École Polytechnique

INF564 – Compilation

Jean-Christophe Filliâtre

compilation des langages objets
et des langages fonctionnels

aujourd'hui

1. langages objets
   - représentation d'un objet
   - appel dynamique

2. langages fonctionnels
   - fonctions de première classe
   - optimisation de l'appel terminal

compilation des langages objets

expliquons

- comment un objet est représenté
- comment est réalisé l'appel d'une méthode

en prenant le cas de Java (pour l'instant)

exemple

```
class Vehicle {
  static int start = 10;
  int position;
  Vehicle() { position = start; }
  void move(int d) { position += d; } }
```

```
class Car extends Vehicle {
  int passengers;
  void await(Vehicle v) {
    if (v.position < position)
      v.move(position - v.position);
    else
      move(10); } }
```

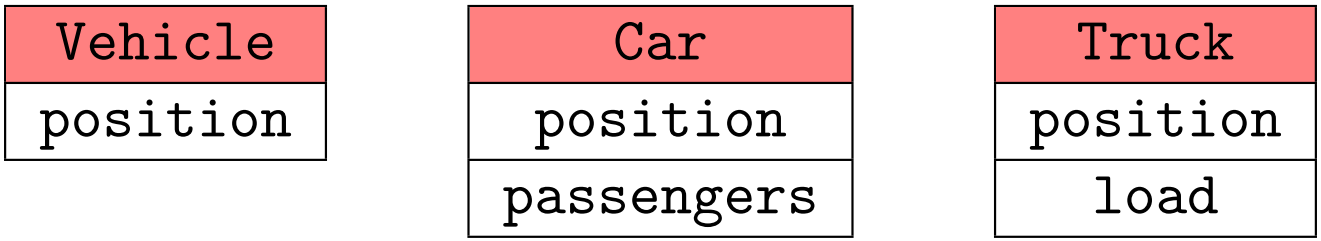
```
class Truck extends Vehicle {
  int load;
  void move(int d) {
    if (d <= 55) position += d; else position += 55; } }
```

un objet est un bloc alloué sur le tas, contenant

- sa classe
- les valeurs de ses champs

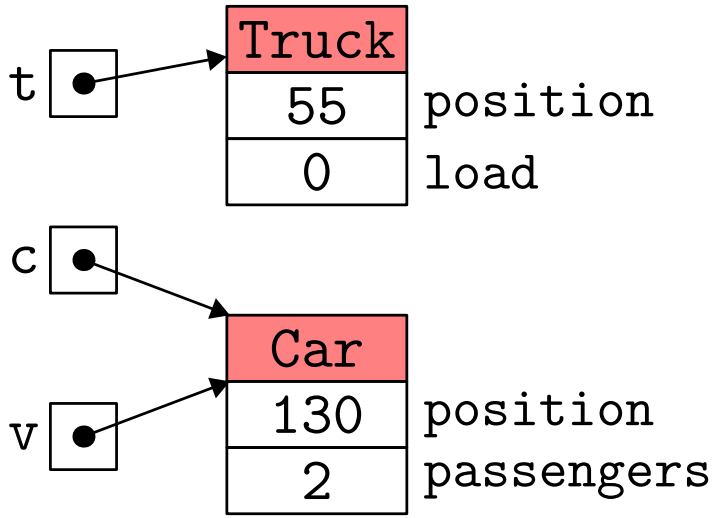
la valeur d'un objet est le pointeur vers le bloc

l'héritage **simple** permet de stocker la valeur d'un champ x à un emplacement constant dans le bloc : les champs propres viennent après les champs hérités



noter l'absence du champ `start`, qui est statique donc alloué ailleurs (par exemple dans le segment de données)

```
Truck t = new Truck();
Car c = new Car();
c.passengers = 2;
c.move(60);
Vehicle v = c;
v.move(70);
c.await(t);
```



pour chaque champ, le compilateur connaît la position où ce champ est rangé, c'est-à-dire le décalage à ajouter au pointeur sur l'objet

si par exemple le champ `position` est rangé à la position +16 alors l'expression `e.position` est compilée comme

```
...                     # on compile e dans %rcx
movl 16(%rcx), %eax     # champ position
```

ceci est correct, alors que le compilateur ne connaît **que le type statique** de `e`, qui peut être différent du type dynamique (la classe de l'objet)

il pourrait même s'agir d'une sous-classe de `Vehicule` non encore définie !

la **redéfinition** (en anglais **overriding**) est la possibilité de redéfinir une méthode dans une sous-classe
(de manière à ce que des objets différents se comportent différemment)

exemple : dans la classe `Truck`

```
class Truck extends Vehicle {
  ...
  void move(int d) { ... }
}
```

la méthode `move`, héritée de la classe `Vehicle`, est redéfinie

toute la subtilité de la compilation des langages à objets est dans l'**appel d'une méthode dynamique** $e.m(e_1, \ldots, e_n)$

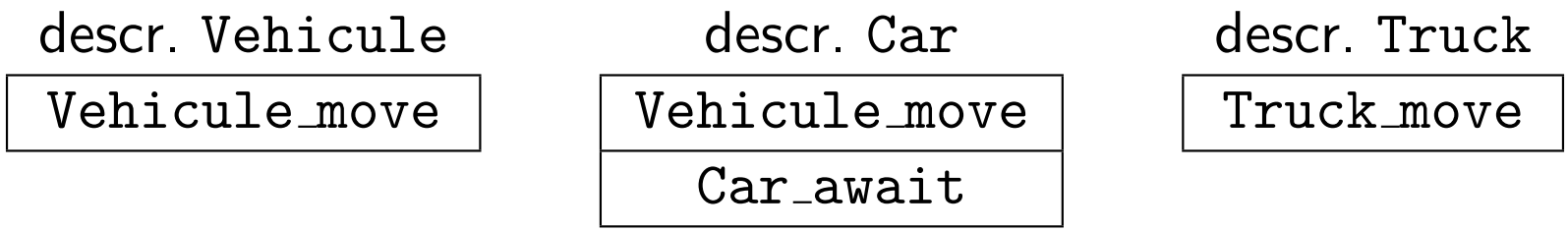
pour cela, on construit pour chaque classe un **descripteur de classe** qui contient les adresses des codes de méthodes dynamiques de cette classe (en anglais **dispatch table**, **vtable**, etc.)

