École Normale Supérieure

Langages de programmation et compilation

Jean-Christophe Filliâtre

compilateur optimisant (1/2)

l'objectif de ce cours et du prochain : produire du **code efficace**

jusqu'ici, nous avons très mal utilisé les capacités de l'assembleur x86-64 :

- très peu de registres utilisés, alors qu'il y en a 16
  - arguments et variables locales systématiquement sur la pile
  - calculs intermédiaires sur la pile également

- instructions mal utilisées
  - exemple : on n'a jamais utilisé

```
        add  $3, %rdi
```

(on utilisait un registre temporaire + la pile !)

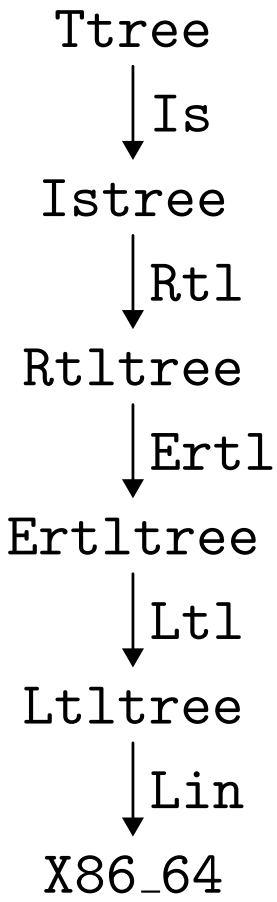
il est illusoire de chercher à produire du code efficace en une seule passe

la production de code va être 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)

phases du compilateur (code OCaml)

le point de départ est l'arbre de syntaxe abstraite issu du typage



cette architecture de compilateur est valable pour tous les grands paradigmes de programmation (impérative, fonctionnelle, objet, etc.)

pour l'illustrer, on fait néanmoins le choix d'un langage particulier, en l'occurrence un petit fragment du langage C

on considère un fragment très simple du langage **C** avec

- des entiers (type `int`)
- des structures allouées sur le tas avec `malloc` (uniquement des pointeurs sur ces structures et pas d'arithmétique de pointeurs)
- des fonctions
- une « primitive » pour afficher un entier : `printf("%d\n",`e`)`

$$
\begin{array}{rcl}
E & \to & n \\
 & | & L \\
 & | & L = E \\
 & | & E\ op\ E \mid \texttt{-}\ E \mid \texttt{!}\ E \\
 & | & x(E, \ldots, E) \\
 & | & \texttt{malloc(sizeof(struct } x\texttt{))} \\
L & \to & x \\
 & | & E\texttt{->}x \\
op & \to & \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
 & | & \texttt{\&\&} \mid \texttt{||} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \\
D & \to & V \\
 & | & T\ x(T\ x, \ldots, T\ x)\ B \\
 & | & \texttt{struct } x\ \{V \ldots V\}\texttt{;}
\end{array}
$$

$$
\begin{array}{rcl}
S & \to & E\texttt{;} \\
 & | & \texttt{if (}E\texttt{)}\ S \\
 & | & \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S \\
 & | & \texttt{while (}E\texttt{)}\ S \\
 & | & \texttt{return}\ E\texttt{;} \\
 & | & \texttt{printf("\%d\textbackslash n",}E\texttt{);} \\
 & | & B \\
B & \to & \{\ V \ldots V\ S \ldots S\ \} \\
V & \to & \texttt{int}\ x, \ldots, x\texttt{;} \\
 & | & \texttt{struct}\ x\ \texttt{*}x, \ldots, \texttt{*}x\texttt{;} \\
T & \to & \texttt{int} \mid \texttt{struct}\ x\ \texttt{*} \\
P & \to & D \ldots D
\end{array}
$$

exemples

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

```
struct list { int val; struct list *next; };

int print(struct list *l) {
  while (l) {
    printf("%d\n", l->val);
    l = l->next;
  }
  return 0;
}
```

on suppose l'analyse sémantique effectuée ; l'arbre résultant est le suivant :

```
type file = { gvars: decl_var list; funs: decl_fun list; }
and decl_var = typ * ident
and decl_fun = {
  fun_typ:     typ;           fun_name: ident;
  fun_formals: decl_var list; fun_body: block; }
and block = decl_var list * stmt list
and stmt =
  | Sskip
  | Sexpr   of expr
  | Sif     of expr * stmt * stmt
  | Swhile  of expr * stmt
  | Sblock  of block
  | Sreturn of expr
  | Sprintf of expr
```

