École Polytechnique

INF564 – Compilation

Jean-Christophe Filliâtre

production de code (2/3)

aujourd'hui

1. fonctions comme valeurs de première classe
2. optimisation des appels terminaux
3. poursuite de mini-C

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 somme n =
  let f x = x * x in
  let rec boucle i =
    if i = n then 0 else f i + boucle (i+1)
  in
  boucle 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 carré 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 $\{\texttt{eps}, \texttt{f}, \texttt{sum}\}$

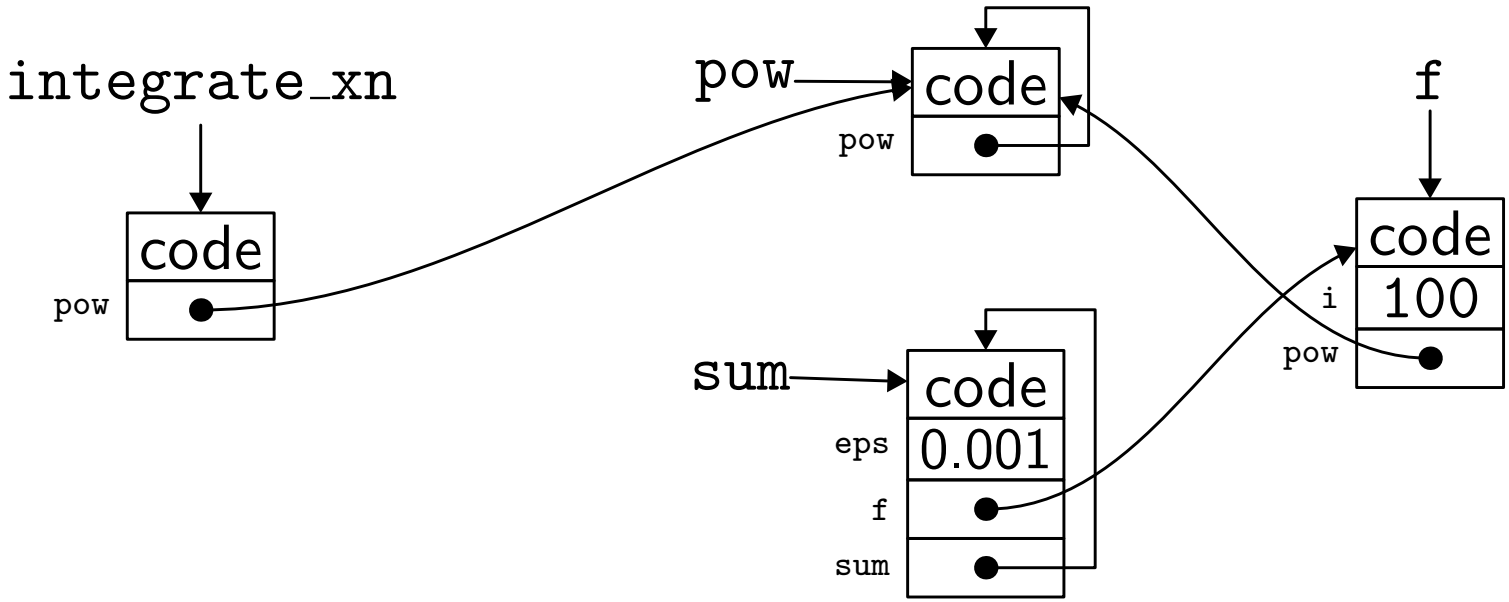
représentation de la fermeture

la fermeture peut être représentée de la manière suivante :