comme pour les champs, l'héritage simple permet de ranger l'adresse du code de la méthode m à un emplacement constant dans le descripteur

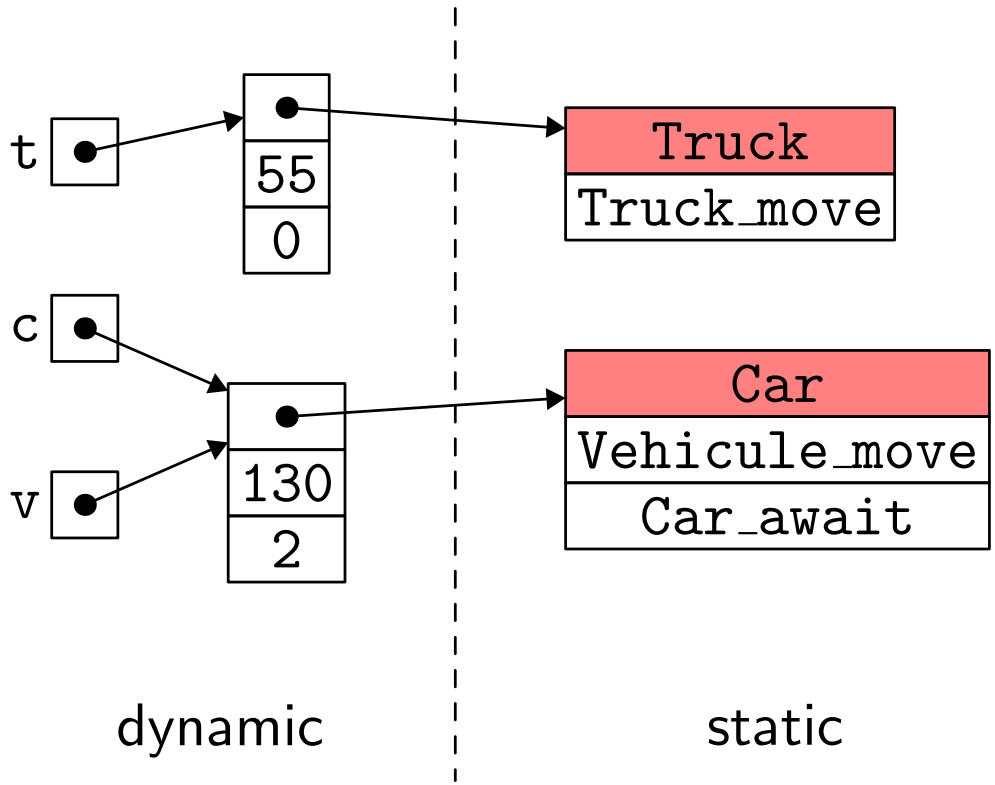
les descripteurs de classes peuvent être alloués dans le segment de données ; chaque objet contient dans son premier champ un pointeur vers le descripteur de sa classe

exemple

```
class Vehicule                  { void move(int d) {...} }
class Car   extends Vehicule { void await(Vehicule v) {...}}
class Truck extends Vehicule { void move(int d) {...} }
```



```
Truck t = new Truck();
Car c = new Car();
c.passengers = 2;
c.move(60);
Vehicle v = c;
v.move(70);
c.await(t);
```



appel de méthode

pour compiler un appel comme `e.move(10)`

1. on compile `e` ; sa valeur est un pointeur vers un objet
2. cet objet contient un pointeur vers le descripteur de sa classe
3. le code de la méthode `move` est situé à un emplacement connu (par exemple +8) dans ce descripteur

```
...                     # compiler e dans %rdi
movq $10, %rsi          # argument
movq (%rdi), %rcx       # descripteur de class
call *8(%rcx)           # méthode move
```

comme pour l'accès au champ, à aucun moment on n'a eu besoin de connaître la classe effective de l'objet (son type dynamique)

si on écrit

```
Truck v = new Truck();
((Vehicule)v).move();
```

c'est toujours la méthode `move` de `Truck` qui est appelée
car l'appel de méthode reste compilé de la même façon

le transtypage n'a ici qu'une influence au moment du typage
(existence de la méthode + résolution de la surcharge ; cf cours 4)

en pratique, le descripteur de la classe C contient également l'indication de la classe dont C hérite, appelée **super classe** de C

la super classe est représentée par un pointeur vers son descripteur (qu'on peut ranger dans le premier champ du descripteur, par exemple)

cela permet entre autres de compiler le test dynamique derrière un *downcast* ou un `instanceof`

quelques mots sur C++

on reprend l'exemple des véhicules

```
class Vehicle {
  static const int start = 10;
public:
  int position;
  Vehicle() { position = start; }
  virtual void move(int d) { position += d; }
};
```

`virtual` signifie que la méthode `move` pourra être redéfinie

```
class Car : public Vehicle {
public:
  int passengers;
  Car() {}
  void await(Vehicle &v) { // passage par référence
    if (v.position < position)
      v.move(position - v.position);
    else
      move(10);
  }
};
```

