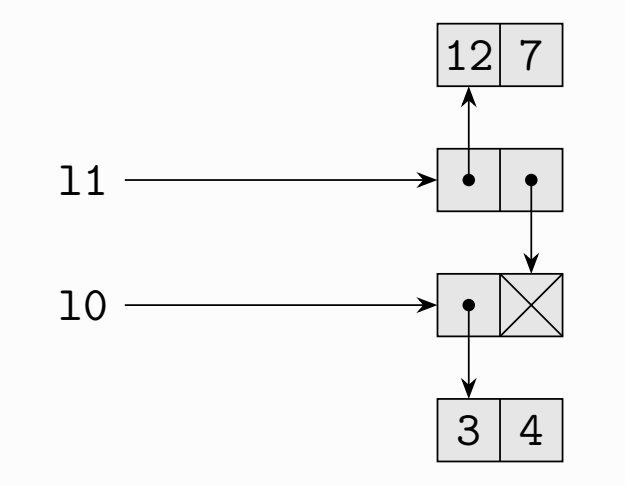
GC (Glaneur de cellules / Garbage collector)

Définition : GC

Un **GC** est un **ensemble de fonctions**, fournies par le runtime ou par une librairie, qui **recherche les cellules in-atteignables** et permet de les **recycler**

Revenons à l'**exemple précédent** :

```
let l =
  let l0 = [ (3, 4) ] in
  let l1 = (12, 7) :: l0 in
```



Comment fonctionne un GC ?

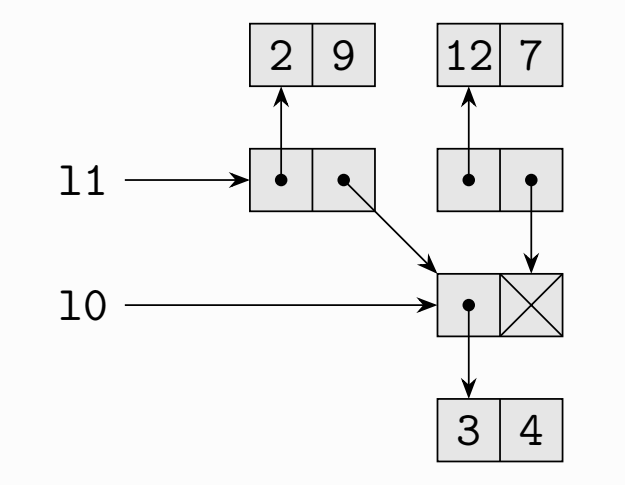
GC (Glaneur de cellules / Garbage collector)

Définition : GC

Un **GC** est un **ensemble de fonctions**, fournies par le runtime ou par une librairie, qui **recherche les cellules in-atteignables** et permet de les **recycler**

Revenons à l'**exemple précédent** :

```
let l =
  let l0 = [ (3, 4) ] in
  let l1 = (12, 7) :: l0 in
  let l1 = (2, 9) :: l0 in
```



Comment fonctionne un GC ?

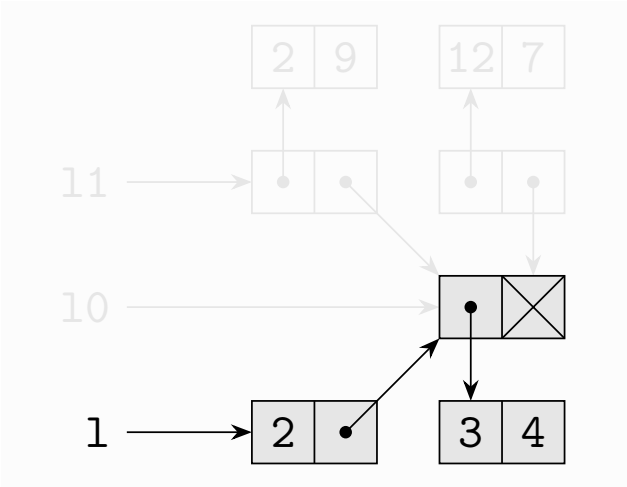
GC (Glaneur de cellules / Garbage collector)

Définition : GC

Un **GC** est un **ensemble de fonctions**, fournies par le runtime ou par une librairie, qui **recherche les cellules in-atteignables** et permet de les **recycler**

Revenons à l'**exemple précédent** :

```
let l =
  let l0 = [ (3, 4) ] in
  let l1 = (12, 7) :: l0 in
  let l1 = (2, 9) :: l0 in
  (List.length l1, l0)
```



Comment fonctionne un GC ?

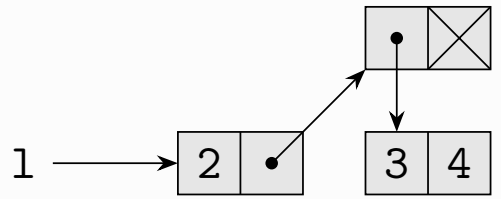
GC (Glaneur de cellules / Garbage collector)

Définition : GC

Un **GC** est un **ensemble de fonctions**, fournies par le runtime ou par une librairie, qui **recherche les cellules in-atteignables** et permet de les **recycler**

Revenons à l'**exemple précédent** :

```
let l =
  let l0 = [ (3, 4) ] in
  let l1 = (12, 7) :: l0 in
  let l1 = (2, 9) :: l0 in
  (List.length l1, l0)
(* declenchement du GC *)
Gc.full_major ()
```



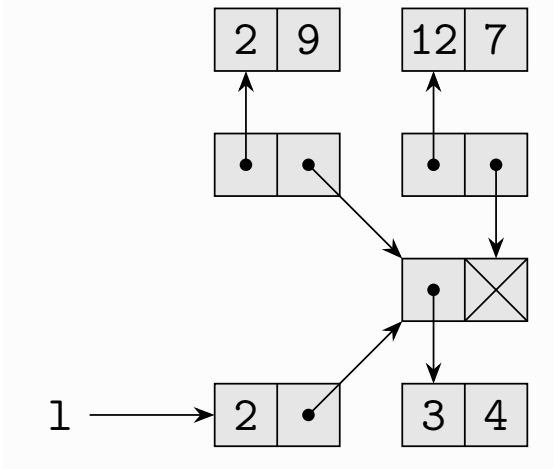
Comment fonctionne un GC ?