- un unique bloc sur le tas, dont
- le premier champ contient l'adresse du code
- les champs suivants contiennent les valeurs des variables libres, et uniquement celles-là

```
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

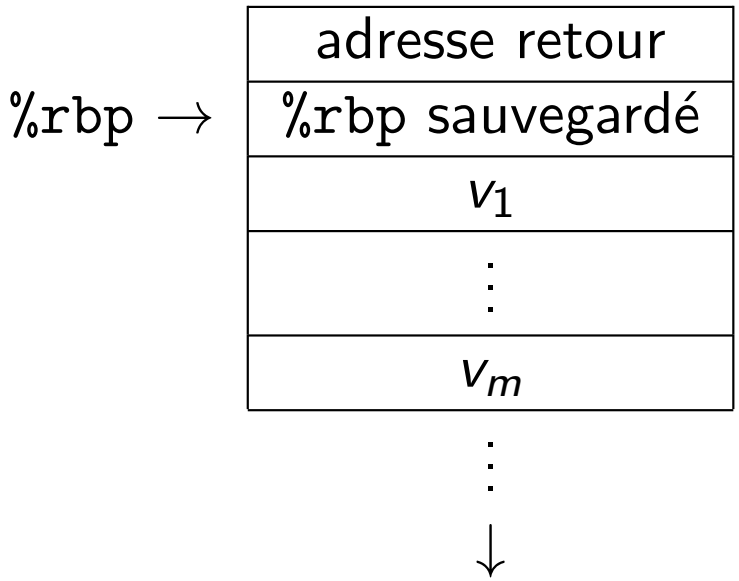
- une **variable globale** introduite par `let`
  (allouée dans le segment de données)
- une **variable locale** introduite par `let in`
  (allouée dans le tableau d'activation / un registre)
- une **variable contenue dans une fermeture**
- l'**argument** d'une fonction (le `x` de `fun x -> e`)

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 `malloc`)
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

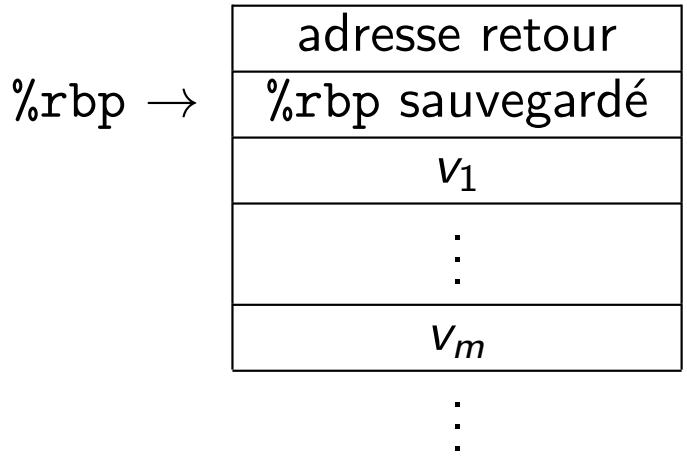
variable locale
: la valeur se trouve en n(`%rbp`) / dans un registre

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 31

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 qui ne fait pas déborder la pile

```
void quicksort(int a[], int l, int r) {
  if (l >= r - 1) return;
  int m = partition(a, l, r);
  if (m - l < r - m - 1) { // plus petit côté en premier
    quicksort(a, l,     m);
    quicksort(a, m + 1, r);
  } else {
    quicksort(a, m + 1, r);
    quicksort(a, l,     m);
  }
}
```

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
```

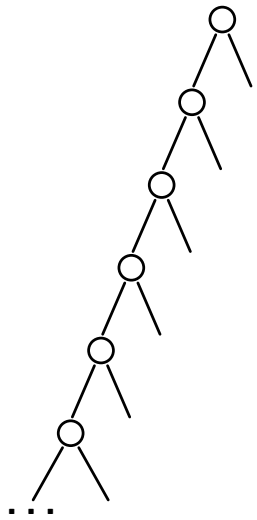
écrire une fonction qui calcule la hauteur d'un arbre

```
  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)))
```

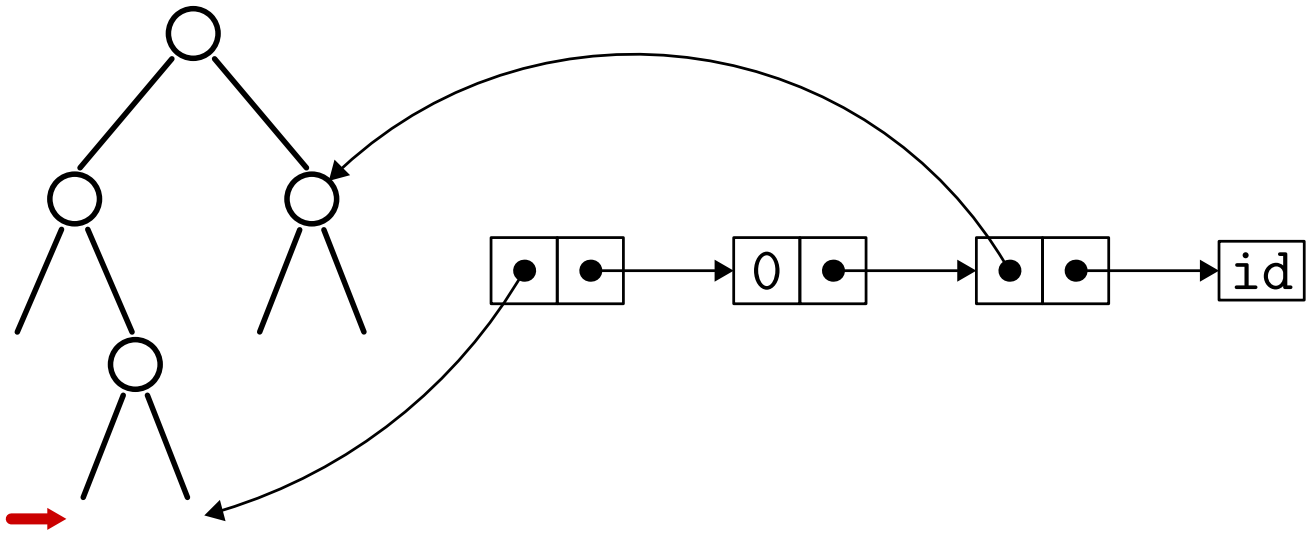
on constate que **tous** les appels à `height` et `k` sont **terminaux**