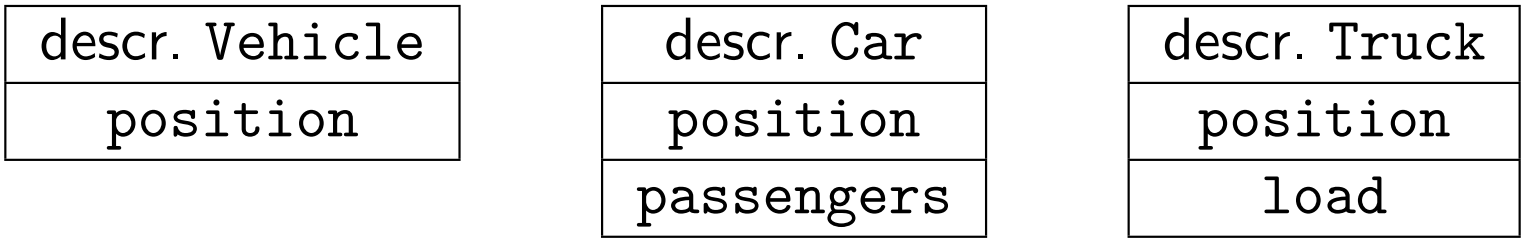
exemple (suite)

```
class Truck : public Vehicle {
public:
  int load;
  Truck() {}
  void move(int d) {
    if (d <= 55) position += d; else position += 55;
  }
};
```

exemple (suite)

```
#include <iostream>
using namespace std;

int main() {
  Truck t; // objets alloués ici sur la pile
  Car c;
  c.passengers = 2;
  c.move(60);
  Vehicle *v = &c; // alias
  v->move(70);
  c.await(t);
}
```

sur cet exemple, la représentation d'un objet n'est pas différente de Java



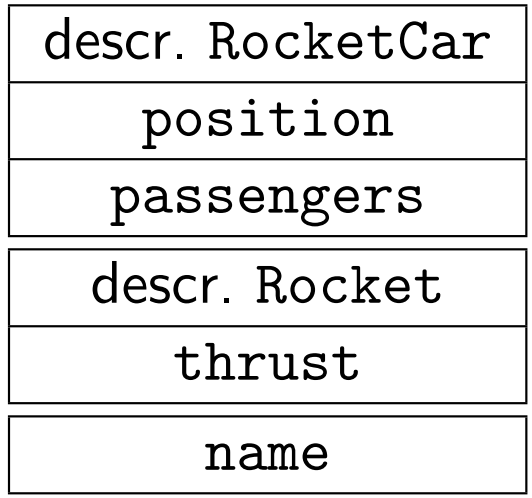
mais en C++, on trouve aussi de l'**héritage multiple**

conséquence : on ne peut plus (toujours) utiliser le principe selon lequel

- la représentation d'un objet d'une super classe de C est un préfixe de la représentation d'un objet de la classe C
- de même pour les descripteurs de classes

héritage multiple

```
class Rocket {
public:
  float thrust;
  Rocket() { }
  virtual void display() {}
};

class RocketCar : public Car, public Rocket {
public:
  char *name;
  void move(int d) { position += 2*d; }
};
```



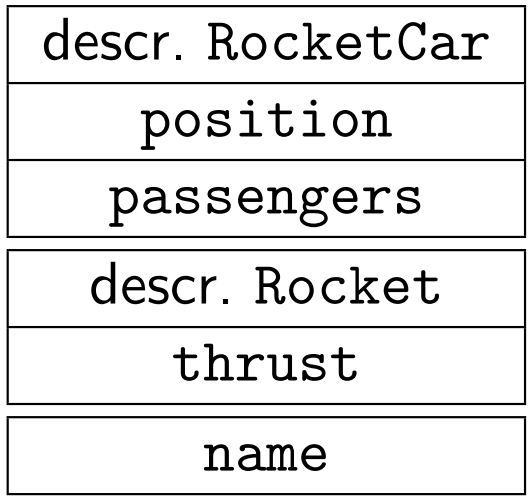
les représentations de Car et Rocket sont juxtaposées

en particulier, un transtypage comme

```
RocketCar rc;
... (Rocket)rc ...
```

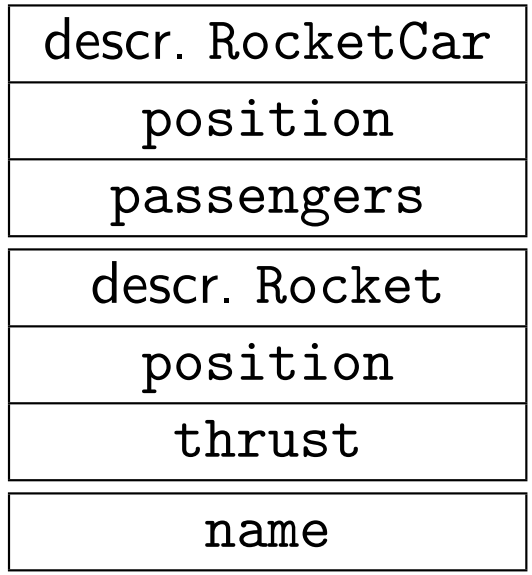
est traduit par une arithmétique de pointeur

```
... rc + 12 ...
```



supposons maintenant que `Rocket` hérite également de `Vehicle`

```
class Rocket : public Vehicle {
public:
  float thrust;
  Rocket() { }
  virtual void display() {}
};

class RocketCar : public Car, public Rocket {
public:
  char *name;
  ...
};
```



on a maintenant **deux** champs `position`

et donc une ambiguïté potentielle

```
class RocketCar : public Car, public Rocket {
public:
  char *name;
  void move(int d) { position += 2*d; }
};
```

```
vehicles.cc: In member function ‘virtual void RocketCar::move(int)’
vehicles.cc:51:22: error: reference to ‘position’ is ambiguous
```

il faut préciser de quel champ `position` il s'agit

```
class RocketCar : public Car, public Rocket {
public:
  char *name;
  void move(int d) { Rocket::position += 2*d; }
};
```

pour n'avoir qu'une seule instance de `Vehicle`, il faut modifier la façon dont `Car` et `Rocket` héritent de `Vehicle` (héritage virtuel)