dans les expressions, on a déjà distingué variables locales et globales, et les noms sont uniques

```
and expr = { expr_node: expr_node; expr_typ: typ }
and expr_node =
  | Econst          of int32
  | Eaccess_local   of ident
  | Eassign_local   of ident * expr
  | Eaccess_global  of ident
  | Eassign_global  of ident * expr
  | Eaccess_field   of expr * int          (* indice du champ *)
  | Eassign_field   of expr * int * expr
  | Eunop           of unop * expr                    (* - ! *)
  | Ebinop          of binop * expr * expr  (* + - == etc. *)
  | Ecall           of ident * expr list
  | Emalloc         of structure
```

phase 1 : sélection d'instructions

la première phase est la **sélection d'instructions**

objectif :

- remplacer les opérations arithmétiques du C par celles de x86-64
- remplacer les accès aux champs de structures par des opérations `mov`

on peut naïvement traduire chaque opération arithmétique de C par l'instruction correspondante de x86-64

cependant, x86-64 fournit des instructions permettant une plus grande efficacité, notamment

- addition d'un registre et d'une constante
- décalage des bits vers la gauche ou la droite, correspondant à une multiplication ou à une division par une puissance de deux
- comparaison d'un registre avec une constante

d'autre part, il est souhaitable d'évaluer autant d'expressions que possible pendant la compilation (évaluation partielle)

exemples : on peut simplifier

- $(1 + e_1) + (2 + e_2)$ en $e_1 + e_2 + 3$
- $e + 1 < 10$ en $e < 9$
- $!(e_1 < e_2)$ en $e_1 \geq e_2$
- $0 \times e$ en 0, mais seulement si e est **pure** *i.e.* sans effet de bord

on se donne de nouveaux arbres pour le résultat de la sélection d'instructions

`Ttree.mli` (avant)

```
type expr =
  ...
type stmt =
  ...
type file =
  ...
```

`Istree.mli` (après)

```
type expr =
  ...
type stmt =
  ...
type file =
  ...
```

l'objectif est d'écrire des fonctions

```
val expr    : Ttree.expr -> Istree.expr
val stmt    : Ttree.stmt -> Istree.stmt
val program: Ttree.file -> Istree.file
```

nouvelle syntaxe abstraite

les opérations sont maintenant celles de x86-64

```
type munop = Maddi of int32 | Msetei of int32 | ...
type mbinop = Mmov | Madd | Msub ... | Msete | Msetne ...
```

et elles remplacent les opérations du C

```
type expr =
  | Emunop  of munop * expr              (* remplace Eunop *)
  | Embinop of mbinop * expr * expr    (* remplace Ebinop *)
  | ...
```

on conserve cependant && et || pour l'instant

```
  | Eand of expr * expr
  | Eor  of expr * expr
```

pour réaliser l'évaluation partielle, on va utiliser des *smart constructors*

plutôt que d'écrire `Embinop (Madd, e1, e2)` directement, on introduit une fonction

```
val mk_add: expr -> expr -> expr
```

qui effectue d'éventuelles simplifications et se comporte comme le constructeur sinon

le constructeur de l'addition

voici quelques simplifications possibles pour l'addition

```
let rec mk_add e1 e2 = match e1, e2 with
  | Econst n1, Econst n2 ->
      Econst (Int32.add n1 n2)
  | e, Econst 0l | Econst 0l, e ->
      e
  | Emunop (Maddi n1, e), Econst n2
  | Econst n2, Emunop (Maddi n1, e) ->
      mk_add (Econst (Int32.add n1 n2)) e
  | e, Econst n | Econst n, e ->
      Emunop (Maddi n, e)
  | _ ->
      Embinop (Madd, e1, e2)
```

deux aspects sont essentiels concernant ces simplifications

- la sémantique des programmes doit être préservée
  - exemple : si un ordre d'évaluation gauche/droite est spécifié, on ne peut pas simplifier $(0 - e_1) + e_2$ en $e_2 - e_1$ si e_1 ou e_2 n'est pas pure