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)

poursuite de mini-C

la production de code optimisé a été décomposée en **plusieurs phases**

1. sélection d'instructions
2. RTL (*Register Transfer Language*)
3. ERTL (*Explicit Register Transfer Language*)
4. LTL (*Location Transfer Language*)
5. code linéarisé (assembleur)

```
int fact(int x) {
  if (x <= 1) return 1;
  return x * fact(x-1);
}
```

phase 1 : la sélection d'instructions

```
int fact(int x) {
  if (Mjlei 1 x) return 1;
  return Mmul x fact((Maddi -1) x);
}
```

phase 2 : RTL (*Register Transfer Language*)

```
#2 fact(#1)
  entry : L10
  exit  : L1
  locals:
  L10: mov #1 #6  --> L9
  L9 : jle $1 #6  --> L8, L7
  L8 : mov $1 #2  --> L1
```

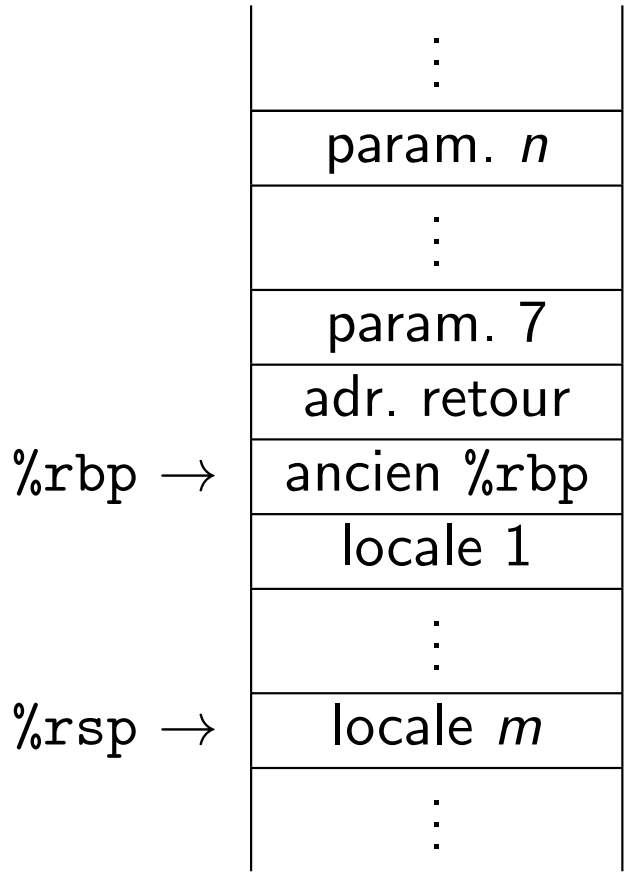
```
  L7: mov #1 #5             --> L6
  L6: add $-1 #5            --> L5
  L5: #3 <- call fact(#5) --> L4
  L4: mov #1 #4             --> L3
  L3: mov #3 #2             --> L2
  L2: imul #4 #2            --> L1
```

la troisième phase est la transformation de RTL vers le langage **ERTL** (*Explicit Register Transfer Language*)

objectif : expliciter les **conventions d'appel**, en l'occurrence ici

- les six premiers paramètres sont passés dans `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` et les suivants sur la pile
- le résultat est renvoyé dans `%rax`
- en particulier, `putchar` et `sbrk` sont des fonctions de bibliothèques, avec paramètre dans `%rdi` et résultat dans `%rax`
- la division `idivq` impose dividende et quotient dans `%rax`
- les registres *callee-saved* doivent être sauvegardés par l'appelé (`%rbx`, `%r12`, `%r13`, `%r14`, `%r15`, `%rbp`)

le tableau d'activation s'organise ainsi :



la zone des m variables locales contiendra tous les pseudo-registres qui ne pourront être stockés dans des registres physiques ; c'est l'allocation de registres (phase 4) qui déterminera m

instructions ERTL (1/3)

dans ERTL, on trouve les mêmes instructions que dans RTL :