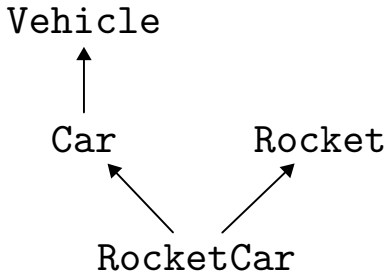
```
class Vehicle { ... };

class Car : public virtual Vehicle { ... };

class Rocket : public virtual Vehicle { ... };

class RocketCar : public Car, public Rocket {
```

il n'y a plus d'ambiguïté quant à `position` :

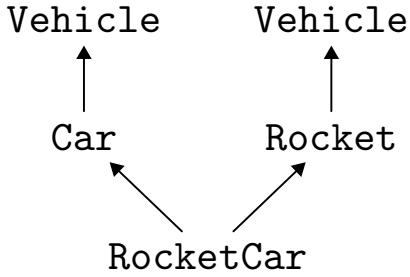
```
public:
  char *name;
  void move(int d) { position += 2*d; }
};
```

trois situations différentes

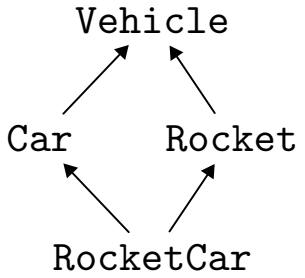
```
class Vehicle { ... };
class Car : Vehicle { ... };
class Rocket { ... };
class RocketCar : Car, Rocket { ... };
```




```
class Vehicle { ... };
class Car : Vehicle { ... };
class Rocket : Vehicle { ... };
class RocketCar : Car, Rocket { ... };
```




```
class Vehicle { ... };
class Car : virtual Vehicle { ... };
class Rocket : virtual Vehicle { ... };
class RocketCar : Car, Rocket { ... };
```



(le diamant)

si vous êtes curieux

l'option `-fdump-lang-class` de `g++` produit un fichier texte contenant une description des tables de méthodes et des représentations des objets

les interfaces de Java compliquent la compilation d'un appel de méthode, de façon analogue à l'héritage multiple

```
interface I {
  void m();
}

class C {
  void foo(I x) { x.m(); }
}
```

la compilation de `x.m()` ne peut préjuger de la classe qu'aura effectivement cet objet `x`

plutôt que d'utiliser uniquement le type de l'objet pour l'appel dynamique, on peut utiliser les types de **tous** les paramètres

on parle alors de multiméthode (**multiple dispatch** en anglais)

un exemple : Julia, un langage orienté mathématiques

```
function +(x::Int64  , y::Int64  ) ... end
function +(x::Float64, y::Float64) ... end
function +(x::Date   , y::Time   ) ... end
```

un autre exemple : CLOS (Common Lisp Object System)

le **filtrage**, comme celui d'OCaml, par ex.

```
let rec eval = function
| Const n -> ...
| Call ("print", [e]) -> ...
| Call (f, el) -> ...
```

est une forme d'appel dynamique : la branche est sélectionnée à l'exécution à partir d'une information dynamique

le polycopié (section 7.3) explique comment le compilateur transforme le filtrage en opérations élémentaires

voir aussi la comparaison fonctionnel/objet du cours 2

compilation des langages fonctionnels

fonctions comme valeurs de première classe

ce qui fait la particularité d'un langage fonctionnel, c'est de pouvoir manipuler les fonctions **comme des valeurs de première classe** c'est-à-dire exactement comme des valeurs d'un autre type

ainsi, on peut

- recevoir une fonction en argument
- renvoyer une fonction comme résultat
- stocker une fonction dans une structure de données
- construire de nouvelles fonctions dynamiquement

la possibilité d'imbriquer des fonctions ou de passer des fonctions en arguments existe depuis longtemps (Algol, Pascal, Ada, etc.)

de même, la notion de pointeur de fonction existe depuis longtemps (Fortran, C, C++, etc.)

ce que les langages fonctionnels proposent est strictement plus puissant

illustrons-le avec OCaml

considérons un fragment minimal d'OCaml

$$
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & \texttt{fun}\ x \rightarrow e \\
& | & e\ e \\
& | & \texttt{let}\ [\texttt{rec}]\ x = e\ \texttt{in}\ e \\
& | & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
\\
d & ::= & \texttt{let}\ [\texttt{rec}]\ x = e \\
\\
p & ::= & d\ \ldots\ d
\end{array}
$$

les fonctions peuvent être imbriquées

```
let sum n =
  let f x = x * x in
  let rec loop i =
    if i = n then 0 else f i + loop (i+1)
  in
  loop 0
```

la portée statique est usuelle

(on écrit `let f x = x * x` pour `let f = fun x -> x * x`)

il est également possible de prendre des fonctions en argument

```
let square f x =
  f (f x)
```

et d'en renvoyer

```
let f x =
  if x < 0 then fun y -> y - x else fun y -> y + x
```

dans ce dernier cas, la valeur renvoyée par `f` est une fonction qui utilise `x` mais le tableau d'activation de `f` vient précisément de disparaître !

on ne peut donc pas compiler les fonctions de manière traditionnelle

la solution consiste à utiliser une **fermeture** (en anglais *closure*)

c'est une structure de données allouée sur le tas (pour survivre aux appels de fonctions) contenant

- un **pointeur vers le code** de la fonction à appeler
- les valeurs des variables susceptibles d'être utilisées par ce code ; cette partie s'appelle l'**environnement**

P. J. Landin, *The Mechanical Evaluation of Expressions*,
The Computer Journal, 1964

quelles sont justement les variables qu'il faut mettre dans l'environnement de la fermeture représentant $\texttt{fun}\ x \to e$?