- la fonction de simplification doit terminer
  - il faut trouver une grandeur positive sur l'expression simplifiée qui diminue strictement à chaque appel récursif du *smart constructor*

la traduction se fait alors mot à mot

```
let rec expr e = match e.Ttree.expr_node with
  | Ttree.Ebinop (Badd, e1, e2) ->
      mk_add (expr e1) (expr e2)
  | Ttree.Ebinop (Bsub, e1, e2) ->
      mk_sub (expr e1) (expr e2)
  | Ttree.Eunop (Unot, e) ->
      mk_not (expr e)
  | Ttree.Eunop (Uminus, e) ->
      mk_sub (Econst 0l) (expr e)
  | ...
```

et c'est un morphisme pour les autres constructions (`Eaccess_local`, `Eaccess_global`, `Ecall`, etc.)

la sélection d'instruction introduit également des opérations explicites d'accès à la mémoire

une adresse mémoire est donnée par une expression et un décalage (pour tirer parti de l'adressage indirect)

```
type expr =
  | ...
  | Eload  of expr * int
  | Estore of expr * int * expr
```

dans notre cas, ce sont les accès aux champs de structures qui sont transformés en accès à la mémoire

on adopte ici un schéma simple où chaque champ occupe exactement un mot (on représente donc le type `int` sur 64 bits)

d'où

```
let rec expr e = match e.Ttree.expr_node with
  | ...
  | Ttree.Eaccess_field (e, n) ->
      Eload (expr e, n * word_size)
  | Ttree.Eassign_field (e1, n, e2) ->
      Estore (expr e1, n * word_size, expr e2)
```

avec ici

```
let word_size = 8    (* architecture 64 bits *)
```

pour le reste

pour le reste, rien à signaler (la sélection d'instructions est un morphisme en ce qui concerne les instructions du C)

on en profite cependant pour oublier les types et regrouper l'ensemble des variables locales au niveau de la fonction

```
type deffun = {
  fun_name    : ident;
  fun_formals: ident list;
  fun_locals  : ident list;
  fun_body    : stmt list;
}

type file = {
  gvars: ident list;
  funs : deffun list;
}
```

la deuxième phase est la transformation vers le langage **RTL** (*Register Transfer Language*)

objectif :

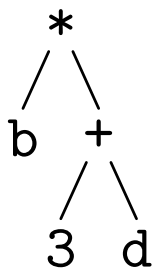
- détruire la structure arborescente des expressions et des instructions au profit d'un **graphe de flot de contrôle**, qui facilitera les phases ultérieures ; on supprime en particulier la distinction entre expressions et instructions

- introduire des **pseudo-registres** pour représenter les calculs intermédiaires ; ces pseudo-registres sont en nombre illimité et deviendront plus tard soit des registres x86-64, soit des emplacements de pile

considérons l'expression C

```
b * (3 + d)
```

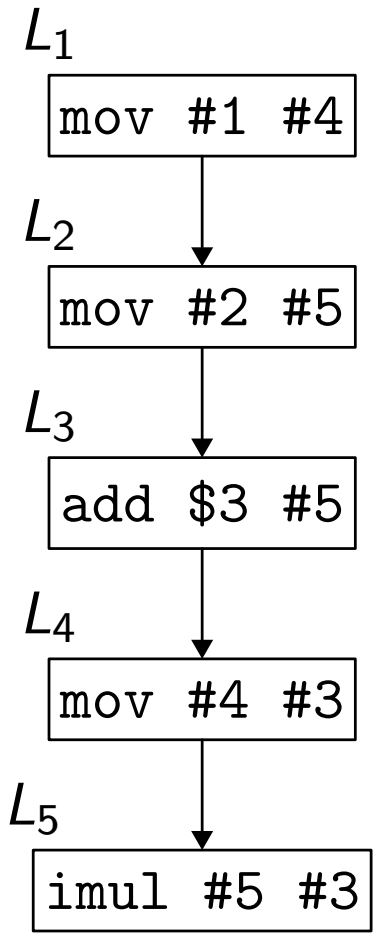
c'est-à-dire l'arbre



supposons que b et d sont dans les pseudo-registres #1 et #2

et le résultat dans #3

alors on construit le graphe



on se donne un module `Register` pour les pseudo-registres

```
type t

val fresh: unit -> t

module S: Set.S with type elt = t
type set = S.t
```