`mov` $n\ r \to L$	
`load` $n(r_1)\ r_2 \to L$	
`store` $r_1\ n(r_2) \to L$	
`unop` $op\ r \to L$	opération unaire (`neg`, etc.)
`binop` $op\ r_1\ r_2 \to L$	opération binaire (`add`, `mov`, etc.)
`ubranch` $br\ r \to L_1, L_2$	branchement unaire (`jz`, etc.)
`bbranch` $br\ r_1\ r_2 \to L_1, L_2$	branchement binaire (`jle`, etc.)
`goto` $\to L$	

dans RTL, on avait une instruction

$$\texttt{call}\ r \leftarrow f(r_1, \ldots, r_n) \rightarrow L$$

dans ERTL, on a maintenant seulement

$$\texttt{call}\ f(k) \rightarrow L$$

i.e. il ne reste que le nom de la fonction à appeler, car de nouvelles instructions vont être insérées pour charger les paramètres dans des registres et sur la pile, et pour récupérer le résultat dans `%rax`

on conserve néanmoins le nombre k de paramètres passés dans des registres (qui sera utilisé par la phase 4)

instructions ERTL (3/3)

enfin, on trouve de **nouvelles** instructions :

`alloc_frame` $\rightarrow L$	allouer le tableau d'activation
`delete_frame` $\rightarrow L$	désallouer le tableau d'activation (note : on ne connaît pas encore sa taille)
`get_param` n r $\rightarrow L$	accéder à un paramètre sur la pile (avec n(`%rbp`))
`push_param` r $\rightarrow L$	empiler la valeur de r
`return`	instruction explicite de retour

insertion des nouvelles instructions

on ne change pas la structure du graphe de flot de contrôle ; on se contente d'insérer de **nouvelles instructions**

- au début de chaque fonction, pour
  - allouer le tableau d'activation
  - sauvegarder les registres *callee-saved*
  - copier les paramètres dans les pseudo-registres correspondants
- à la fin de chaque fonction, pour
  - copier le pseudo-registre contenant le résultat dans `%rax`
  - restaurer les registres *callee-saved*
  - désallouer le tableau d'activation
  - exécuter `return`
- à chaque appel, pour
  - copier les pseudo-registres contenant les paramètres dans `%rdi`, ... et sur la pile avant l'appel
  - copier `%rax` dans le pseudo-registre contenant le résultat après l'appel
  - dépiler les arguments, le cas échéant

on traduit chaque instruction de RTL vers une/plusieurs instructions ERTL

presque partout l'identité, si ce n'est les appels (`call`) et la division

dividende et quotient sont dans `%rax`

l'instruction RTL

$$L_1 : \texttt{ binop div } r_1\ r_2 \to L$$

devient trois instructions ERTL

$$\begin{aligned} L_1 &: \texttt{ binop mov } r_2\ \texttt{\%rax} \to L_2 \\ L_2 &: \texttt{ binop div } r_1\ \texttt{\%rax} \to L_3 \\ L_3 &: \texttt{ binop mov \%rax } r_2 \to L \end{aligned}$$

où L_2 et L_3 sont des étiquettes fraîches

(attention au sens : on divise ici r_2 par r_1)

on traduit l'instruction RTL

$$L_1 : \texttt{ call } r \leftarrow f(r_1, \ldots, r_n) \rightarrow L$$

par une séquence d'instructions ERTL

1. copier $\min(n, 6)$ arguments $r_1, r_2, \ldots$ dans `%rdi`,`%rsi`,...
2. si $n > 6$, passer les autres sur la pile avec `push_param`
3. exécuter `call` $f(\min(n, 6))$
4. copier `%rax` dans r
5. si $n > 6$, dépiler $8 \times (n - 6)$ octets

qui commence à la même étiquette L_1 et transfert le contrôle au final à la même étiquette L

le code RTL

```
L5: #3 <- call fact(#5)  --> L4
```

est traduit en ERTL par

```
L5 : mov #5 %rdi    --> L12
L12: call fact(1)   --> L11
L11: mov %rax #3    --> L4
```

il reste à traduire chaque fonction

RTL

```
r f(r,..,r)
  locals : { r, ... }
  entry  : L
  exit   : L
  cfg    : ...
```