GC : quelques techniques

Algorithme "mark and sweep" : le fondement de la plupart des GC

1. **coloration** de chaque zone mémoire utilisée en **blanc**
2. visite des zones mémoires atteignables à partir des variables et coloration en **noir**
3. **marquage** de toute zone blanche comme réutilisable

Entrée **Mark** **Sweep**

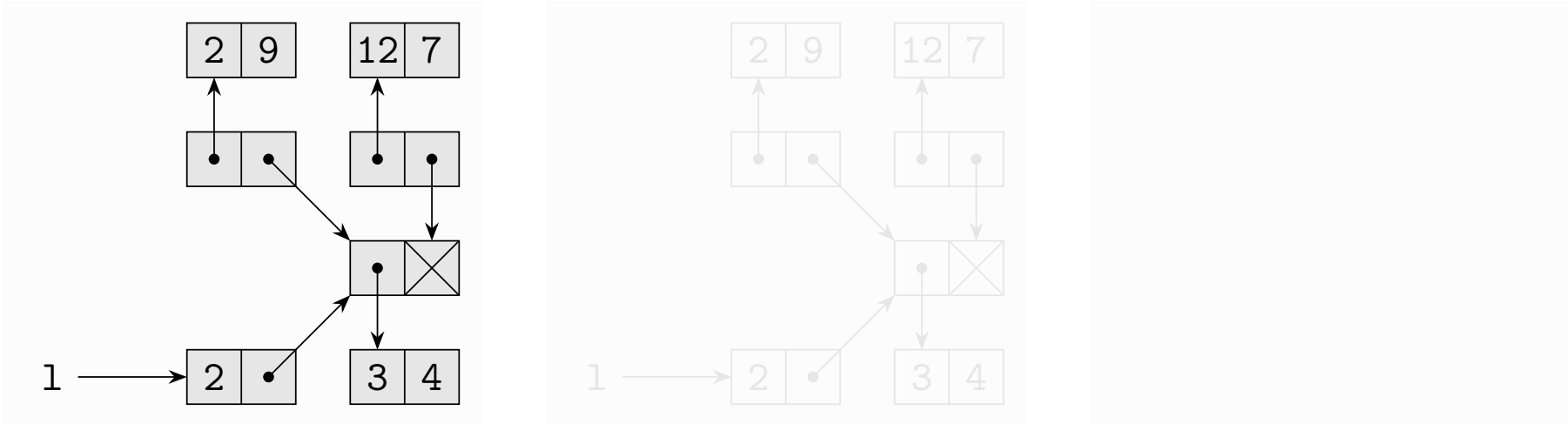


GC : quelques techniques

Algorithme "mark and sweep" : le fondement de la plupart des GC

1. **coloration** de chaque zone mémoire utilisée en **blanc**
2. visite des zones mémoires atteignables à partir des variables et coloration en **noir**
3. **marquage** de toute zone blanche comme réutilisable

Entrée **Mark** **Sweep**

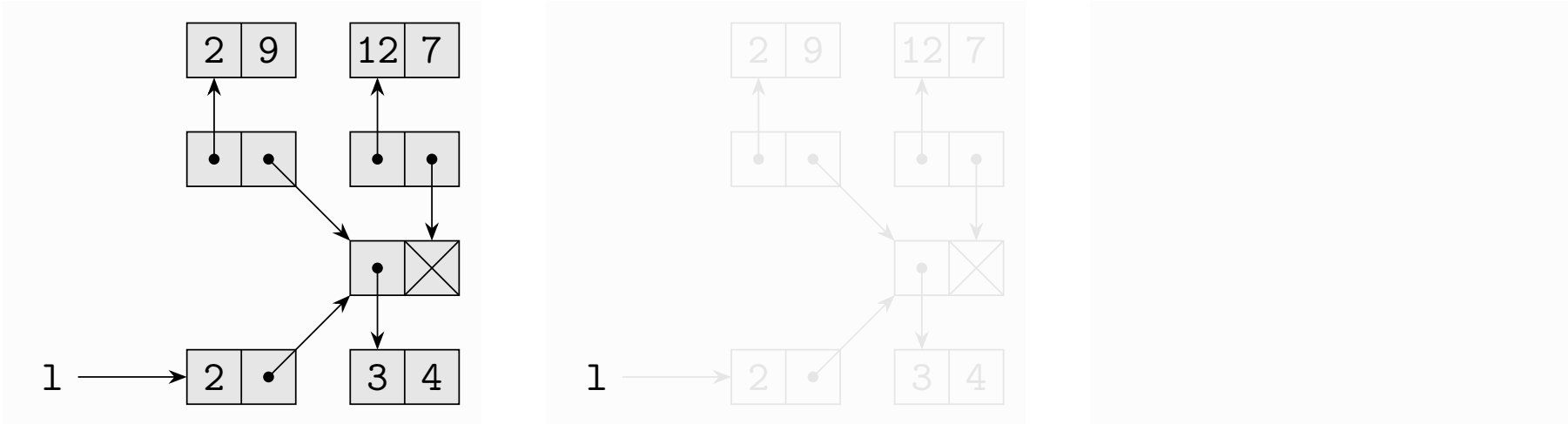


GC : quelques techniques

Algorithme "mark and sweep" : le fondement de la plupart des GC

1. **coloration** de chaque zone mémoire utilisée en **blanc**
2. visite des zones mémoires atteignables à partir des variables et coloration en **noir**
3. **marquage** de toute zone blanche comme réutilisable

