

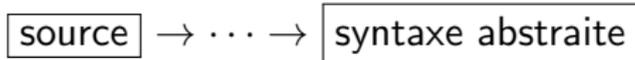
École Normale Supérieure

Langages de programmation et compilation

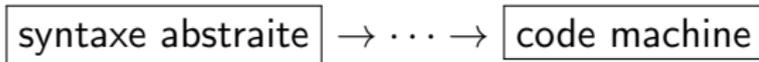
Jean-Christophe Filliâtre

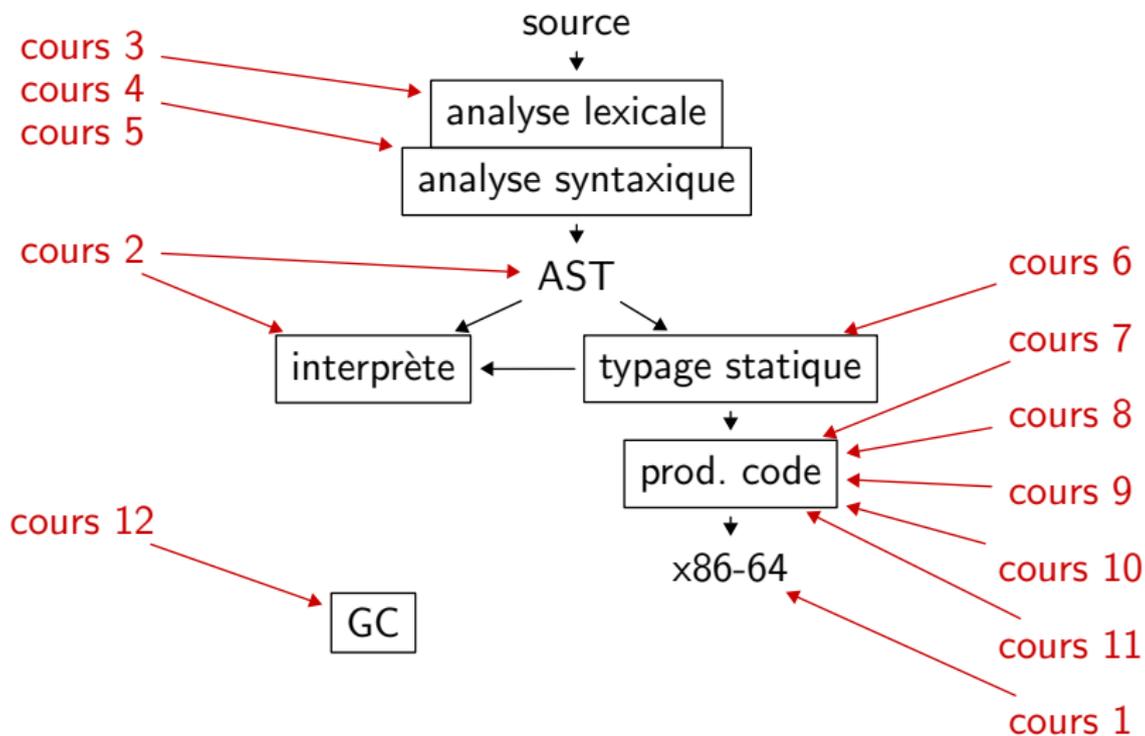
mode de passage des paramètres

on a terminé la phase d'**analyse**



on considère maintenant la phase de **synthèse**





1. stratégie d'évaluation et passage des paramètres
 - C
 - C++
 - OCaml
 - Java
 - Python
2. compilation du passage par valeur et par référence
3. correction de la compilation

stratégie d'évaluation et passage des paramètres

dans la **déclaration** d'une fonction

```
function f(x1, ..., xn) =  
  ...
```

les variables x_1, \dots, x_n sont appelées **paramètres formels** de f

et dans l'**appel** de cette fonction

```
f(e1, ..., en)
```

les expressions e_1, \dots, e_n sont appelées **paramètres effectifs** de f

dans un langage comprenant des modifications en place, une affectation

```
e1 := e2
```

modifie un emplacement mémoire désigné par l'expression e1

l'expression e1 est limitée à certaines constructions,
car des affectations comme

```
42 := 17  
f(34) := false
```

n'ont en général pas de sens

on parle de **valeur gauche** (*left value*) pour désigner les expressions
légales à gauche d'une affectation

la stratégie d'évaluation d'un langage spécifie l'ordre dans lequel les calculs sont effectués

on peut la définir à l'aide d'une sémantique formelle (cf cours 2)

le compilateur se doit de respecter la stratégie d'évaluation

en particulier, la stratégie d'évaluation **peut** spécifier

- à quel moment les paramètres effectifs d'un appel sont évalués
- l'ordre d'évaluation des opérandes et des paramètres effectifs

certain aspects de l'évaluation peuvent cependant rester **non spécifiés**

cela laisse alors de la latitude au compilateur, notamment pour effectuer des optimisations (par exemple en ordonnant les calculs comme il le souhaite)

on distingue

- **l'évaluation stricte** : les opérandes / paramètres effectifs sont évalués avant l'opération / l'appel

exemples : C, C++, Java, OCaml, Python, mini-ML du cours 2

- **l'évaluation paresseuse** : les opérandes / paramètres effectifs ne sont évalués que si nécessaire

exemples : Haskell, Clojure

mais aussi les connectives logiques `&&` et `||` de la plupart des langages

un langage **impératif** adopte systématiquement une évaluation stricte, pour garantir une séquentialité des effets de bord qui coïncide avec le texte source

par exemple, le programme OCaml

```
let r = ref 0
let id x = r := !r + x; x
let f x y = !r
let () = print_int (f (id 40) (id 2))
```

affiche 42 car les deux arguments de `f` ont été évalués

une exception est faite pour les connectives logiques `&&` et `||` de la plupart des langages, ce qui est bien pratique

```
let insertion_sort a =  
  for i = 1 to Array.length a - 1 do  
    let v = a.(i) and j = ref i in  
    while 0 < !j && v < a.(!j - 1) do  
      a.(!j) <- a.(!j - 1);  
      decr j  
    done;  
    a.(!j) <- v  
  done
```

la non-terminaison est également un effet

ainsi, le programme

```
let rec loop () = loop ()  
let f x y = x + 1  
let v = f 41 (loop ())
```

ne termine pas, bien que l'argument `y` n'est pas utilisé

un langage **purement applicatif**, c'est-à-dire sans effets de bord, peut en revanche adopter la stratégie d'évaluation de son choix, car une expression aura toujours la même valeur (on parle de **transparence référentielle**)

en particulier, il peut faire le choix d'une évaluation paresseuse

le programme Haskell

```
loop () = loop ()  
f x y = x  
main = putChar (f 'a' (loop ()))
```

termine (après avoir affiché a)

la sémantique précise également le **mode de passage** des paramètres lors d'un appel

on distingue notamment

- l'**appel par valeur** (*call by value*)
- l'**appel par référence** (*call by reference*)
- l'**appel par nom** (*call by name*)
- l'**appel par nécessité** (*call by need*)