ERTL

```
f(n)
  locals : { r, ... }
  entry  : L

  cfg    : ...
```

registres *callee-saved*

à chaque registre *callee-saved*, on associe un nouveau pseudo-registre pour sa sauvegarde, qu'on ajoute aux variables locales de la fonction

note : on ne se pose pas pour l'instant la question de savoir quels sont les registres *callee-saved* qui seront effectivement utilisés

à l'entrée de la fonction, il faut

- allouer le tableau d'activation avec `alloc_frame`
- sauvegarder les registres *callee-saved*
- copier les paramètres dans leurs pseudo-registres

exemple : factorielle

RTL

```
#2 fact(#1)
  entry : L10
  exit  : L1
  locals:




  L10: mov #1 #6     --> L9
  ...
```

ERTL

```
fact(1)
  entry : L17

  locals: #7, #8
  L17: alloc_frame   --> L16
  L16: mov %rbx #7   --> L15
  L15: mov %r12 #8   --> L14
  L14: mov %rdi #1   --> L10
  L10: mov #1 #6     --> L9
  ...
```

(on suppose ici que les seuls registres *callee-saved* sont `%rbx` et `%r12`)

à la sortie de la fonction, il faut

- copier le pseudo-registre contenant le résultat dans `%rax`
- restaurer les registres sauvegardés
- désallouer le tableau d'activation

exemple : factorielle

RTL

```
#2 fact(#1)
  entry : L10
  exit  : L1
  locals:
  ...
  L8 : mov $1 #2     --> L1
  ...
  L2 : imul #4 #2    --> L1
```

ERTL

```
fact(1)
  entry : L17

  locals: #7, #8
  ...
  L8 : mov $1 #2     --> L1
  ...
  L2 : imul #4 #2    --> L1
  L1 : mov #2 %rax   --> L21
  L21: mov #7 %rbx   --> L20
  L20: mov #8 %r12   --> L19
  L19: delete_frame --> L18
  L18: return
```

au total, on obtient le code ERTL suivant :

```
fact(1)
  entry : L17
  locals: #7,#8
  L17: alloc_frame  --> L16
  L16: mov %rbx #7  --> L15
  L15: mov %r12 #8  --> L14
  L14: mov %rdi #1  --> L10
  L10: mov #1 #6    --> L9
  L9 : jle $1 #6 --> L8, L7
  L8 : mov $1 #2    --> L1
  L1 : goto         --> L22
  L22: mov #2 %rax  --> L21
  L21: mov #7 %rbx  --> L20
  L20: mov #8 %r12  --> L19
  L19: delete_frame --> L18
  L18: return
  L7 : mov #1 #5    --> L6
  L6 : add $-1 #5   --> L5
  L5 : goto         --> L13
  L13: mov #5 %rdi  --> L12
  L12: call fact(1) --> L11
  L11: mov %rax #3  --> L4
  L4 : mov #1 #4    --> L3
  L3 : mov #3 #2    --> L2
  L2 : imul #4 #2   --> L1
```

c'est encore loin de ce que l'on imagine être un bon code x86-64 pour la factorielle

à ce point, il faut comprendre que

- l'allocation de registres (phase 4) tâchera d'associer des registres physiques aux pseudo-registres de manière à limiter l'usage de la pile mais aussi de supprimer certaines instructions

  si par exemple on réalise #8 par `%r12`, on supprime tout simplement les deux instructions `L15` et `L20`

- le code n'est pas encore organisé linéairement (le graphe est seulement affiché de manière arbitraire) ; ce sera le travail de la phase 5, qui tâchera notamment de minimiser les sauts

c'est au niveau de la traduction RTL $\rightarrow$ ERTL qu'il faut réaliser l'optimisation des **appels terminaux** si on le souhaite

en effet, les instructions à produire ne sont pas les mêmes, et ce changement aura une influence dans la phase suivante d'allocation des registres

il y a une difficulté cependant, si la fonction appelée par un appel terminal n'a pas le même nombre d'arguments passés sur la pile ou de variables locales, car le tableau d'activation doit être modifié

deux solutions au moins

- limiter l'optimisation de l'appel terminal aux cas où le tableau d'activation n'est pas modifié : c'est le cas s'il s'agit d'un appel terminal d'une fonction récursive à elle-même
- l'appelant modifie le tableau d'activation et transfère le contrôle à l'appelé *après* l'instruction de création de son tableau d'activation

la quatrième phase est la traduction de ERTL vers **LTL** (*Location Transfer Language*)

il s'agit de faire disparaître les pseudo-registres au profit

- de registres physiques, de préférence
- d'emplacements de pile, sinon

c'est ce que l'on appelle l'**allocation de registres**

l'allocation de registres est une phase complexe, que l'on va elle-même décomposer en plusieurs étapes