Entrée **Mark** **Sweep**

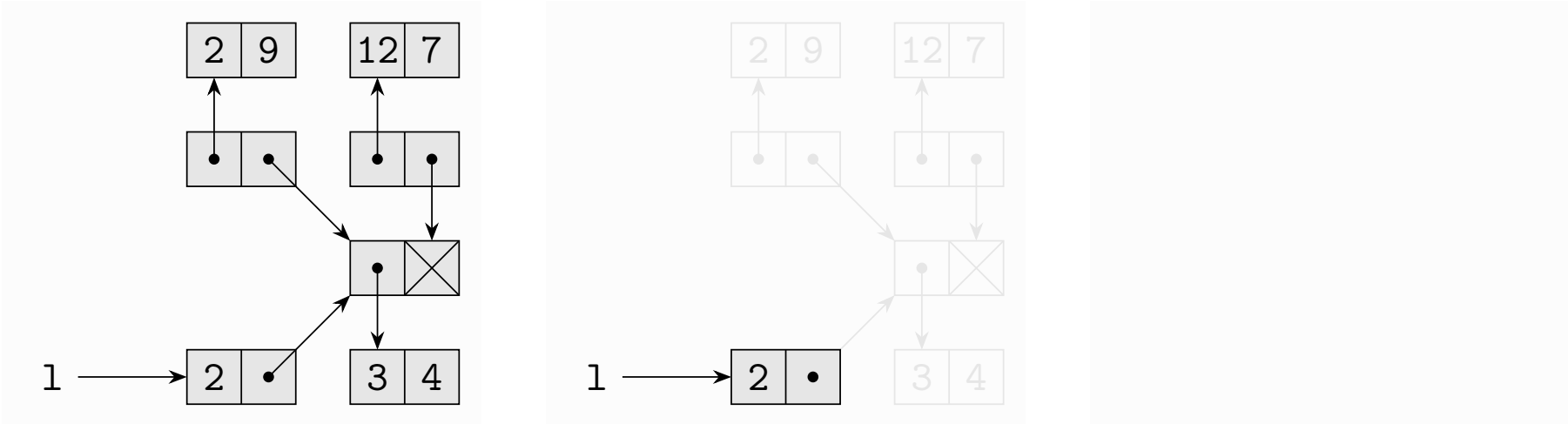


GC : quelques techniques

Algorithme "mark and sweep" : le fondement de la plupart des GC

1. **coloration** de chaque zone mémoire utilisée en **blanc**
2. visite des zones mémoires atteignables à partir des variables et coloration en **noir**
3. **marquage** de toute zone blanche comme réutilisable

Entrée **Mark** **Sweep**

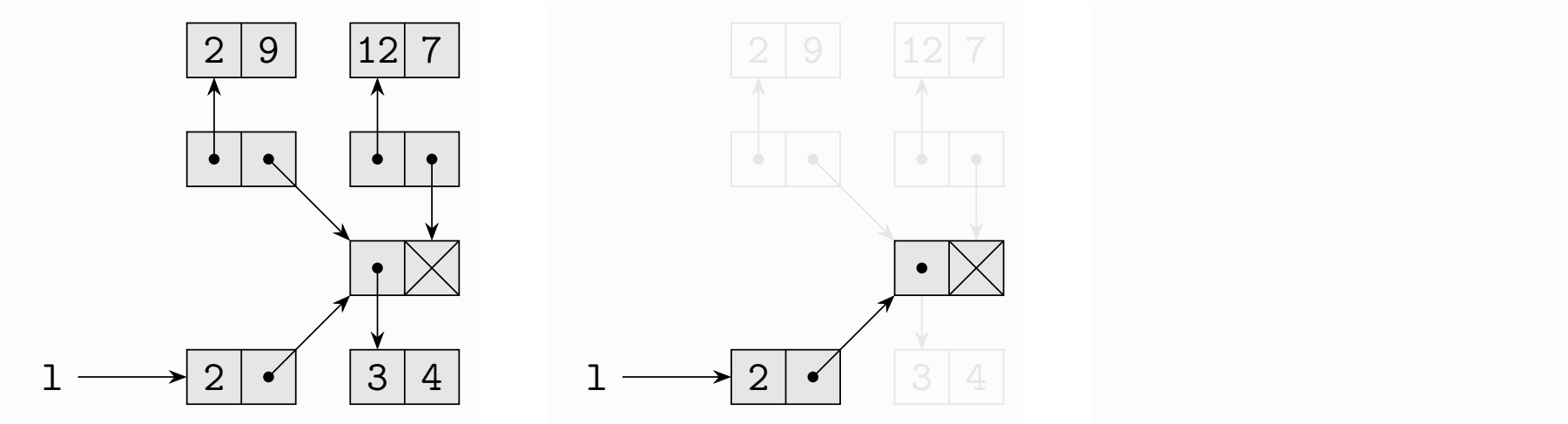


GC : quelques techniques

Algorithme "mark and sweep" : le fondement de la plupart des GC

1. **coloration** de chaque zone mémoire utilisée en **blanc**
2. visite des zones mémoires atteignables à partir des variables et coloration en **noir**
3. **marquage** de toute zone blanche comme réutilisable

Entrée **Mark** **Sweep**

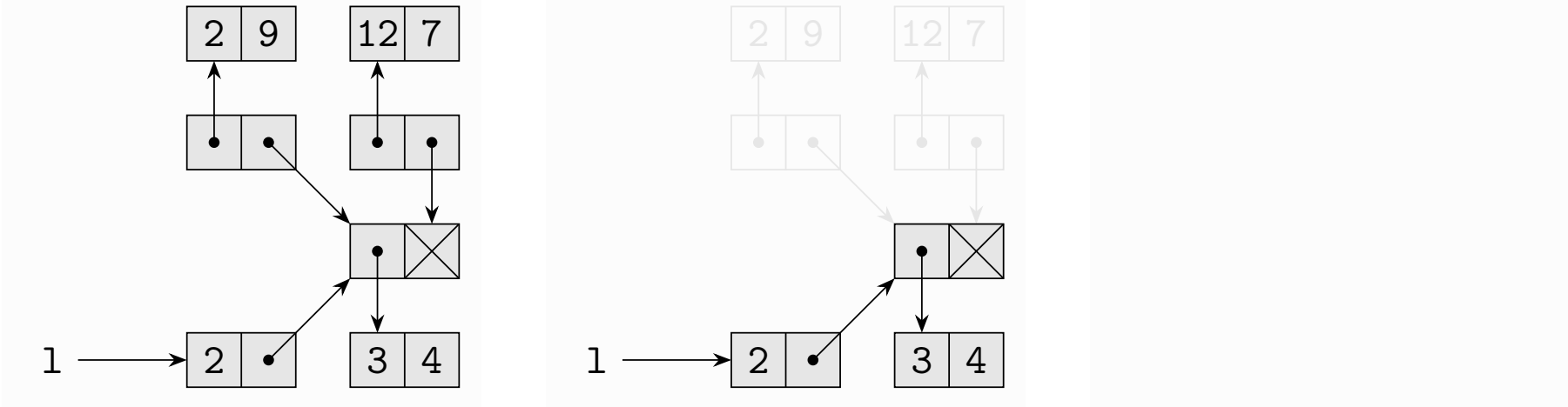


GC : quelques techniques

Algorithme "mark and sweep" : le fondement de la plupart des GC

1. **coloration** de chaque zone mémoire utilisée en **blanc**
2. visite des zones mémoires atteignables à partir des variables et coloration en **noir**
3. **marquage** de toute zone blanche comme réutilisable