ce sont les **variables libres** de $\texttt{fun}\ x \to e$

l'ensemble des variables libres d'une expression se calcule ainsi

$$
\begin{aligned}
fv(c) &= \emptyset \\
fv(x) &= \{x\} \\
fv(\texttt{fun}\ x \to e) &= fv(e) \setminus \{x\} \\
fv(e_1\ e_2) &= fv(e_1) \cup fv(e_2) \\
fv(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) &= fv(e_1) \cup (fv(e_2) \setminus \{x\}) \\
fv(\texttt{let rec}\ x = e_1\ \texttt{in}\ e_2) &= (fv(e_1) \cup fv(e_2)) \setminus \{x\} \\
fv(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) &= fv(e_1) \cup fv(e_2) \cup fv(e_3)
\end{aligned}
$$

considérons le programme suivant qui approxime $\int_0^1 x^n dx$

```
let rec pow i x = if i = 0 then 1. else x *. pow (i-1) x

let integrate_xn n =
  let f = pow n in
  let eps = 0.001 in
  let rec sum x =
    if x >= 1. then 0. else f x +. sum (x +. eps) in
  sum 0. *. eps
```

faisons apparaître la construction `fun` explicitement et examinons les différentes fermetures

```
let rec pow =
  fun i ->
    fun x -> if i = 0 then 1. else x *. pow (i-1) x
```

- dans la première fermeture, `fun i ->`, l'environnement est $\{\texttt{pow}\}$
- dans la seconde, `fun x ->`, il vaut $\{\texttt{i}, \texttt{pow}\}$

```
let integrate_xn = fun n ->
  let f = pow n in
  let eps = 0.001 in
  let rec sum =
    fun x -> if x >= 1. then 0. else f x +. sum (x+.eps) in
  sum 0. *. eps
```

- pour `fun n ->`, l'environnement vaut $\{\texttt{pow}\}$
- pour `fun x ->`, l'environnement vaut $\{\texttt{eps}, \texttt{f}, \texttt{sum}\}$

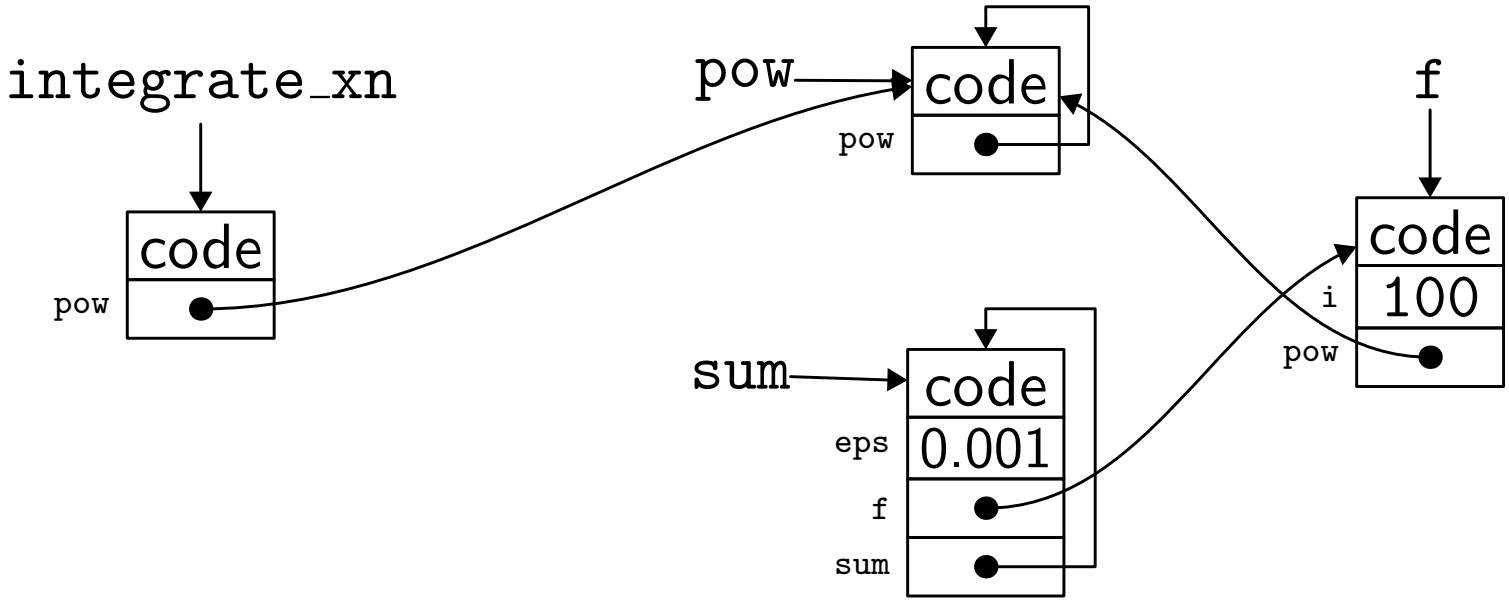
représentation de la fermeture

la fermeture est un bloc alloué sur le tas, dont

- le premier champ contient l'adresse du code
- les champs suivants contiennent les valeurs des variables libres (l'environnement)

```
let rec pow i x = if i = 0 then 1. else x *. pow (i-1) x
let integrate_xn n =
  let f = pow n in
  let eps = 0.001 in
  let rec sum x = if x >= 1. then 0. else f x +. sum (x+.eps) in
  sum 0. *. eps
```

pendant l'exécution de `integrate_xn 100`, on a quatre fermetures :



une façon relativement simple de compiler les fermetures consiste à procéder en deux temps