1. analyse de **durée de vie**
   - il s'agit de déterminer à quels moments précis la valeur d'un pseudo-registre est nécessaire pour la suite du calcul

2. construction d'un **graphe d'interférence**
   - il s'agit d'un graphe indiquant quels sont les pseudo-registres qui ne peuvent pas être réalisés par le même emplacement

3. allocation de registres par **coloration de graphe**
   - c'est l'affectation proprement dite de registres physiques et d'emplacements de pile aux pseudo-registres

dans la suite on appelle *variable* un pseudo-registre ou un registre physique

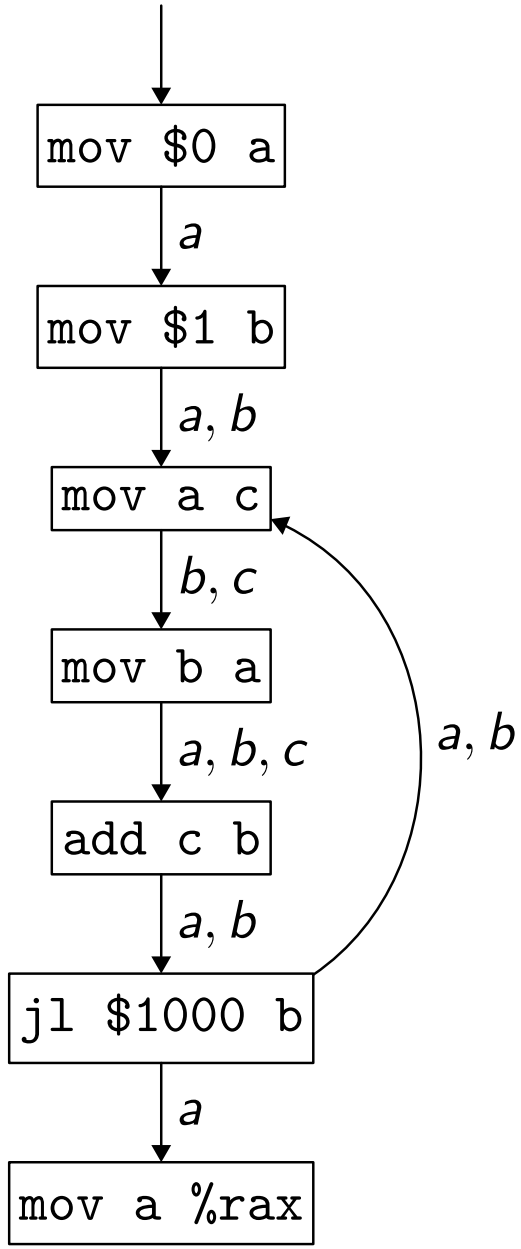
Définition (variable vivante)

En un point de programme, une variable est dite **vivante** *si la valeur qu'elle contient peut être utilisée dans la suite de l'exécution.*

on dit ici « peut être utilisée » car la propriété « est utilisée » n'est pas décidable ; on se contente donc d'une approximation correcte

attachons les variables vivantes aux *arêtes* du graphe de flot de contrôle

```
    mov $0 a
    mov $1 b
L1: mov a c
    mov b a
    add c b
    jl  $1000 b L1
    mov a %rax
```



la notion de variable vivante se déduit des **définitions** et des **utilisations** des variables effectuées par chaque instruction

Définition

Pour une instruction l du graphe de flot de contrôle, on note

- *$def(l)$ l'ensemble des variables définies par cette instruction, et*
- *$use(l)$ l'ensemble des variables utilisées par cette instruction.*

exemple : pour l'instruction add r_1 r_2 on a

$$def(l) = \{r_2\} \quad \text{et} \quad use(l) = \{r_1, r_2\}$$

pour calculer les variables vivantes, il est commode de les associer non pas aux arêtes mais plutôt aux *nœuds* du graphe de flot de contrôle, c'est-à-dire à chaque instruction

mais il faut alors distinguer les variables **vivantes à l'entrée** d'une instruction et les variables **vivantes à la sortie**