Entrée **Mark** **Sweep**

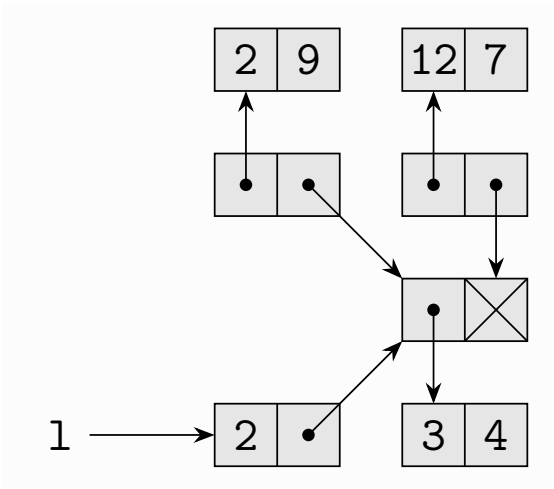


GC : quelques techniques

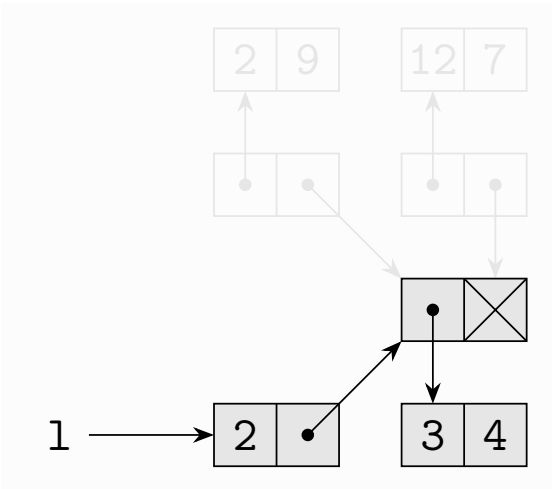
Algorithme "mark and sweep" : le fondement de la plupart des GC

1. **coloration** de chaque zone mémoire utilisée en **blanc**
2. visite des zones mémoires atteignables à partir des variables et coloration en **noir**
3. **marquage** de toute zone blanche comme réutilisable

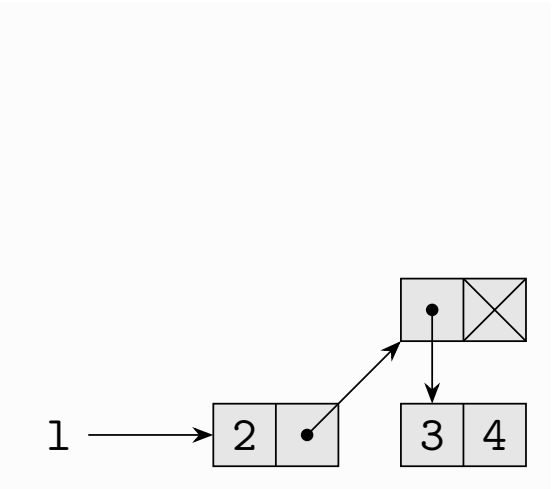
Entrée



Mark



Sweep



GC : quelques techniques

Algorithme "mark and sweep" : le fondement de la plupart des GC

1. **coloration** de chaque zone mémoire utilisée en **blanc**
2. visite des zones mémoires atteignables à partir des variables et coloration en **noir**
3. **marquage** de toute zone blanche comme réutilisable

Il existe **beaucoup d'autres techniques**.

Notion de **générations** :

- de nombreuses valeurs ont une durée de vie **courte**
  arguments d'une fonction, valeurs intermédiaires
- **division du tas** :
  tas **mineur** : données récentes (durée de vie souvent courte)
  tas **majeur** : données copiées depuis le tas mineurs, après un GC de celui-ci

GC : quelques techniques

Algorithme "mark and sweep" : le fondement de la plupart des GC

1. **coloration** de chaque zone mémoire utilisée en **blanc**
2. visite des zones mémoires atteignables à partir des variables et coloration en **noir**
3. **marquage** de toute zone blanche comme réutilisable

Faire un GC efficace est **très difficile** :

- un cycle (complet ou incomplet) de GC **arrête** le programme
- on ne veut **pas arrêter trop souvent** et **limiter la mémoire demandée au système**
- **verrou global** : GC concurrent plus complexe, absent de OCaml

Gestion manuelle vs gestion dynamique

Comme d'habitude, chaque technique a **des avantages** et **des inconvénients**.

Gestion manuelle :

- permet un **contrôle très fin**
  taille des blocs, restitution de mémoire à l'OS
- mais **difficile à maîtriser**
  erreurs subtiles : pointeurs invalides, fuites de mémoire...

Gestion automatique :

- **très sûre** en général, puisque le langage fait tout
  même pas de pointeurs nuls en OCaml !
- **moins flexible**

Dans la plupart des cas, la gestion automatique est un meilleur choix.
Si nécessaire, on peut mélanger les deux (exemple : on peut mélanger du code C avec du code OCaml en créant une interface...).