(on parle aussi parfois de **passage** par valeur, par référence, etc.)

de **nouvelles** variables représentant les paramètres formels reçoivent les **valeurs** des paramètres effectifs

```
function f(x) =  
  x := x + 1  
  
main() =  
  int v := 41  
  f(v)  
  print(v)    // affiche 41
```

les paramètres formels désignent les **mêmes valeurs gauches** que les paramètres effectifs

```
function f(x) =  
  x := x + 1  
  
main() =  
  int v := 41  
  f(v)  
  print(v)    // affiche 42
```

les paramètres effectifs sont **substitués** aux paramètres formels, textuellement, et donc évalués seulement si nécessaire

```
function f(x, y, z) =  
    return x*x + y*y  
  
main() =  
    print(f(1+2, 2+2, 1/0)) // affiche 25  
    // on a évalué (1+2)*(1+2) + (2+2)*(2+2)  
    // 1+2 est évalué deux fois  
    // 2+2 est évalué deux fois  
    // 1/0 n'est jamais évalué
```

les paramètres effectifs ne sont évalués que si nécessaire,
mais **au plus une fois**

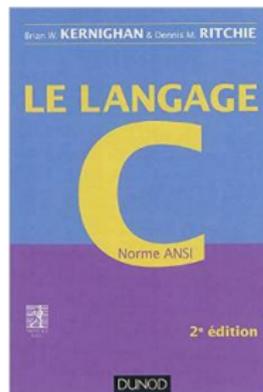
```
function f(x, y, z) =  
    return x*x + y*y  
  
main() =  
    print(f(1+2, 2+2, 1/0)) // affiche 25  
    // 1+2 est évalué une fois  
    // 2+2 est évalué une fois  
    // 1/0 n'est jamais évalué
```

quelques mots sur le langage C

le langage C est un langage impératif relativement bas niveau, notamment parce que la notion de pointeur, et d'arithmétique de pointeur, y est explicite

on peut le considérer inversement comme un assembleur de haut niveau

un ouvrage toujours d'actualité :
Le langage C
de Brian Kernighan et Dennis Ritchie



le langage C est muni d'une stratégie d'évaluation stricte,
avec appel **par valeur**

l'ordre d'évaluation n'est pas spécifié

- on trouve des types de base tels que `char`, `int`, `bool`, `float`, etc.
- un type τ^* des pointeurs vers des valeurs de type τ
si p est un pointeur de type τ^* , alors $*p$ désigne la valeur pointée par p , de type τ
si e est une valeur gauche de type τ , alors $\&e$ est un pointeur sur l'emplacement mémoire correspondant, de type τ^*
- des enregistrements, appelés *structures*, tels que

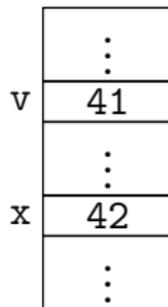
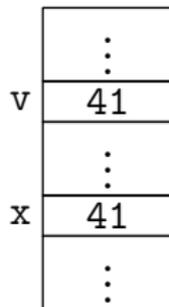
```
struct L { int head; struct L *next; };
```

si e a le type `struct L`, on note `e.head` l'accès au champ

en C, une valeur gauche est de la forme

- x , une variable
- $*e$, le déréférencement d'un pointeur
- $e.x$, l'accès à un champ de structure, si e est elle-même une valeur gauche
- $t[e]$, qui n'est autre que $*(t+e)$
- $e \rightarrow x$, qui n'est autre que $(*e).x$

```
void f(int x) {  
    x = x + 1;  
}  
  
int main() {  
    int v = 41;  
    f(v);  
    // v vaut toujours 41  
}
```



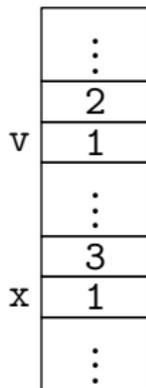
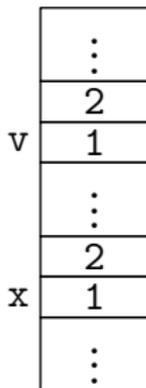
l'appel par valeur implique que les structures sont **copiées** lorsqu'elles sont passées en paramètres ou renvoyées

les structures sont également copiées lors des affectations de structures, *i.e.* des affectations de la forme $x = y$, où x et y ont le type `struct S`

```
struct S { int a; int b; };
```

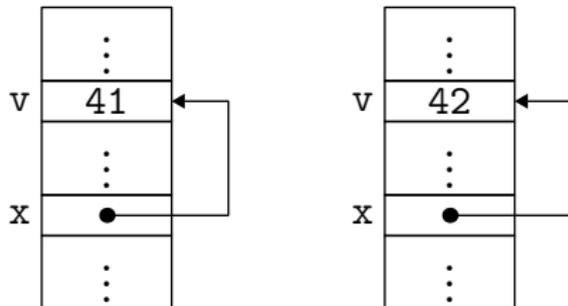
```
void f(struct S x) {
    x.b = x.b + 1;
}
```

```
int main() {
    struct S v = { 1, 2 };
    f(v);
    // v.b vaut toujours 2
}
```



on peut **simuler** un passage par référence en passant un pointeur explicite

```
void incr(int *x) {  
    *x = *x + 1;  
}  
  
int main() {  
    int v = 41;  
    incr(&v);  
    // v vaut maintenant 42  
}
```