1. on recherche dans le code toutes les constructions `fun` $x \to e$ et on les remplace par une opération explicite de construction de fermeture

   $$\texttt{clos}\ f\ [y_1, \ldots, y_n]$$

   où les y_i sont les variables libres de `fun` $x \to e$ et f le nom donné à une déclaration globale de fonction de la forme

   $$\texttt{letfun}\ f\ [y_1, \ldots, y_n]\ x = e'$$

   où e' est obtenu à partir de e en y supprimant récursivement les constructions `fun` (*closure conversion*)
2. on compile le code obtenu, qui ne contient plus que des déclarations de fonctions de la forme `letfun`

sur l'exemple, cela donne

```
letfun fun2 [i,pow] x =
  if i = 0 then 1. else x *. pow (i-1) x
letfun fun1 [pow] i =
  clos fun2 [i,pow]
let rec pow =
  clos fun1 [pow]
letfun fun3 [eps,f,sum] x =
  if x >= 1. then 0. else f x +. sum (x +. eps)
letfun fun4 [pow] n =
  let f = pow n in
  let eps = 0.001 in
  let rec sum = clos fun3 [eps,f,sum] in
  sum 0. *. eps
let integrate_xn =
  clos fun4 [pow]
```

avant

$$
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & \texttt{fun}\ x \to e \\
& | & e\ e \\
& | & \texttt{let}\ [\texttt{rec}]\ x = e\ \texttt{in}\ e \\
& | & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
\\
d & ::= & \texttt{let}\ [\texttt{rec}]\ x = e \\
\\
p & ::= & d\ \ldots\ d
\end{array}
$$

après

$$
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & \texttt{clos}\ f\ [x, \ldots, x] \\
& | & e\ e \\
& | & \texttt{let}\ [\texttt{rec}]\ x = e\ \texttt{in}\ e \\
& | & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
\\
d & ::= & \texttt{let}\ [\texttt{rec}]\ x = e \\
& | & \texttt{letfun}\ f\ [x, \ldots, x]\ x = e \\
\\
p & ::= & d\ \ldots\ d
\end{array}
$$

en particulier, un identificateur x peut représenter

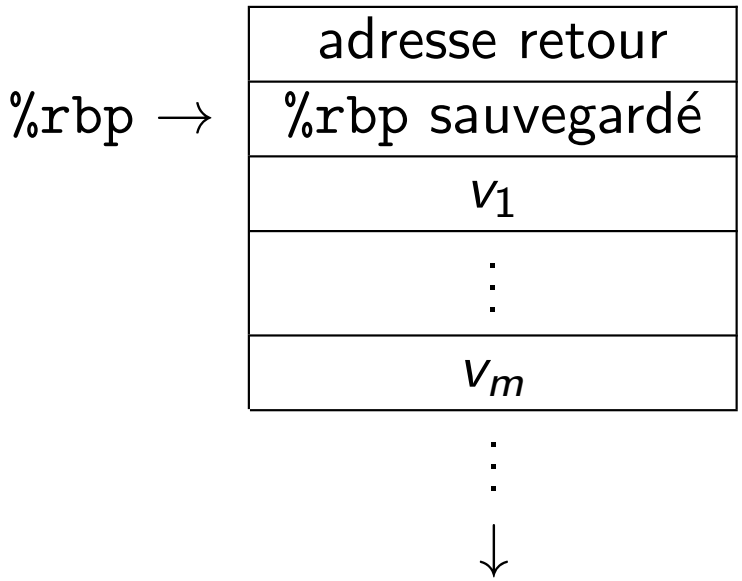
- une **variable globale** introduite par `let`
  (allouée dans le segment de données)
- une **variable locale** introduite par `let in`
  (allouée dans le tableau d'activation / un registre)
- une **variable contenue dans une fermeture**
- l'**argument** d'une fonction (le `x` de `fun x -> e`)

chaque fonction a un unique argument, qu'on passera dans `%rdi`

la fermeture sera passée dans `%rsi`

le tableau d'activation ressemble à ceci,
où $v_1, \ldots, v_m$ sont les variables locales

il est intégralement construit par l'appelé



expliquons maintenant comment compiler

- la construction d'une fermeture `clos` f $[y_1, \ldots, y_n]$
- un appel de fonction e_1 e_2
- l'accès à une variable x
- une déclaration de fonction `letfun` f $[y_1, \ldots, y_n]$ $x = e$

pour compiler la construction

$$\texttt{clos}\ f\ [y_1, \ldots, y_n]$$

on procède ainsi

1. on alloue un bloc de taille $n+1$ sur le tas (avec un GC)
2. on stocke l'adresse de f dans le champ 0
   (f est une étiquette dans le code et on obtient son adresse avec $\$f$)
3. on stocke les valeurs des variables $y_1, \ldots, y_n$ dans les champs 1 à n
4. on renvoie le pointeur sur le bloc

note : on se repose sur un GC pour libérer ce bloc lorsque ce sera possible (le fonctionnement d'un GC sera expliqué au cours 9)

pour compiler un appel de la forme

$$e_1\ e_2$$

on procède ainsi

1. on compile e_1 dans le registre `%rsi`
   (sa valeur est un pointeur p_1 vers une fermeture)
2. on compile e_2 dans le registre `%rdi`
3. on appelle la fonction dont l'adresse est contenue dans le premier champ de la fermeture, avec `call *(%rsi)`

   c'est un saut à une **adresse calculée**

pour compiler un accès à une variable x on distingue quatre cas

variable globale
: la valeur se trouve à l'adresse donnée par l'étiquette x

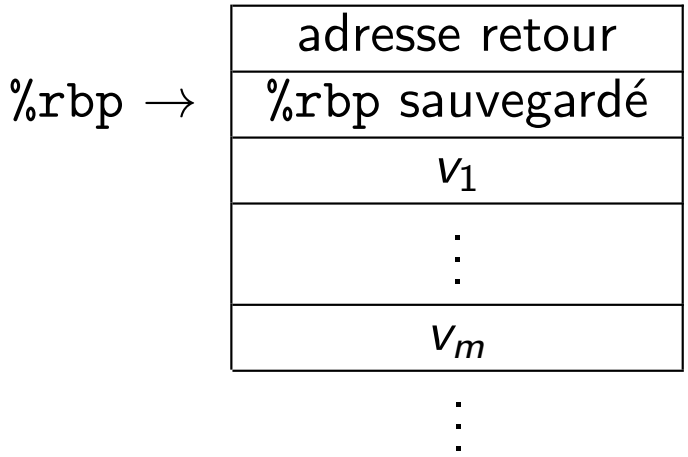
variable locale
: la valeur se trouve en n(`%rbp`) / dans un registre