Représentation des données

À chaque langage son paradigme :

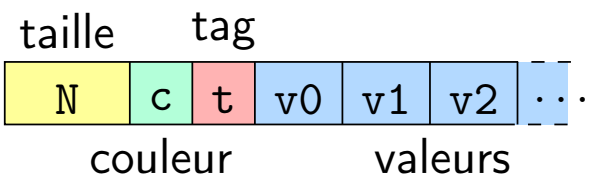
- **OCaml** maîtrise complètement **la représentation des données**
- **Java** également
- mais en **C** ou **C++**, le **programmeur choisit la représentation**

Même s'il n'est pas indispensable de connaître la représentation physique des données OCaml, c'est tout de même intéressant.

L'unité de stockage de base : bloc

Un **bloc** décrit une **zone contiguë** contenant un élément de structure (une valeur d'un type somme, un tableau, une paire, un enregistrement...).

Bloc : en-tête + contenu



- **taille** : nombre de mots (32 ou 64 bits)
- **couleur** : utilisée par le GC
- **tag** : caractérise le contenu
- **valeurs** : contenu du bloc

Valeurs et blocs

On distingue **deux types de valeurs**.

Valeurs non boxées : sont **stockées directement, dans une cellule**

- **entiers** : sur 31 ou 63 bits (un bit à 1 —distinction pointeurs)
- **booléens** : un entier valant 0 ou 1
- **type unité** : 0
- **types sommes, constructeurs sans argument** : un code entier...

Valeurs boxées : nécessitent **un bloc complet, avec un tag particulier**

- **uplets** : une valeur par élément
- **tableaux** : une valeur par cellule
- **types sommes, constructeurs avec argument** :
  un code entier dans le tag + une valeur par argument
- **flottants**

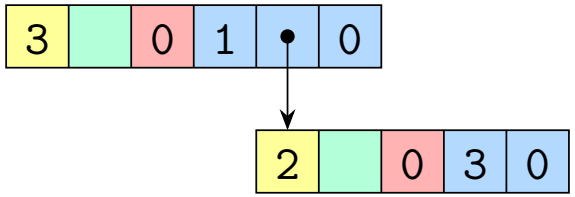
Représentation d'une valeur composée

Quelques **tags** :

- **0** : enregistrement, tableau, ou premier constructeur
- **1** : second constructeur...
- **247** : fonction ou clôture
- **253** : flottant

Exemple :

```
type t = A of t*(int*bool)*t
       | B | C
let x = (A (C, (3, true), B))
```



Applications :

- **interfaces** avec d'autres langages :
  nécessitent de connaître la représentation
- le module `Obj` permet de jouer avec la représentation...
  mais non documenté et sans garantie (donc, vraiment juste pour jouer)

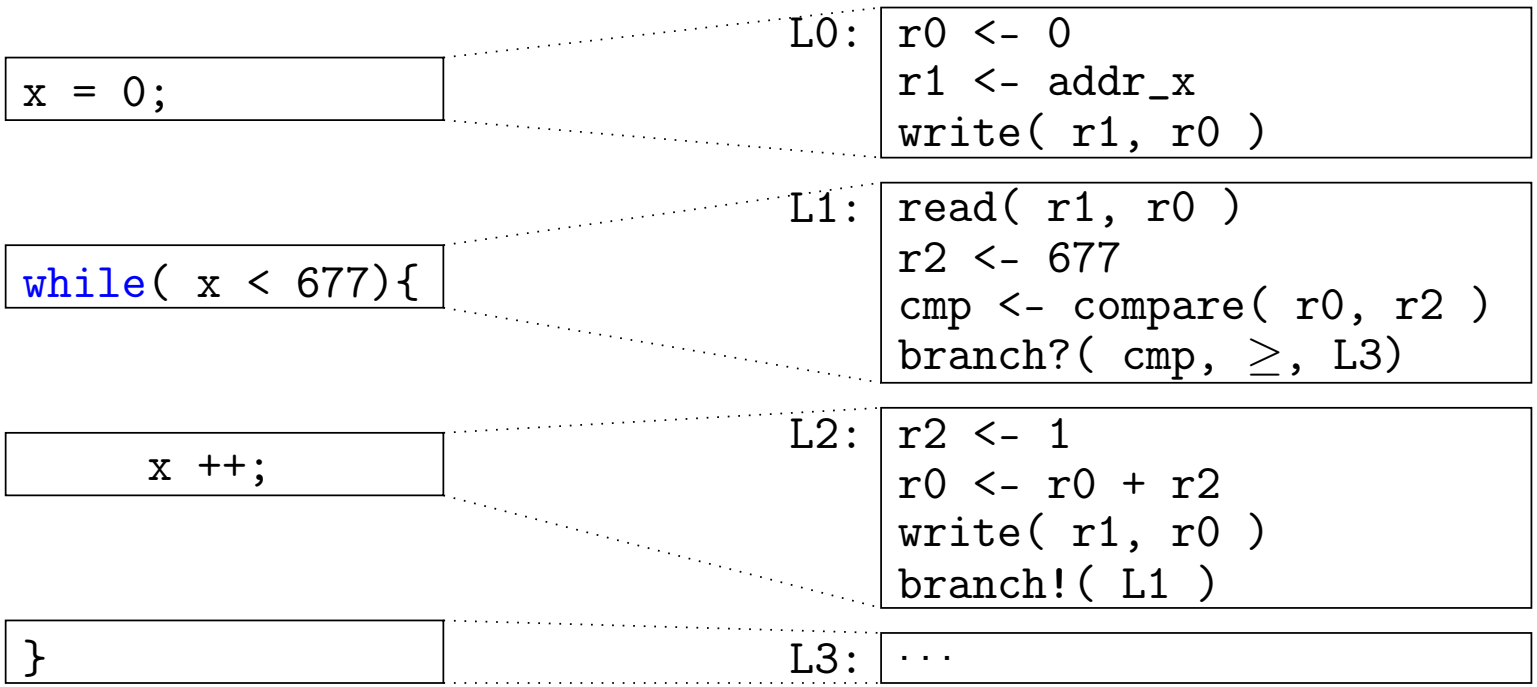
Outline

Compilation d'un langage impératif

Dans le cas d'un **programme impératif**, à première vue, la **compilation** vers du code machine est **relativement évidente**.