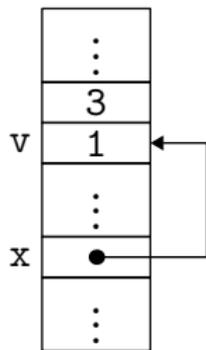
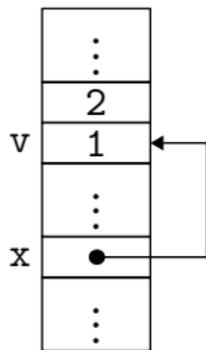
mais ce n'est que le passage d'un pointeur **par valeur**

pour éviter le coût des copies, on passe des pointeurs sur les structures le plus souvent

```
struct S { int a; int b; };
```

```
void f(struct S *x) {  
    x->b = x->b + 1;  
}
```

```
int main() {  
    struct S v = { 1, 2 };  
    f(&v);  
    // v.b vaut maintenant 3  
}
```



la manipulation explicite de pointeurs peut être dangereuse

```
int* p() {  
    int x;  
    ...  
    return &x;  
}
```

cette fonction renvoie un pointeur qui correspond à un emplacement sur la pile qui vient justement de disparaître (à savoir le tableau d'activation de `p`), et qui sera très probablement réutilisé rapidement par un autre tableau d'activation

on parle de référence fantôme (*dangling reference*)

on peut déclarer un tableau ainsi :

```
int t[10];
```

la notation $t[i]$ n'est que du sucre syntaxique pour $*(t+i)$ où

- t désigne un pointeur sur le début d'une zone contenant 10 entiers
- $+$ désigne une opération d'*arithmétique de pointeur* (qui consiste à ajouter à t la quantité $4i$ pour un tableau d'entiers 32 bits)

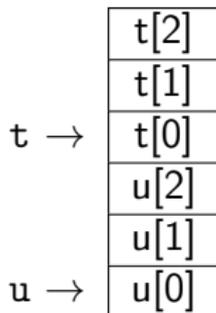
le premier élément du tableau est donc $t[0]$ c'est-à-dire $*t$

quand on passe un tableau en paramètre, on ne fait que passer le pointeur (par valeur, toujours)

on ne peut affecter des tableaux, seulement des pointeurs

ainsi, on ne peut pas écrire

```
void p() {  
    int t[3];  
    int u[3];  
    t = u;    // <- erreur  
}
```



car `t` et `u` sont des tableaux (alloués sur la pile) et l'affectation de tableaux n'est pas autorisée

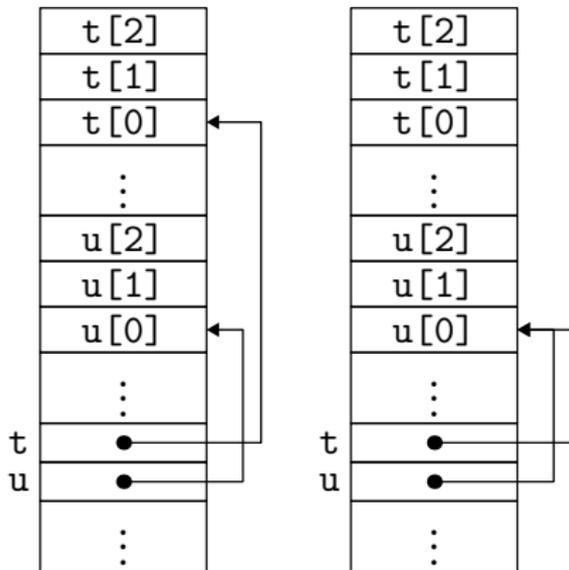
en revanche on peut écrire

```
void q(int t[3], int u[3]) {
    t = u;
}
```

car c'est exactement la même chose que

```
void q(int *t, int *u) {
    t = u;
}
```

et l'affectation de pointeurs est autorisée



quelques mots sur le langage C++

en C++, on trouve (entre autres) les types et constructions du C, avec une stratégie d'évaluation stricte

le mode de passage est **par valeur** par défaut

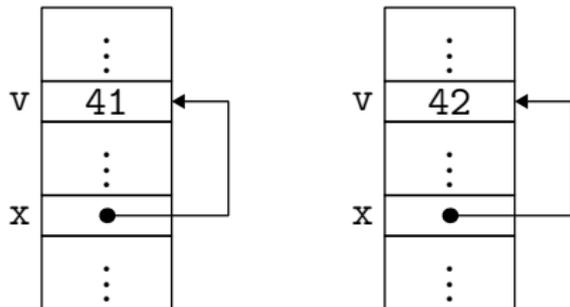
mais on trouve aussi un passage **par référence** indiqué par le symbole & au niveau de l'argument formel

```

void f(int &x) {
    x = x + 1;
}

int main() {
    int v = 41;
    f(v);
    // v vaut maintenant 42
}

```



en particulier, c'est le compilateur qui

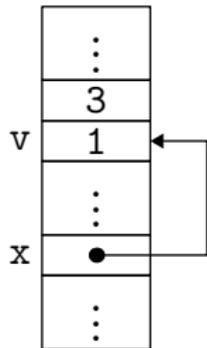
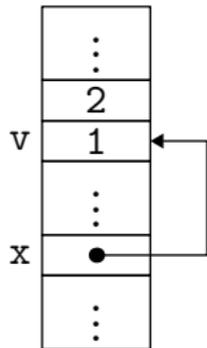
- a pris l'adresse de `v` au moment de l'appel
- a déréférencé l'adresse `x` dans la fonction `f`

on peut passer une structure par référence

```
struct S { int a; int b; };
```

```
void f(struct S &x) {  
    x.b = x.b + 1;  
}
```

```
int main() {  
    struct S v = { 1, 2 };  
    f(v);  
    // v.b vaut maintenant 3  
}
```



on peut passer un pointeur par référence

par exemple pour ajouter un élément dans un arbre

```
struct Node { int elt; Node *left, *right; };

void add(Node* &t, int x) {
    if (t == NULL) t = create(NULL, x, NULL);
    else if (x < t->elt) add(t->left, x);
    else if (x > t->elt) add(t->right, x);
}
```

quelques mots sur le langage OCaml

OCaml est muni d'une stratégie d'évaluation stricte, avec appel **par valeur**

l'ordre d'évaluation n'est pas spécifié

une valeur est

- soit d'un type primitif (booléen, caractère, entier machine, etc.)
- soit un pointeur vers un bloc mémoire (tableau, enregistrement, constructeur non constant, etc.) alloué sur le tas en général