variable capturée dans une fermeture
: la valeur se trouve en n(`%rsi`)

argument de la fonction
: la valeur se trouve dans `%rdi`

enfin, pour compiler la déclaration

$$\texttt{letfun}\ f\ [y_1, \ldots, y_n]\ x = e$$



on procède comme pour une déclaration usuelle de fonction

1. on y sauvegarde `%rbp` et on positionne `%rbp`
2. on alloue le tableau d'activation (pour les variables locales de e)
3. on évalue e dans `%rax`
4. on désalloue le tableau d'activation et on restaure `%rbp`
5. on exécute `ret`

on trouve aujourd'hui des fermetures dans

- Java (depuis 2014 et Java 8)
- C++ (depuis 2011 et C++11)

dans ces langages, les fonctions anonymes sont appelées des **lambdas**

une fonction est un objet comme un autre, avec une méthode `apply`

```
LinkedList<B> map(LinkedList<A> l, Function<A, B> f) {
  ... f.apply(x) ...
}
```

une fonction anonyme est introduite avec `->`

```
map(l, x -> { System.out.print(x); return x+y; })
```

le compilateur construit un objet fermeture (qui capture ici `y`) avec une méthode `apply`

les fermetures de C++

une fonction anonyme est introduite avec `[]`

```
for_each(v.begin(), v.end(), [y](int &x){ x += y; });
```

on spécifie les variables capturées dans la fermeture (ici `y`)

on peut spécifier une capture par référence (ici de `s`)

```
for_each(v.begin(), v.end(), [y,&s](int x){ s += y*x; });
```

là encore, le compilateur construit une fermeture
(d'un type qui ne nous est pas accessible)

optimisation des appels terminaux

Définition

On dit qu'un **appel** $f(e_1, \ldots, e_n)$ *qui apparaît dans le corps d'une fonction g est* **terminal** *(tail call) si c'est la dernière chose que g calcule avant de renvoyer son résultat.*

par extension, on peut dire qu'une fonction est **récursive terminale** (*tail recursive function*) s'il s'agit d'une fonction récursive dont tous les appels récursifs sont des appels terminaux

l'appel terminal n'est pas nécessairement un appel récursif

```
int g(int x) {
  int y = x * x;
  return f(y);
}
```

dans une fonction récursive, on peut avoir des appels récursifs terminaux et d'autres qui ne le sont pas

```
int f91(int n) {
  if (n > 100) return n - 10;
  return f91(f91(n + 11));
}
```

quel intérêt du point de vue de la compilation ?

on peut détruire le tableau d'activation de la fonction où se trouve l'appel **avant** de faire l'appel, puisqu'il ne servira plus ensuite

mieux, on peut le réutiliser pour l'appel terminal que l'on doit faire
(en particulier, l'adresse de retour sauvegardée y est la bonne)

dit autrement, on peut faire un saut (`jump`) plutôt qu'un appel (`call`)

considérons

```
int fact(int acc, int n) {
  if (n <= 1) return acc;
  return fact(acc * n, n - 1);
}
```

compilation classique

```
fact:   cmpq  $1, %rsi
        jle   L0
        imulq %rsi, %rdi
        decq  %rsi
        call  fact
        ret
L0:     movq  %rdi, %rax
        ret
```

optimisation

```
fact:   cmpq  $1, %rsi
        jle   L0
        imulq %rsi, %rdi
        decq  %rsi
        jmp   fact   # <--

L0:     movq  %rdi, %rax
        ret
```

le résultat est une **boucle**

le code est en effet identique à ce qu'aurait donné la compilation de

```
int fact(int acc, int n) {
  while (n > 1) {
    acc *= n;
    n--;
  }
  return acc;
}
```

le compilateur `gcc` optimise les appels terminaux si on lui passe l'option `-foptimize-sibling-calls` (inclus dans l'option `-O2`)

observons le code produit par `gcc -O2` sur des programmes comme `fact` ou comme ceux du transparent **??**

en particulier, on remarque que le programme

```
int f91(int n) {
  if (n > 100) return n - 10;
  return f91(f91(n + 11));
}
```

est compilé **exactement** comme si on avait écrit

```
int f91(int n) {
  while (n <= 100)
    n = f91(n + 11);
  return n - 10;
}
```

le compilateur OCaml optimise les appels terminaux par défaut

la compilation de

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

donne une boucle, comme en C

alors même qu'on n'a pas de traits impératifs dans le langage considéré ici (les variables `acc` et `n` sont immuables)

le programme obtenu est plus efficace

en particulier car on accède moins à la mémoire
(on n'utilise plus `call` et `ret` qui manipulent la pile)

sur l'exemple de `fact`, **l'espace de pile utilisé devient constant**

en particulier, on évite ainsi tout débordement de pile qui serait dû à un trop grand nombre d'appels imbriqués

```
Stack overflow during evaluation (looping recursion?).
```

```
Fatal error: exception Stack_overflow
```

```
Exception in thread "main" java.lang.StackOverflowError
```

```
Segmentation fault
```

etc.

un tri rapide en C

```
void quicksort(int a[], int l, int r) {
  if (r - l <= 1) return;
  // on partitionne a[l..r[ en trois
  //     l         lo        hi        r
  //     +---------+---------+---------+
  //   a|...<p...|...=p...|...>p...|
  //     +---------+---------+---------+
  ...
  quicksort(a, l, lo);
  quicksort(a, hi, r);
}
```

mais un tel code pourrait faire déborder la pile

sur la plus petite des deux moitiés d'abord

```
void quicksort(int a[], int l, int r) {
  ...
  if (lo - l < r - hi) {
    quicksort(a, l, lo);
    quicksort(a, hi, r);
  } else {
    quicksort(a, hi, r);
    quicksort(a, l, lo);
  }
}
```

le second appel est terminal

et une taille de pile logarithmique est maintenant garantie

application : quicksort

et si mon compilateur n'optimise pas les appels terminaux (ex. Java) ?