Exemple :

- machine avec registres et arithmétiques ($+$, $\leq$) sur registres
- lectures et écritures en mémoire via des instructions spécifiques
- branchements (formes de `goto`) conditionnels ou non

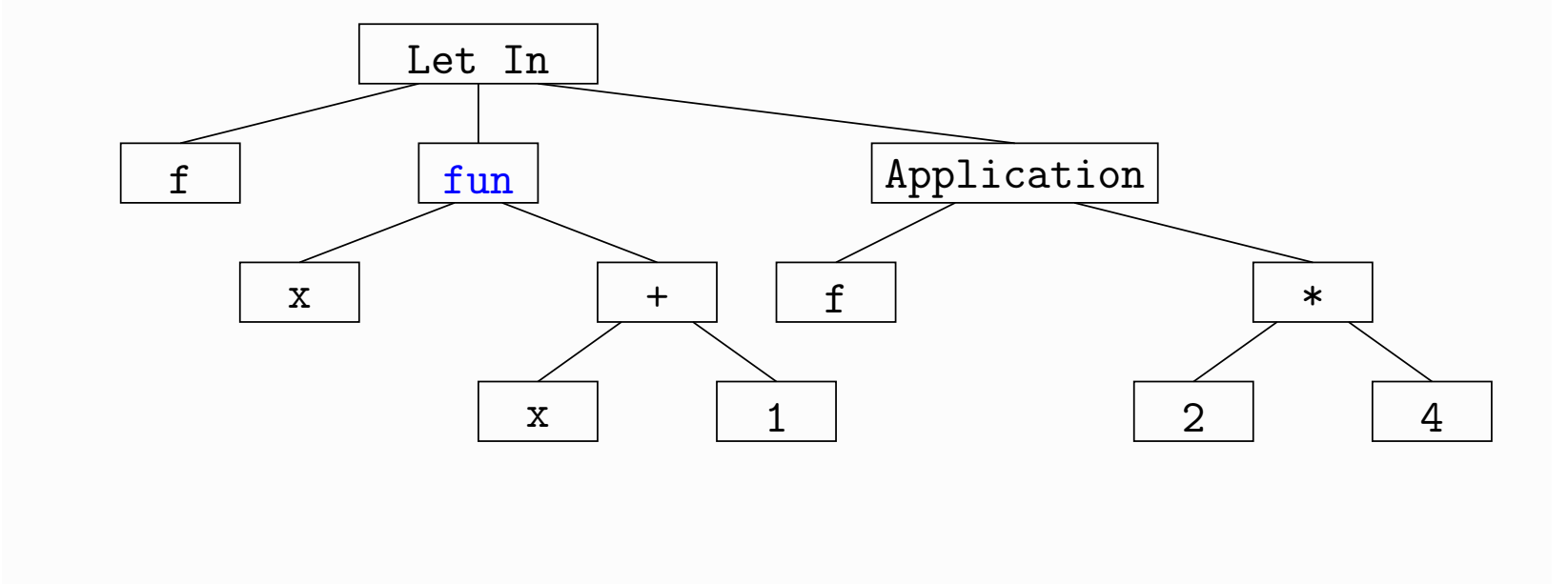


Évaluation d'un langage fonctionnel

La traduction instruction par instruction se prête aux langages impératifs, mais **pas aux langages fonctionnels à base d'expressions**.

Interprétation d'une **expression OCaml** simple :

- **expression** `let f = fun x -> x + 1 in f 8`
- **évaluation** pas à pas :



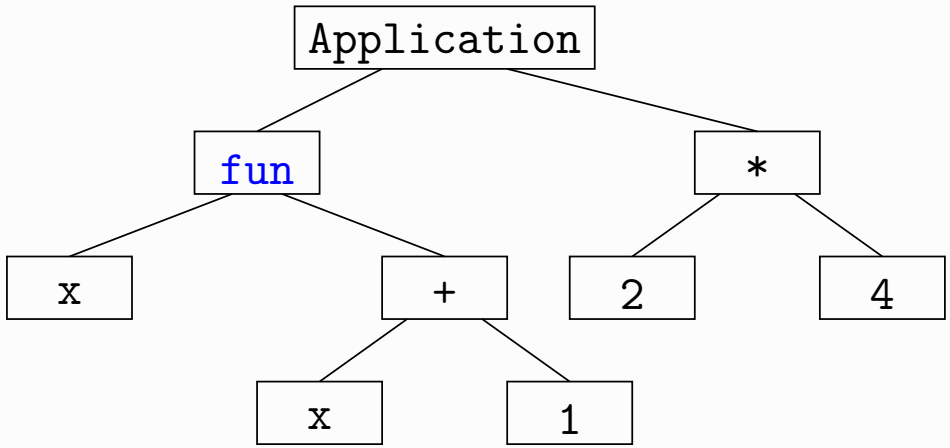
1. état initial

Évaluation d'un langage fonctionnel

La traduction instruction par instruction se prête aux langages impératifs, mais **pas aux langages fonctionnels à base d'expressions**.

Interprétation d'une **expression OCaml** simple :

- **expression** `let f = fun x -> x + 1 in f 8`
- **évaluation** pas à pas :



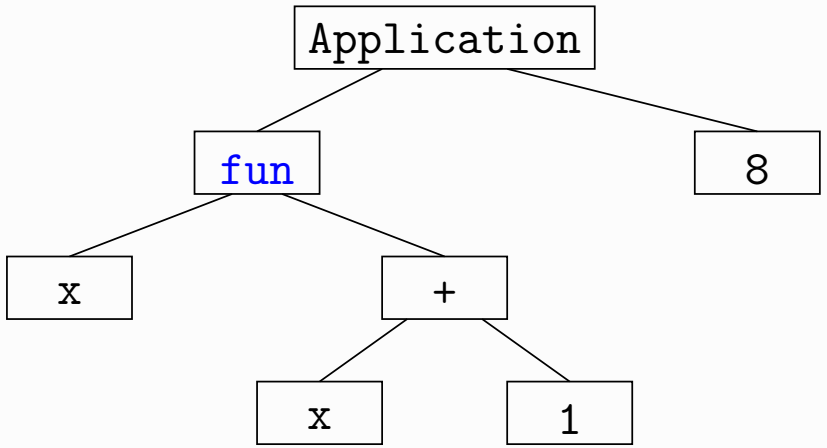
2. **résolution** de `f`, évaluation du corps

Évaluation d'un langage fonctionnel

La traduction instruction par instruction se prête aux langages impératifs, mais **pas aux langages fonctionnels à base d'expressions**.