les valeurs gauches sont les éléments de tableaux

```
a.(2) <- true
```

et les champs mutables d'enregistrements

```
x.age <- 42
```

rappel : une référence est un enregistrement

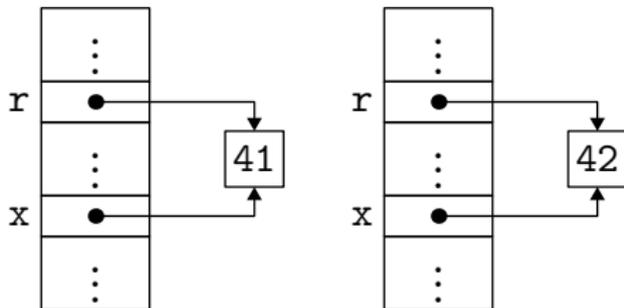
```
type 'a ref = { mutable contents: 'a }
```

et les opérations := et ! sont définies par

```
let (!) r = r.contents  
let (:=) r v = r.contents <- v
```

une référence est allouée sur le tas

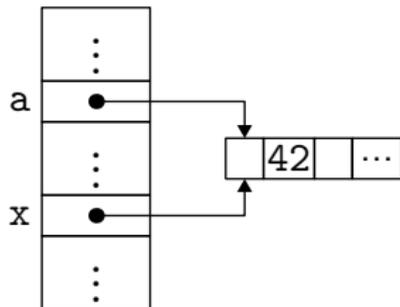
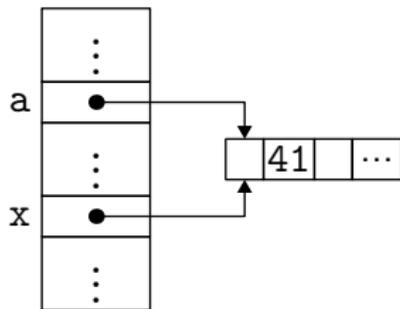
```
let f x =  
  x := !x + 1  
  
let main () =  
  let r = ref 41 in  
  f r  
  (* !r vaut maintenant 42 *)
```



c'est toujours un passage par valeur,
d'une valeur qui est un pointeur (implicite) vers une valeur mutable

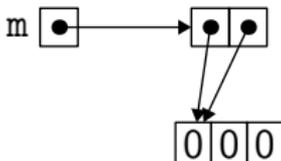
un tableau est également alloué sur le tas

```
let incr x =  
  x.(1) <- x.(1) + 1  
  
let main () =  
  let a = Array.make 17 0 in  
  a.(1) <- 41;  
  incr a  
  (* a.(1) vaut maintenant 42 *)
```



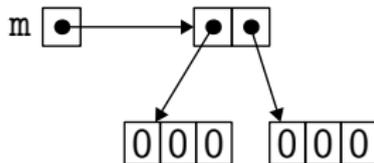
pour construire une matrice, on n'écrit pas

```
let m = Array.make 2 (Array.make 3 0)
```



mais

```
let m = Array.make_matrix 2 3 0
```



on peut **simuler l'appel par nom** en OCaml, en remplaçant les arguments par des fonctions

ainsi, la fonction

```
let f x y =  
  if x = 0 then 42 else y + y
```

peut être réécrite en

```
let f x y =  
  if x () = 0 then 42 else y () + y ()
```

et appelée comme ceci

```
let v = f (fun () -> 0) (fun () -> failwith "oups")
```

plus subtilement, on peut aussi **simuler l'appel par nécessité** en OCaml

on commence par introduire un type pour représenter les calculs paresseux

```
type 'a value = Value of 'a
              | Frozen of (unit -> 'a)
```

```
type 'a by_need = 'a value ref
```

et une fonction qui évalue un tel calcul si ce n'est pas déjà fait

```
let force l = match !l with
  | Value v -> v
  | Frozen f -> let v = f () in l := Value v; v
```

(c'est de la mémoïsation)

on définit alors la fonction `f` comme ceci

```
let f x y =  
    if force x = 0 then 42 else force y + force y
```

et on l'utilise ainsi

```
let v = f (ref (Frozen (fun () -> 1)))  
          (ref (Frozen (fun () -> ...gros calcul...)))
```

note : la construction `lazy` d'OCaml fait quelque chose de semblable (un peu plus subtilement)

quelques mots sur le langage Java

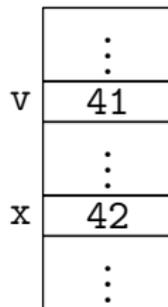
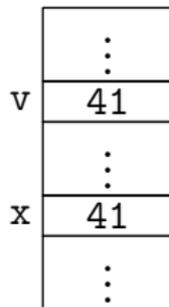
Java est muni d'une stratégie d'évaluation stricte, avec appel **par valeur**

l'ordre d'évaluation est spécifié gauche-droite

une valeur est

- soit d'un type primitif (booléen, caractère, entier machine, etc.)
- soit un pointeur vers un objet alloué sur le tas

```
void f(int x) {  
    x = x + 1;  
}  
  
int main() {  
    int v = 41;  
    f(v);  
    // v vaut toujours 41  
}
```

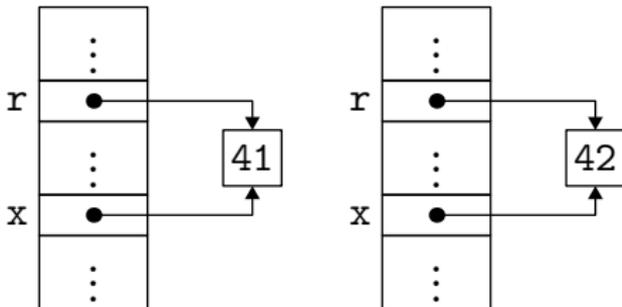


un objet est alloué sur le tas

```
class C { int f; }

void incr(C x) {
    x.f += 1;
}

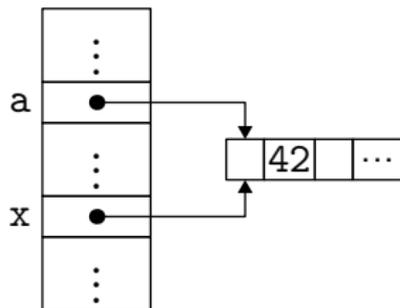
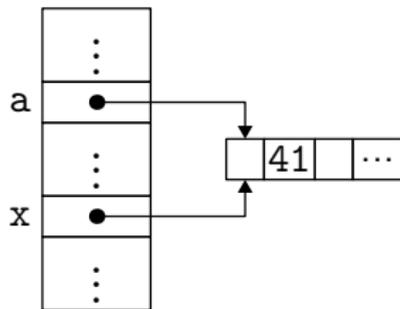
void main () {
    C r = new C();
    r.f = 41;
    incr(r);
    // r.f vaut maintenant 42
}
```