pas grave, on le fait soi-même !

```
void quicksort(int a[], int l, int r) {
  while (r - l > 1) {
    ...
    if (lo - l < r - hi) {
      quicksort(a, l, lo);
      l = hi;
    } else {
      quicksort(a, hi, r);
      r = lo;
    }
  }
}
```

il est important de noter que la notion d'appel terminal

- peut être optimisée dans tous les langages *a priori*
  (mais Java et Python ne le font pas, par exemple)

- n'est pas liée à la récursivité
  (même si c'est le plus souvent une fonction récursive qui fera déborder la pile)

il n'est pas toujours facile de remplacer les appels par des appels terminaux

exemple : étant donné un type d'arbres binaires immuables tel que

```
type 'a tree = Empty | Node of 'a tree * 'a * 'a tree
```

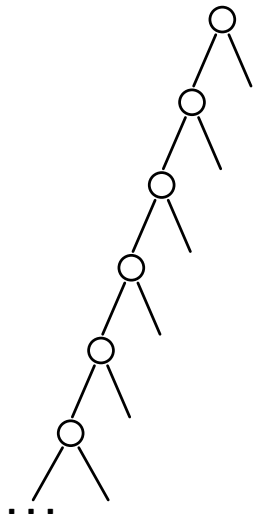
écrire une fonction qui calcule la hauteur d'un arbre

```
  val height: 'a tree -> int
```

le code naturel

```
let rec height = function
| Empty           -> 0
| Node (l, _, r) -> 1 + max (height l) (height r)
```

va provoquer un débordement de pile sur un arbre de grande hauteur



au lieu de calculer la hauteur h de l'arbre, calculons $k(h)$ pour une fonction k quelconque, appelée **continuation**

```
val height: 'a tree -> (int -> 'b) -> 'b
```

on appelle cela la **programmation par continuations**
(en anglais *continuation-passing style*, ou CPS)

le programme voulu s'en déduira avec la continuation identité,

```
height t (fun h -> h)
```

le code prend alors la forme suivante

```
let rec height t k = match t with
 | Empty ->
     k 0
 | Node (l, _, r) ->
     height l (fun hl ->
     height r (fun hr ->
     k (1 + max hl hr)))
```

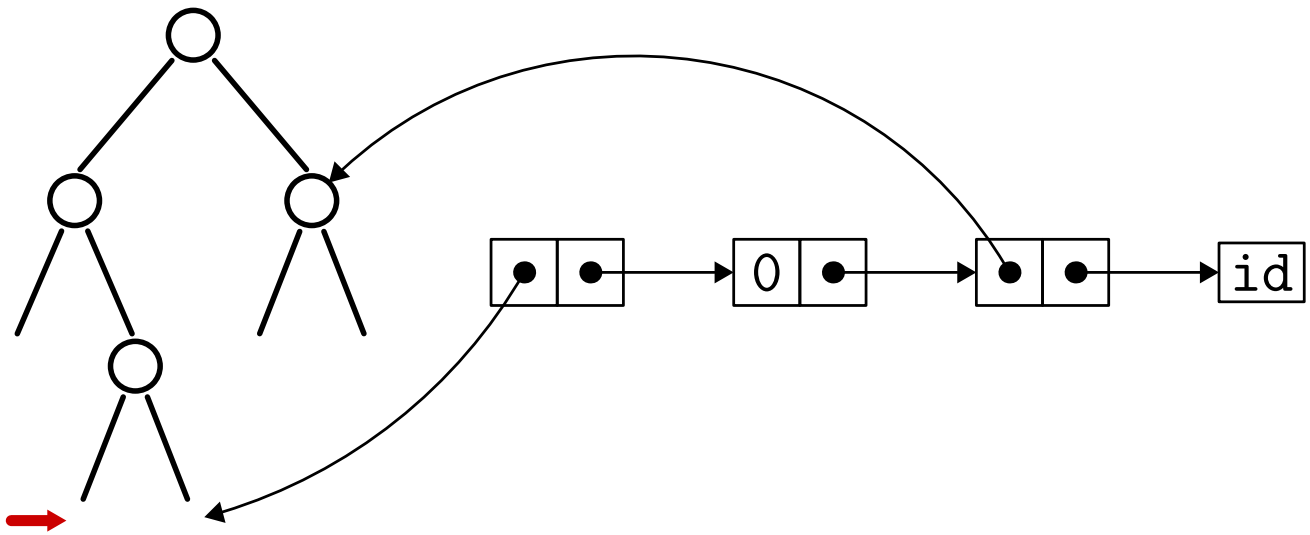
on constate que **tous** les appels à `height` et `k` sont **terminaux**

le calcul de `height` se fait donc en espace de pile constant

on a remplacé l'espace sur la pile par de l'espace **sur le tas**

il est occupé par les fermetures

la première fermeture capture `r` et `k`, la seconde `hl` et `k`



bien sûr, il y a d'autres solutions, ad hoc, pour calculer la hauteur d'un arbre sans faire déborder la pile (par exemple un parcours en largeur)

de même qu'il y a d'autres solutions si le type d'arbres est plus complexe (arbres mutables, pointeurs parents, etc.)

mais la solution à base de CPS a le mérite d'être **mécanique**

et si le langage optimise l'appel terminal
mais ne propose pas de fonctions anonymes (par exemple C) ?

il suffit de construire des fermetures soi-même, à la main
(une structure contenant un pointeur de fonction et l'environnement)

on peut même le faire avec un type ad hoc

```
enum kind { Kid, Kleft, Kright };

struct Kont {
  enum kind kind;
  union { struct Node *r; int hl; };
  struct Kont *kont;
};
```

et une fonction pour l'appliquer

```
int apply(struct Kont *k, int v) { ... }
```

cela s'appelle la **défonctionnalisation** (Reynolds 1972)

la suite

- TD 6
  - projet Mini Go
- cours 7
  - compilateur optimisant= 1/2