Interprétation d'une **expression OCaml** simple :

- **expression** `let f = fun x -> x + 1 in f 8`
- **évaluation** pas à pas :



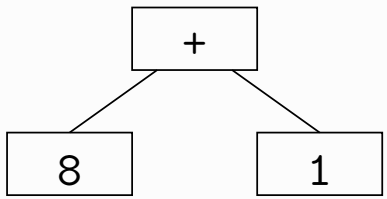
3. **évaluation** de l'argument

Évaluation d'un langage fonctionnel

La traduction instruction par instruction se prête aux langages impératifs, mais **pas aux langages fonctionnels à base d'expressions**.

Interprétation d'une **expression OCaml** simple :

- **expression** `let f = fun x -> x + 1 in f 8`
- **évaluation** pas à pas :



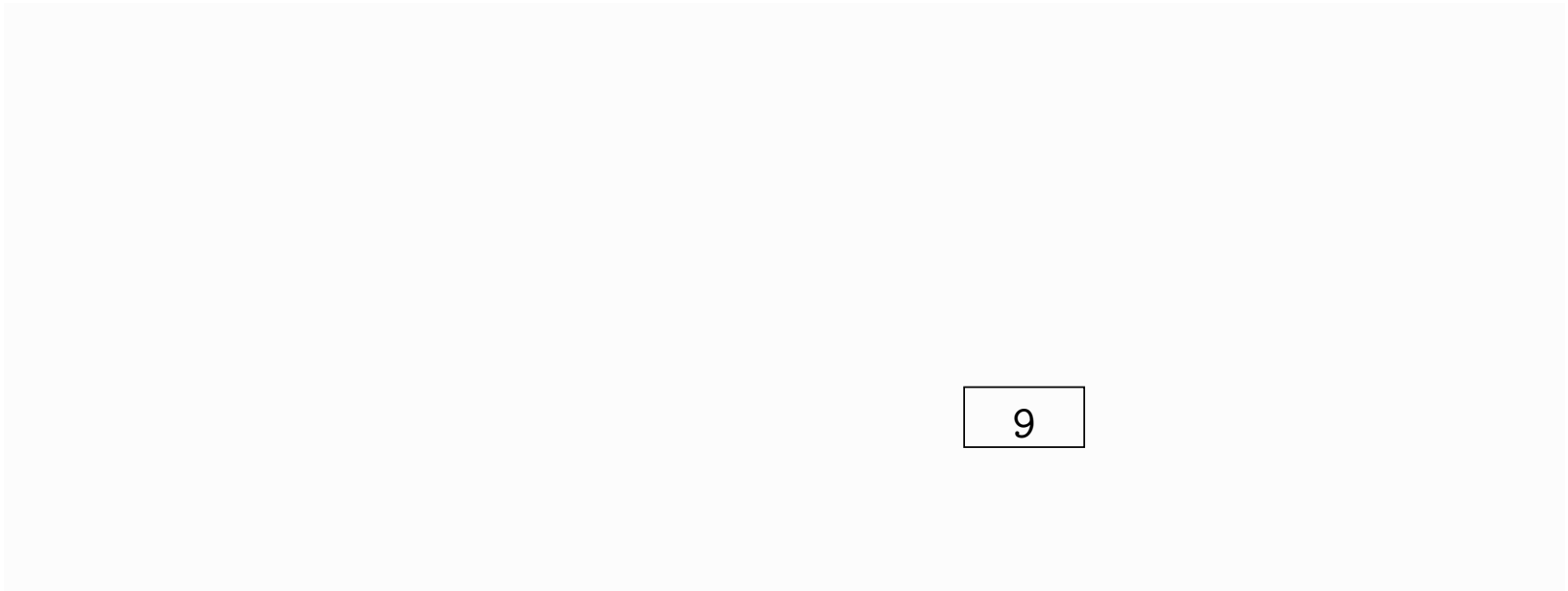
4. passage du paramètre

Évaluation d'un langage fonctionnel

La traduction instruction par instruction se prête aux langages impératifs, mais **pas aux langages fonctionnels à base d'expressions**.

Interprétation d'une **expression OCaml** simple :

- **expression** `let f = fun x -> x + 1 in f 8`
- **évaluation** pas à pas :



5. **évaluation** de l'expression

Techniques pour la compilation d'un langage fonctionnel

Le calcul est **beaucoup plus dynamique** qu'une suite d'instructions :

- l'**environnement** (variables visibles) change à chaque point
- **fonctions**, **paramètres**, **clôtures**...

Dans la suite, nous étudions un embryon de la machine abstraite de OCaml (avec plusieurs simplifications importantes).

Astuce 1 : **séparation** environnement / pile

- **environnement** : liste des noms visibles dans l'expression courante
- **pile** : espace pour le stockage des arguments déjà évalués

Astuce 2 : annotation de chaque occurrence de variable par un **entier** correspondant à son **rang dans l'environnement** (indice de De Bruijn)

- expression `fun x -> fun y -> y (x + 1)`
- expression **annotée** : `fun x -> fun y -> y[0] (x[1] + 1)`

Vers une machine abstraite

Évaluation d'un appel de fonction :

`fonct arg` $\longrightarrow$

1. évaluation de `arg`
2. placement du résultat sur la pile
3. évaluation de `fonct`

Évaluation du corps d'une fonction :

`fun x -> corps` $\longrightarrow$

1. si pile vide, ne rien faire
2. sinon, placement du sommet de pile `x` sur l'environnement
3. évaluation de `corps`

Notes :

- on se concentre ici sur **le noyau fonctionnel**
  dans la réalité, il faut ajouter arithmétique, types de données, etc. . .
- **définition locales** : `let x = e0 in e1` peut se voir comme `(fun x -> e1) e0`