c'est toujours un passage **par valeur**,
d'une valeur qui est un pointeur (implicite) vers un objet

un tableau est aussi un objet

```
void incr(int[] x) {  
    x[1] += 1;  
}  
  
void main () {  
    int[] a = new int[17];  
    a[1] = 41;  
    incr(a);  
    // a[1] vaut maintenant 42  
}
```



quelques mots sur le langage Python

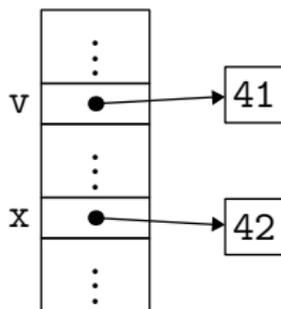
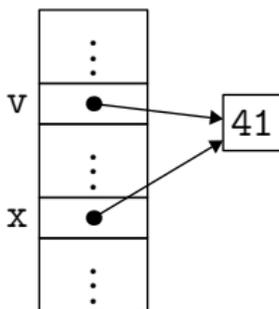
Python est muni d'une stratégie d'évaluation stricte, avec appel **par valeur**

l'ordre d'évaluation est spécifié gauche-droite
(mais droite-gauche pour une affectation)

une valeur est un pointeur vers un objet alloué sur le tas

un entier est un objet immuable

```
def f(x):  
    x += 1  
  
v = 41  
f(v)  
print(v) # affiche 41
```

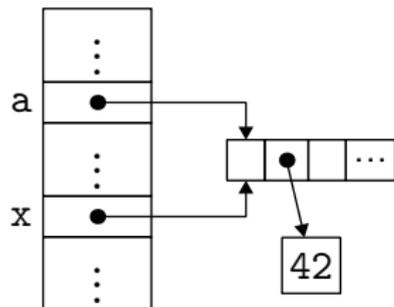
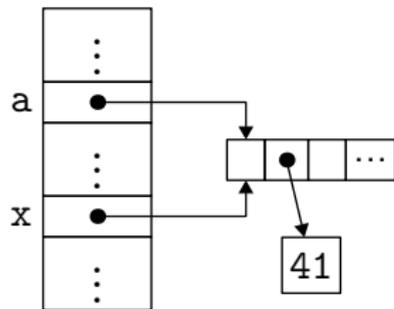


c'est toujours un passage **par valeur**,
d'une valeur qui est un pointeur (implicite) vers un objet

un tableau est un objet mutable

```
def incr(x):  
    x[1] += 1
```

```
a = [0] * 17  
a[1] = 41  
incr(a)  
# a[1] vaut maintenant 42
```



les modèles d'exécution de Java, OCaml et Python sont **très semblables** :
uniquement du passage par valeurs, de valeurs atomiques (64 bits)

même si leurs langages de surface sont très différents

C	entier par valeur	pointeur par valeur	pointeur par valeur
C++	entier par valeur	pointeur par valeur entier par référence	pointeur par valeur ou par référence
OCaml	entier par valeur	—	pointeur par valeur (par ex. type ref)
Java	entier par valeur	—	pointeur par valeur (objet)
Python	—	—	pointeur par valeur (objet)

compilation du passage par valeur et par référence

considérons la compilation d'un micro fragment de C++ avec

- des entiers
- des fonctions (mais qui ne renvoient rien)
- du passage par valeur et par référence

on considère le fragment suivant

$$\begin{array}{l}
 E \rightarrow n \\
 \quad | \quad x \\
 \quad | \quad E + E \mid E - E \\
 \quad | \quad E * E \mid E / E \\
 \quad | \quad - E \\
 \\
 C \rightarrow E == E \mid E != E \\
 \quad | \quad E < E \mid E <= E \mid E > E \mid E >= E \\
 \quad | \quad C \&\& C \\
 \quad | \quad C \parallel C \\
 \quad | \quad ! C
 \end{array}$$

$$\begin{array}{l}
 S \rightarrow x = E; \\
 \quad | \quad \text{if} (C) S \\
 \quad | \quad \text{if} (C) S \text{ else } S \\
 \quad | \quad \text{while} (C) S \\
 \quad | \quad f(E, \dots, E); \\
 \quad | \quad \text{printf}("%d\n", E); \\
 \quad | \quad \text{int } x, \dots, x; \\
 \quad | \quad B \\
 \\
 B \rightarrow \{ S \dots S \} \\
 \\
 F \rightarrow \text{void } f(X, \dots, X) B \\
 \\
 X \rightarrow \text{int } x \\
 \quad | \quad \text{int } \&x \\
 \\
 P \rightarrow F \dots F \\
 \quad \text{int main() } B
 \end{array}$$

```
void fib(int n, int &r) {
    if (n <= 1)
        r = n;
    else {
        int tmp;
        fib(n - 2, tmp);
        fib(n - 1, r);
        r = r + tmp;
    }
}

int main() {
    int f;
    fib(10, f);
    printf("%d\n", f);
}
```

la **portée** définit les portions du programme où une variable est visible

ici, si le corps d'une fonction f mentionne une variable x alors

- soit x est un paramètre de f
- soit x est déclarée plus haut dans un bloc englobant (y compris le bloc courant)

par ailleurs, une variable peut en cacher une autre

```
void f(int n) {
    printf("%d\n", n); // affiche 34
    if (n > 0) {
        int n; n = 89;
        printf("%d\n", n); // affiche 89
    }
    if (n > 21) {
        printf("%d\n", n); // affiche 34
        int n; n = 55;
        printf("%d\n", n); // affiche 55
    }
    printf("%d\n", n); // affiche 34
}

int main() {
    f(34);
}
```

ici la portée ne dépend que du texte source (on parle de **portée lexicale**)
et on peut la réaliser avant ou pendant le typage

la syntaxe abstraite conserve une trace de cette analyse,
en identifiant chaque variable de façon unique

avant

arbres de syntaxe abstraite issus
de l'analyse syntaxique

```
type pint_expr =
  | PEconst of int
  | PEvar   of string
  | ...
type pstmt =
  | PSvars   of string list
  | PSblock of pstmt list
  | ...
```

pour l'instant, les variables sont
des chaînes de caractères

après

arbres de syntaxe abstraite après
le typage

```
type int_expr =
  | Econst of int
  | Evar   of ident
  | ...
type func = {
  locals: ident list;
  ...
```

maintenant `ident` est un
identifiant : entier, nom unique,
enregistrement, etc.