Définition

Pour une instruction l du graphe de flot de contrôle, on note

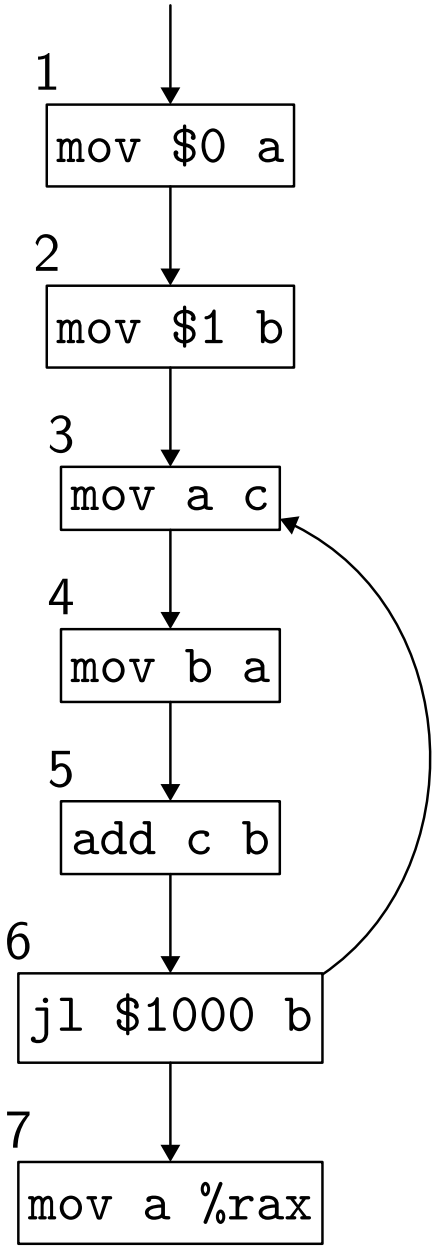
- *$in(l)$ l'ensemble des variables vivantes sur l'ensemble des arêtes arrivant sur l, et*
- *$out(l)$ l'ensemble des variables vivantes sur l'ensemble des arêtes sortant de l.*

les équations qui définissent $in(l)$ et $out(l)$ sont les suivantes

$$\begin{cases} in(l) &= use(l) \cup (out(l) \setminus def(l)) \\ out(l) &= \bigcup_{s \in succ(l)} in(s) \end{cases}$$

il s'agit d'équations récursives dont la plus petite solution est celle qui nous intéresse

nous sommes dans le cas d'une fonction monotone sur un domaine fini et nous pouvons donc appliquer le théorème de Tarski (cf cours 3)



$$\begin{cases} in(l) &= use(l) \cup (out(l) \setminus def(l)) \\ out(l) &= \bigcup_{s \in succ(l)} in(s) \end{cases}$$

	use	def	in	out	in	out		in	out
1		a					...		a
2		b				a	...	a	a,b
3	a	c	a		a	b	...	a,b	b,c
4	b	a	b		b	b,c	...	b,c	a,b,c
5	b,c	b	b,c		b,c	b	...	a,b,c	a,b
6	b		b		b	a	...	a,b	a,b
7	a		a		a		...	a	

on obtient le point fixe après 7 itérations

en supposant un graphe de flot de contrôle contenant N sommets et N variables, un calcul brutal a une complexité $O(N^4)$ dans le pire des cas

on peut améliorer l'efficacité du calcul de plusieurs façons

- en calculant dans l'≪ ordre inverse ≫ du graphe de flot de contrôle, et en calculant *out* avant *in* (sur l'exemple précédent, on converge alors en 3 itérations au lieu de 7)
- en fusionnant les sommets qui n'ont qu'un unique prédécesseur et qu'un unique successeur avec ces derniers (*basic blocks*)
- en utilisant un algorithme plus subtil qui ne recalcule que les valeurs de *in* et *out* qui peuvent avoir changé ; c'est l'algorithme de Kildall

idée : si $in(l)$ change, alors il faut refaire le calcul pour les prédécesseurs de l uniquement

$$\begin{cases} out(l) & = & \bigcup_{s \in succ(l)} in(s) \\ in(l) & = & use(l) \cup (out(l) \setminus def(l)) \end{cases}$$

d'où l'algorithme suivant

```
soit WS un ensemble contenant tous les sommets
tant que WS n'est pas vide
    extraire un sommet l de WS
    old_in <- in(l)
    out(l) <- ...
    in(l)  <- ...
    si in(l) est différent de old_in(l) alors
        ajouter tous les prédécesseurs de l dans WS
```

le calcul des ensemble $def(l)$ (définitions) et $use(l)$ (utilisations) est immédiat pour la plupart des instructions

exemples

	def	*use*
`mov` n r	$\{r\}$	$\emptyset$
`mov` r_1 r_2	$\{r_2\}$	$\{r_1\}$
`unop` op r	$\{r\}$	$\{r\}$
`goto`	$\emptyset$	$\emptyset$
...		