représentation du graphe de flot de contrôle

on se donne également un module `Label` pour des étiquettes représentant les sommets du graphe de flot de contrôle

```
type t

val fresh: unit -> t

module M: Map.S with type key = t
type 'a map = 'a M.t
```

un graphe est alors simplement un dictionnaire associant une instruction RTL à chaque étiquette

```
type graph = instr Label.map
```

inversement, chaque instruction RTL indique quelle est l'étiquette suivante (ou les étiquettes suivantes) ; ainsi

```
type instr =
  | Econst of int32 * register * label
  | ...
```

l'instruction `Econst` (*n*, *r*, *l*) signifie « charger la valeur *n* dans le pseudo-registre *r* et transférer le contrôle à l'étiquette *l* »

de même, on trouve l'accès aux variables globales (toujours représentées par des identificateurs), les accès à la mémoire, `malloc` et `printf`

```
type instr =
  ...
  | Eaccess_global of ident * register * label
  | Eassign_global of register * ident * label

  | Eload  of register * int * register * label
  | Estore of register * register * int * label

  | Emalloc of register * int32 * label
  | Eprintf of register * label
```

enfin, les opérations arithmétiques manipulent maintenant des pseudo-registres

```
type instr =
  ...
  | Emunop  of munop  * register * label
  | Embinop of mbinop * register * register * label
```

pour construire le graphe de flot de contrôle, on le stocke (temporairement) dans une référence

```
let graph = ref Label.M.empty
```

et on se donne une fonction pour ajouter une instruction dans le graphe

```
let generate i =
  let l = Label.fresh () in
  graph := Label.M.add l i !graph;
  l
```

on traduit les expressions grâce à une fonction

```
val expr: register -> expr -> label -> label
```

qui prend en arguments

- le registre de destination de la valeur de l'expression
- l'expression à traduire
- l'étiquette de sortie *i.e.* correspondant à la suite du calcul

et renvoie l'étiquette d'entrée du calcul de cette expression

on construit donc le graphe en partant de la fin de chaque fonction

la traduction est relativement aisée

```
let rec expr destr e destl = match e with
  | Istree.Econst n ->
      generate (Econst (n, destr, destl))
```

lorsque nécessaire, on introduit des pseudo-registres frais

```
  | Istree.Embinop (op, e1, e2) ->
      let tmp2 = Register.fresh () in
      expr destr e1 (
      expr tmp2 e2 (generate (
      Embinop (op, tmp2, destr, destl))))
```

pour les variables locales, on se donne une table indiquant quel pseudo-registre est associé à chaque variable

```
  | Istree.Eaccess_local x ->
      let rx = Hashtbl.find locals x in
      generate (Embinop (Mmov, rx, destr, destl))
```

où `Mmov` est l'opération mov de x86-64

etc.

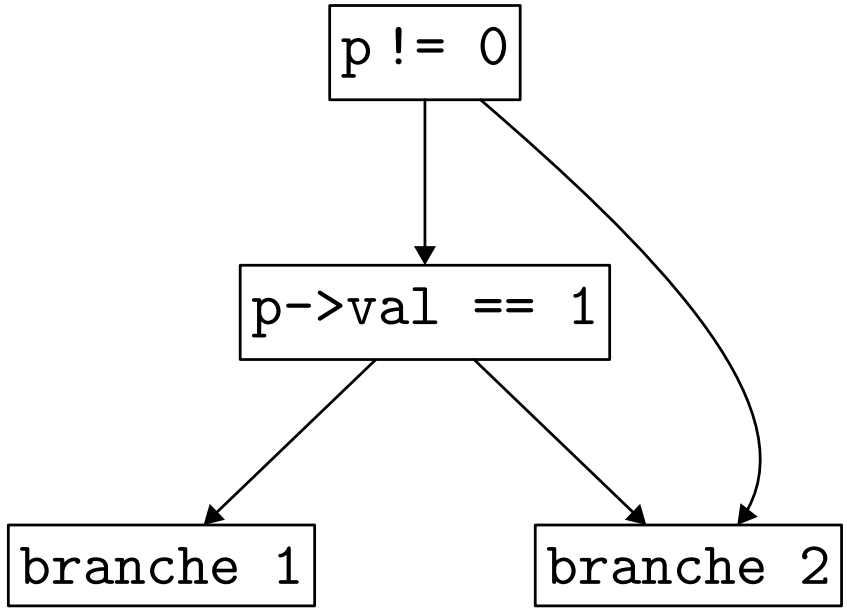
pour traduire les opérateurs `&&` et `||` et les instructions `if` et `while`
il nous faut des instructions RTL de **branchement**

```
type instr =
  ...
  | Emubranch of mubranch * register * label * label
  | Embbranch of mbbranch * register * register
                                     * label * label
  | Egoto     of label
```