on a maintenant un arbre de syntaxe abstraite qui correspond à

```
void f(int n0) {  
    printf("%d\n", n0);  
    if (n0 > 0) {  
        int n1; n1 = 89;  
        printf("%d\n", n1);  
    }  
    if (n0 > 21) {  
        printf("%d\n", n0);  
        int n2; n2 = 55;  
        printf("%d\n", n2);  
    }  
    printf("%d\n", n0);  
}
```

il existe des langages où la portée est **dynamique** i.e. dépend de l'exécution du programme

exemple : `bash`

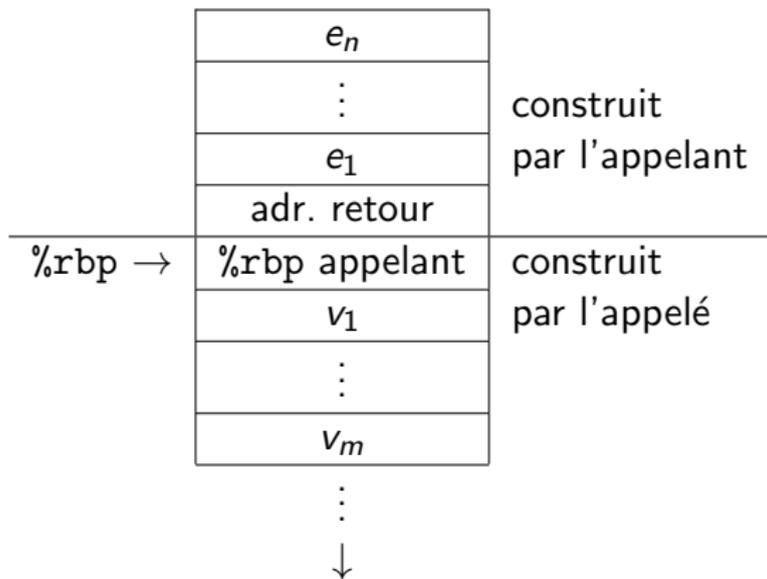
il faut choisir un emplacement mémoire pour chaque variable et être capable de **calculer** cet emplacement à l'exécution

ici les variables vont toutes être stockées sur la pile

à chaque fonction en cours d'exécution correspond une portion de la pile, appelée **tableau d'activation** (cf cours 1), qui contient notamment

- ses paramètres effectifs
- l'adresse de retour
- ses variables locales

tableau d'activation correspondant à un appel $f(e_1, \dots, e_n)$ d'une fonction f avec n paramètres



```

void g(int a, int b) {
    if (...) {
        int c;
        ...
    }
    if (...) {
        int d;
        ...
        int e;
        ...
    }
}

int main() {
    g(100, 10);
}

```

b	10
a	100
	adr. retour
%rbp →	%rbp appelant
c, d	...
e	...

positionner ainsi le registre `%rbp` permet de retrouver facilement l'emplacement d'une variable (par ex. `%rbp + 16` ou `%rbp - 8`)

en effet, le sommet de pile peut bouger si

- on y stocke des calculs intermédiaires
- on est en train de préparer un appel de fonction

pour chaque variable, le compilateur détermine une position dans la pile

par exemple dans le type `ident`

```
type ident = { offset: int; ... }
```

- pour les paramètres, ce sont +16, +24, etc.
- pour les variables locales, ce sont -8, -16, etc., avec souvent plusieurs solutions possibles, certaines plus économes

détaillons maintenant la production d'assembleur x86-64 pour micro C++
on se limite pour commencer au **passage par valeur**

pour produire du code x86-64, on utilise le module OCaml X86_64 fourni sur la page du cours

avec ce module, on écrit par exemple

```
movq (imm 42) (reg rdi)
```

pour construire l'instruction assembleur

```
movq $42, %rdi
```

et on concatène les morceaux d'assembleur avec une opération ++

on suit un schéma de compilation simpliste, utilisant la pile pour stocker les résultats intermédiaires (on verra comment utiliser efficacement les registres aux cours 10–11)

écrivons une fonction `int_expr` qui compile une expression arithmétique

```
val int_expr: int_expr -> X86_64.text
```

principe : à l'issue de l'exécution de `int_expr e`, la valeur de l'expression `e` se trouve dans le registre `%rdi` (choix arbitraire)

on commence par les constantes entières

```
let rec int_expr = function
  | Econst n ->
      movq (imm n) (reg rdi)
```

et les opérations arithmétiques

```
| Ebinop (Badd, e1, e2) ->
    int_expr e1 ++
    pushq (reg rdi) ++
    int_expr e2 ++
    popq rsi ++
    addq (reg rsi) (reg rdi)
| Ebinop (Bsub, e1, e2) ->
    ...
```

bien entendu, c'est extrêmement naïf; le code pour 1+2 est

```
movq  $1, %rdi
pushq %rdi
movq  $2, %rdi
popq  %rsi
addq  %rsi, %rdi
```

alors même que l'on dispose de 16 registres

pour une **variable**, on utilise l'adressage indirect, car la position par rapport à `%rbp` est une constante connue du compilateur

```
| Evar { offset = ofs } ->  
    movq (ind ~ofs rbp) (reg rdi)
```