il est un peu plus subtil en ce qui concerne les appels

pour un appel, on exprime notamment que tous les registres *caller-saved* peuvent être écrasés par l'appel

	def	*use*
`call` $f(k)$	*caller-saved*	les k premiers de `%rdi`,`%rsi`,...,`%r9`

enfin, pour `return`, `%rax` et tous les registres *callee-saved* peuvent être utilisés

	def	*use*
`return`	$\emptyset$	$\{$`%rax`$\} \cup$ *callee-saved*

reconsidérons la forme ERTL de la factorielle

```
fact(1)
  entry : L17
  locals: #7,#8
  L17: alloc_frame  --> L16
  L16: mov %rbx #7  --> L15
  L15: mov %r12 #8  --> L14
  L14: mov %rdi #1  --> L10
  L10: mov #1 #6    --> L9
  L9 : jle $1 #6 --> L8, L7
  L8 : mov $1 #2    --> L1
  L1 : goto         --> L22
  L22: mov #2 %rax  --> L21
  L21: mov #7 %rbx  --> L20
  L20: mov #8 %r12  --> L19
  L19: delete_frame --> L18
  L18: return
  L7 : mov #1 #5    --> L6
  L6 : add $-1 #5   --> L5
  L5 : goto         --> L13
  L13: mov #5 %rdi  --> L12
  L12: call fact(1) --> L11
  L11: mov %rax #3  --> L4
  L4 : mov #1 #4    --> L3
  L3 : mov #3 #2    --> L2
  L2 : imul #4 #2   --> L1
```

sur l'exemple de la factorielle

```
L17: alloc_frame --> L16   in = %r12,%rbx,%rdi   out = %r12,%rbx,%rdi
L16: mov %rbx #7 --> L15   in = %r12,%rbx,%rdi   out = #7,%r12,%rdi
L15: mov %r12 #8 --> L14   in = #7,%r12,%rdi     out = #7,#8,%rdi
L14: mov %rdi #1 --> L10   in = #7,#8,%rdi       out = #1,#7,#8
L10: mov #1 #6   --> L9    in = #1,#7,#8         out = #1,#6,#7,#8
L9 : jle $1 #6 -> L8, L7   in = #1,#6,#7,#8      out = #1,#7,#8
L8 : mov $1 #2   --> L1    in = #7,#8            out = #2,#7,#8
L1 : goto        --> L22   in = #2,#7,#8         out = #2,#7,#8
L22: mov #2 %rax --> L21   in = #2,#7,#8         out = #7,#8,%rax
L21: mov #7 %rbx --> L20   in = #7,#8,%rax       out = #8,%rax,%rbx
L20: mov #8 %r12 --> L19   in = #8,%rax,%rbx     out = %r12,%rax,%rbx
L19: delete_frame--> L18   in = %r12,%rax,%rbx   out = %r12,%rax,%rbx
L18: return                in = %r12,%rax,%rbx   out =
L7 : mov #1 #5   --> L6    in = #1,#7,#8         out = #1,#5,#7,#8
L6 : add $-1 #5  --> L5    in = #1,#5,#7,#8      out = #1,#5,#7,#8
L5 : goto        --> L13   in = #1,#5,#7,#8      out = #1,#5,#7,#8
L13: mov #5 %rdi --> L12   in = #1,#5,#7,#8      out = #1,#7,#8,%rdi
L12: call fact(1)--> L11   in = #1,#7,#8,%rdi    out = #1,#7,#8,%rax
L11: mov %rax #3 --> L4    in = #1,#7,#8,%rax    out = #1,#3,#7,#8
L4 : mov #1 #4   --> L3    in = #1,#3,#7,#8      out = #3,#4,#7,#8
L3 : mov #3 #2   --> L2    in = #3,#4,#7,#8      out = #2,#4,#7,#8
L2 : imul #4 #2  --> L1    in = #2,#4,#7,#8      out = #2,#7,#8
```

TD 7

---

production du code ERTL
analyse de durée de vie

du code est fourni (pour OCaml et Java)

- syntaxe abstraite de ERTL
- pour une instruction ERTL, calcul de *def*, *use* et des successeurs
- interprète de code ERTL pour tester
- affichage de code ERTL pour débugger
- `main.ml` / `Main.java` pour la ligne de commande

notre module/classe `Register` décrit aussi les registres physiques (avec le même type) et fournit en particulier

`Register.parameters`	les n premiers paramètres (liste)
`Register.result`	pour le résultat d'une fonction
`Register.rax`	pour la division
`Register.callee_saved`	liste des registres *callee-saved*
`Register.rsp`	pour la manipulation de la pile