La machine abstraite de Krivine

Instructions, pour le noyau fonctionnel minimal :

$$\begin{array}{rcll} \mathtt{I} & ::= & \mathtt{Access}(n) & n \in \mathbb{N} \\ & | & \mathtt{Push}(\mathtt{I}_0); \mathtt{I}_1 & \\ & | & \mathtt{Grab}; \mathtt{I}_0 & \end{array}$$

États : Code + Environnement + Pile

$$\begin{array}{rcl} \mathtt{Env} & ::= & (\mathtt{I} \times \mathtt{Env})\mathtt{stack} \\ \mathtt{Pile} & ::= & (\mathtt{I} \times \mathtt{Env})\mathtt{stack} \end{array}$$

Calcul :

$$\begin{array}{rcl} \langle \mathtt{Access}(0) \mid (\mathtt{I}_0, e_0) \cdot e \mid p \rangle & \longrightarrow & \langle \mathtt{I}_0 \mid e_0 \mid p \rangle \\ \langle \mathtt{Access}(1+n) \mid (\mathtt{I}_0, e_0) \cdot e \mid p \rangle & \longrightarrow & \langle \mathtt{Access}(n); \mathtt{I} \mid e \mid p \rangle \\ \langle \mathtt{Push}(\mathtt{I}_0); \mathtt{I} \mid e \mid p \rangle & \longrightarrow & \langle \mathtt{I} \mid e \mid (\mathtt{I}_0, e) \cdot p \rangle \\ \langle \mathtt{Grab}; \mathtt{I} \mid e \mid (\mathtt{I}_0, e_0) \cdot p \rangle & \longrightarrow & \langle \mathtt{I} \mid (\mathtt{I}_0, e_0) \cdot e \mid p \rangle \end{array}$$

Implémentation à l'aide de **piles** et **pointeurs**.

Compilation vers la machine abstraite de Krivine

Compilation : on note $\mathcal{C}(\texttt{P})$ le résultat de la traduction du programme `P`

$$\begin{aligned} \mathcal{C}(\texttt{x[n]}) &= \texttt{Access(n)} \\ \mathcal{C}(\texttt{fonct arg}) &= \texttt{Push}(\mathcal{C}(\texttt{arg}));\mathcal{C}(\texttt{fonct}) \\ \mathcal{C}(\texttt{fun x -> corps}) &= \texttt{Grab};\mathcal{C}(\texttt{corps}) \end{aligned}$$

On retrouve bien :

- **lecture d'une variable** : accès au `n`-ième élément de l'environnement
- **appel de fonction** : évaluation de l'argument sur la pile
- **corps de fonction** : évaluation du corps après avoir transféré l'argument de la pile vers l'environnement

Exemple :

$$\texttt{(fun y -> (fun x -> x)y)} \xrightarrow{\mathcal{C}} \texttt{Grab};\texttt{Push(Access(0))};\texttt{Grab};\texttt{Access(0)}$$

Exemple

Pour fournir un exemple d'exécution, on **ajoute des constantes** :

- on ajoute à la machine virtuelle l'instruction $\texttt{Const}(k)$, qui place la constante k sur la pile
- on définit la compilation $\mathcal{C}(\texttt{k}) = \texttt{Const}(k)$

Résultat de la compilation de `(fun x -> x)((fun y -> y)3)` :

$$
\begin{aligned}
&\langle \texttt{Push}(\texttt{Push}(\texttt{Const}(3)); \texttt{Grab}; \texttt{Access}(0)); \texttt{Grab}; \texttt{Access}(0) \mid \epsilon \mid \epsilon \rangle \\
\longrightarrow\ &\langle \texttt{Grab}; \texttt{Access}(0) \mid \epsilon \mid (\texttt{Push}(\texttt{Const}(3)); \texttt{Grab}; \texttt{Access}(0), \epsilon) \cdot \epsilon \rangle \\
\longrightarrow\ &\langle \texttt{Access}(0) \mid (\texttt{Push}(\texttt{Const}(3)); \texttt{Grab}; \texttt{Access}(0), \epsilon) \cdot \epsilon \mid \epsilon \rangle \\
\longrightarrow\ &\langle \texttt{Push}(\texttt{Const}(3)); \texttt{Grab}; \texttt{Access}(0) \mid \epsilon \mid \epsilon \rangle \\
\longrightarrow\ &\langle \texttt{Grab}; \texttt{Access}(0) \mid \epsilon \mid (\texttt{Const}(3), \epsilon) \cdot \epsilon \rangle \\
\longrightarrow\ &\langle \texttt{Access}(0) \mid (\texttt{Const}(3), \epsilon) \cdot \epsilon \mid \epsilon \rangle \\
\longrightarrow\ &\langle \texttt{Const}(3) \mid \epsilon \mid \epsilon \rangle \\
\longrightarrow\ &\langle \mid \epsilon \mid 3 \cdot \epsilon \rangle
\end{aligned}
$$

On distingue bien les étapes de **passage d'arguments** et **d'évaluation de corps de fonctions**

De la machine abstraite de Krivine à la ZINC

La **machine abstraite de Krivine** est **très éloignée de celle utilisée pour compiler OCaml** :

- **ordre d'évaluation** :
  la machine de Krivine ne force pas l'évaluation des arguments avant appel de fonction...
- **primitives manquantes** :
  arithmétique, types de données et manipulation, etc...
- **optimisation des appels récursifs terminaux**

Machine **ZINC** (Zinc Is Not Caml) :
The ZINC Experiment: An Economical Implementation of the ML Language. Xavier Leroy. 1990.

- réutilise tout de même les principes de base de la machine de Krivine
- intégrée dans le runtime OCaml...

Outline

Principaux éléments à retenir

Compilation :

- une succession de **phases** allant du **texte**, vers un **arbre syntaxique** puis une **machine abstraite**
- plusieurs phases compréhensibles à partir de ce cours
  typage, transformation d'AST...

Mémoire :

- gestion **manuelle** : allocation / libération à la main
- gestion **automatique** : récupération à l'aide d'un GC (algorithmes de parcours de graphes)

Machine abstraite :

- division de ce que nous appelions jusqu'ici pile en **pile** + **environnement**
- instructions bas niveau, implémentable à l'aide de pointeurs