(rappel : on se limite pour l'instant au passage par valeur)

de même, les expressions booléennes sont compilées avec une fonction

```
val bool_expr: bool_expr -> X86_64.text
```

d'une manière très analogue

```
let rec bool_expr = function
| Bcmp (Beq, e1, e2) ->
  int_expr e1 ++ pushq (reg rdi) ++
  int_expr e2 ++ popq rsi ++
  cmpq (reg rdi) (reg rsi) ++
  sete (reg dil) ++ movzbq (reg dil) rdi
| ...
  (* laissé en exercice *) ...
```

attention : les opérateurs `&&` et `||` doivent être évalués paresseusement
i.e. e_2 n'est pas évaluée dans $e_1 \ \&\& \ e_2$ (resp. $e_1 \ || \ e_2$) si e_1 vaut `false`
 (resp. `true`)

les instructions sont compilées avec une fonction

```
val stmt: stmt -> X86_64.text
```

```
let rec stmt = function  
  | Sprintf e ->  
    int_expr e ++ call "print_int"
```

avec

```
print_int: # (en pratique, il faut aussi aligner la pile)  
  movq %rdi, %rsi  
  movq $.Sprintf_int, %rdi  
  movq $0, %rax  
  call printf  
  ret  
  
.data  
.Sprintf_int:  
  .string "%d\n"
```

```
| Sif (e, s1, s2) ->  
    (* laissé en exercice *)  
  
| Swhile (e, s) ->  
    (* laissé en exercice *)  
  
| Sblock sl ->  
    List.fold_left  
        (fun code s -> code ++ stmt s) nop sl
```

pour un appel à une fonction f , il faut

1. empiler les arguments
2. appeler le code situé à l'étiquette f
3. dépiler les arguments

```
| Scall (id, el) ->
  List.fold_left
    (fun acc e -> int_expr e ++ pushq (reg rdi) ++ acc)
    nop el ++
  call (symb id) ++
  addq (imm (8 * List.length el)) (reg rsp)
```

reste l'**affectation** $x = e;$

le membre gauche est ici réduit à une variable x
et on sait où cette variable est stockée sur la pile

```
| Sassign ({ offset = ofs }, e) ->  
  int_expr e ++  
  movq (reg rdi) (ind ~ofs rbp)
```

pour l'instant, on a passé les paramètres **par valeur**

i.e. le paramètre formel est une **nouvelle variable** qui prend comme valeur initiale celle du paramètre effectif

en C++, le qualificatif **&** permet de spécifier un passage **par référence**

dans ce cas, le paramètre formel désigne la **même variable** que le paramètre effectif, qui doit donc être une variable (une valeur gauche, de manière plus générale)

```
void fib(int n, int &r) {
    if (n <= 1)
        r = n;
    else {
        int tmp;
        fib(n - 2, tmp);
        fib(n - 1, r);
        r = r + tmp;
    }
}

int main() {
    int f;
    fib(10, f);          // modifie la valeur de f
    printf("%d\n", f);  // affiche 55
}
```

pour prendre en compte le passage par référence, on étend encore le type `ident` pour indiquer s'il s'agit d'une variable passée par référence

```
type ident = { offset: int; by_reference: bool; ... }
```

(vaut `false` pour une variable locale)

dans un appel tel que $f(e)$ le paramètre effectif e n'est plus typé ni compilé de la même manière selon qu'il s'agit d'un paramètre passé par valeur ou par référence

lorsque le paramètre est passé par référence, le typage va donc

1. vérifier qu'il s'agit bien d'une valeur gauche (une variable ici)
2. indiquer qu'elle doit être passée par référence

une façon de procéder consiste à ajouter une construction de « calcul de valeur gauche » dans la syntaxe des expressions

```
type int_expr =  
  ...  
  | Eaddr of ident
```

et à remplacer, le cas échéant, le paramètre effectif `e` par `Eaddr e`

note : c'est l'opérateur `&` de C++, qui n'est pas dans notre fragment

il faut ajouter le code correspondant dans `int_expr` :

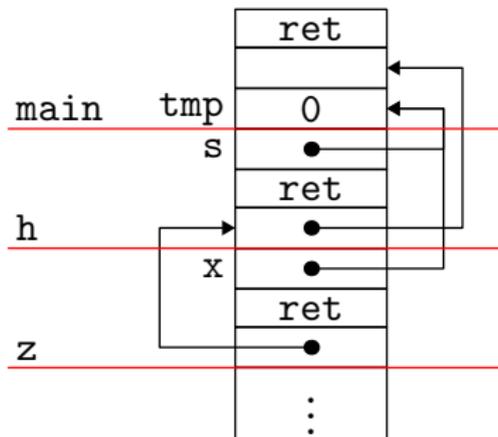
```
let rec int_expr = function
  | Eaddr { offset = ofs; by_reference = br } ->
    leaq (ind ~ofs rbp) rdi ++
    if br then movq (ind rdi) (reg rdi) else nop
```

note : le cas `br = true` correspond au cas d'une variable elle-même passée par référence

```

void z(int &x) { x = 0; }
void h(int &s) { z(s); while (s < 100) s = 2*s+1; }
int main() { int tmp; h(tmp); printf("%d\n", tmp); }

```



il faut aussi modifier le calcul des valeurs droites :

```
| Evar { offset = ofs; by_reference = br } ->  
  movq (ind ~ofs rbp) (reg rdi) ++  
  if br then movq (ind rdi) (reg rdi) else nop
```

ainsi que celui de l'affectation :

```
| Sassign ({ offset = ofs; by_reference = br}, e) ->  
  int_expr e ++  
  (if br then movq (ind ~ofs rbp) (reg rsi)  
   else leaq (ind ~ofs rbp) rsi) ++  
  movq (reg rdi) (ind rsi)
```

en revanche, il n'y a rien à modifier dans l'appel (grâce à la nouvelle construction Eaddr)

il reste à compiler les déclarations des fonctions

```
type function_decl =  
  { fname   : string;  
    formals: ident list;  
    locals  : ident list;  
    body    : stmt; }
```

```
let function_decl f =  
  let fs =  
    List.fold_left (fun fs x -> max fs (abs x.offset))  
      0 f.locals in
```

```
f:  
  pushq %rbp          # sauvegarde %rbp  
  movq  %rsp, %rbp    # le positionne  
  subq  $fs, %rsp     # alloue fs octets
```

```
++ stmt f.body ++
```

```
  movq  %rbp, %rsp    # désalloue le tableau  
  popq  %rbp          # restaure %rbp  
  ret                               # retour à l'appelant
```

```
void swap(int &x, int &y) {
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

y (+24)	
x (+16)	
	adr. retour
%rbp →	%rbp appelant
tmp (-8)	...

```
swap:  pushq %rbp
      movq %rsp, %rbp
      subq $8, %rsp
      movq 16(%rbp), %rdi
      movq 0(%rdi), %rdi
      leaq -8(%rbp), %rsi
      movq %rdi, 0(%rsi)
      movq 24(%rbp), %rdi
      movq 0(%rdi), %rdi
      movq 16(%rbp), %rsi
      movq %rdi, 0(%rsi)
      movq -8(%rbp), %rdi
      movq 24(%rbp), %rsi
      movq %rdi, 0(%rsi)
      movq %rbp, %rsp
      popq %rbp
      ret
```

correction de la compilation

le compilateur doit respecter la **sémantique** du langage (correction)

si le langage source est muni d'une sémantique \rightarrow_s et le langage machine d'une sémantique \rightarrow_m , et si l'expression e est compilée en $C(e)$ alors on doit avoir un « diagramme qui commute » :

$$\begin{array}{ccc}
 e & \xrightarrow{*}_s & v \\
 \downarrow & & \approx \\
 C(e) & \xrightarrow{*}_m & v'
 \end{array}$$

où $v \approx v'$ exprime que les valeurs v et v' coïncident

considérons uniquement les expressions arithmétiques sans variable

$$e ::= n \mid e + e$$

et montrons la correction de la compilation

on se donne une sémantique à réductions pour le langage source

$$\begin{aligned} v & ::= n \\ E & ::= \square \mid E + e \mid v + E \end{aligned}$$

$$n_1 + n_2 \xrightarrow{\epsilon} n \quad \text{avec } n = n_1 + n_2$$

on se donne de même une sémantique à réductions pour le langage cible

$$\begin{aligned}
 m & ::= \text{movq } \$n, r \\
 & \quad | \text{addq } \$n, r \mid \text{addq } r, r \\
 & \quad | \text{movq } (r), r \mid \text{movq } r, (r) \mid \\
 r & ::= \%rdi \mid \%rsi \mid \%rsp
 \end{aligned}$$

un état est la donnée de valeurs pour les registres, R ,
et d'un état de la mémoire, M

$$\begin{aligned}
 R & ::= \{\%rdi \mapsto n; \%rsi \mapsto n; \%rsp \mapsto n\} \\
 M & ::= \mathbb{N} \rightarrow \mathbb{Z}
 \end{aligned}$$

on définit la sémantique d'une instruction m par une réduction

$$R, M, m \xrightarrow{m} R', M'$$

la réduction $R, M, m \xrightarrow{m} R', M'$ est définie par

$$R, M, \text{movq } \$n, r \xrightarrow{m} R\{r \mapsto n\}, M$$

$$R, M, \text{addq } \$n, r \xrightarrow{m} R\{r \mapsto R(r) + n\}, M$$

$$R, M, \text{addq } r_1, r_2 \xrightarrow{m} R\{r_2 \mapsto R(r_1) + R(r_2)\}, M$$

$$R, M, \text{movq } (r_1), r_2 \xrightarrow{m} R\{r_2 \mapsto M(R(r_1))\}, M$$

$$R, M, \text{movq } r_1, (r_2) \xrightarrow{m} R, M\{R(r_2) \mapsto R(r_1)\}$$

$code(n) = \text{movq } \$n, \%rdi$

$code(e_1 + e_2) = code(e_1)$
 $\text{addq } \$ - 8, \%rsp$
 $\text{movq } \%rdi, (\%rsp)$
 $code(e_2)$
 $\text{movq } (\%rsp), \%rsi$
 $\text{addq } \$8, \%rsp$
 $\text{addq } \%rsi, \%rdi$

on souhaite montrer que si

$$e \xrightarrow{*} n$$

et si

$$R, M, \text{code}(e) \xrightarrow{m}^* R', M'$$

alors $R'(\%rdi) = n$

on procède par récurrence structurelle sur e

on établit un résultat plus fort (**invariant**), à savoir :

si $e \xrightarrow{*} n$ et $R, M, \text{code}(e) \xrightarrow{m,*} R', M'$ alors

$$\left\{ \begin{array}{l} R'(\%rdi) = n \\ R'(\%rsp) = R(\%rsp) \\ \forall a \geq R(\%rsp), M'(a) = M(a) \end{array} \right.$$

- cas $e = n$

on a $e \xrightarrow{*} n$ et $code(e) = \text{movq } \$n, \%rdi$ et le résultat est immédiat

- cas $e = e_1 + e_2$

on a $e \xrightarrow{*} n_1 + e_2 \xrightarrow{*} n_1 + n_2$ avec $e_1 \xrightarrow{*} n_1$ et $e_2 \xrightarrow{*} n_2$

ce qui nous permet d'invoquer l'hypothèse de récurrence sur e_1 et e_2

	R, M	
$code(e_1)$	R_1, M_1	par hypothèse de récurrence $R_1(\%rdi) = n_1$ et $R_1(\%rsp) = R(\%rsp)$ $\forall a \geq R(\%rsp), M_1(a) = M(a)$
addq \$-8,%rsp movq %rdi,(%rsp)	R'_1, M'_1	$R'_1 = R_1\{\%rsp \mapsto R(\%rsp) - 8\}$ $M'_1 = M_1\{R(\%rsp) - 8 \mapsto n_1\}$
$code(e_2)$	R_2, M_2	par hypothèse de récurrence $R_2(\%rdi) = n_2$ et $R_2(\%rsp) = R(\%rsp) - 8$ $\forall a \geq R(\%rsp) - 8, M_2(a) = M'_1(a)$
movq (%rsp),%rsi addq \$8,%rsp addq %rsi,%rdi	R', M_2	$R'(\%rdi) = n_1 + n_2$ $R'(\%rsp) = R(\%rsp) - 8 + 8 = R(\%rsp)$ $\forall a \geq R(\%rsp),$ $M_2(a) = M'_1(a) = M_1(a) = M(a)$

une telle preuve peut être effectuée sur un vrai compilateur

exemple : CompCert, un compilateur C produisant du code efficace pour PowerPC, ARM, RISC-V et x86, a été formellement vérifié avec l'assistant de preuve Coq

cf <http://compcert.inria.fr/>

- TD 7
 - aide au projet
- prochain cours
 - compilation des langages fonctionnels