avec

```
type mubranch = Mjz | Mjnz | Mjlei of int32 | ...

type mbbranch = Mjl | Mjle | ...
```

```
if (p != 0 && p->val == 1)
  ...branche 1...
else
  ...branche 2...
```



(les quatres blocs sont schématiques ; ils se décomposent en réalité en sous-graphes)

pour traduire une condition, on définit une fonction

```
val condition: expr -> label -> label -> label
```

les deux étiquettes passées en arguments correpondent à la suite du calcul dans les cas où la condition est respectivement vraie et fausse

on renvoie l'étiquette d'entrée de l'évaluation de la condition

traduction d'une condition

```
let rec condition e truel falsel = match e with
  | Istree.Eand (e1, e2) ->
      condition e1 (condition e2 truel falsel) falsel
  | Istree.Eor (e1, e2) ->
      condition e1 truel (condition e2 truel falsel)
  | Istree.Embinop (Mjle, e1, e2) ->
      let tmp1 = Register.fresh () in
      let tmp2 = Register.fresh () in
      expr tmp1 e1 (
      expr tmp2 e2 (generate (
      Embbranch (Mjle, tmp2, tmp1, truel, falsel))))
  | e ->
      let tmp = Register.fresh () in
      expr tmp e (generate (
      Emubranch (Mjz, tmp, falsel, truel)))
```

(on peut bien entendu traiter plus de cas particuliers)

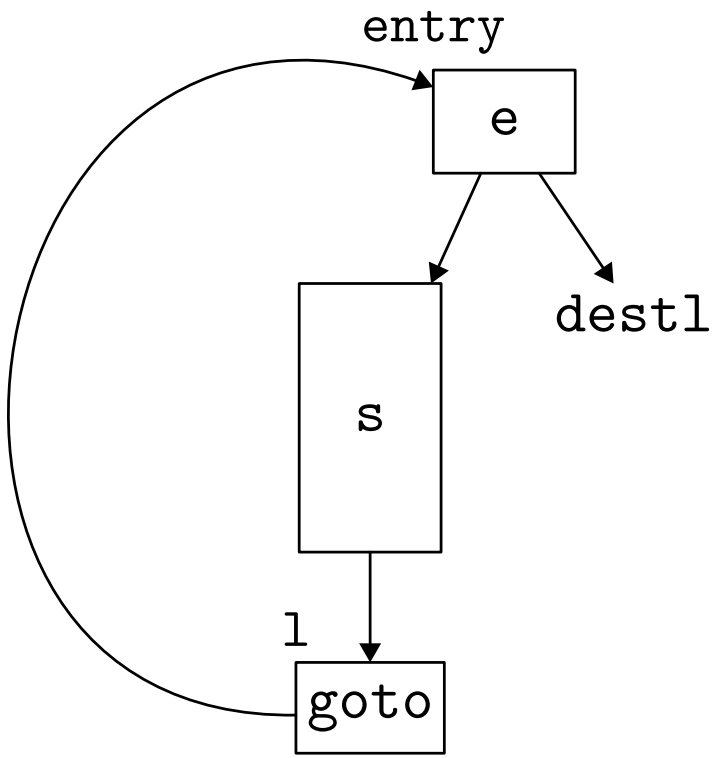
pour traduire `return`, on se donne un pseudo-registre `retr` pour recevoir le résultat de la fonction et une étiquette `exitl` correspondant à la sortie de la fonction ; sinon, on se donne une étiquette `destl` correspondant à la suite du calcul

```
let rec stmt retr s exitl destl = match s with
  | Istree.Sskip ->
      destl

  | Istree.Sreturn e ->
      expr retr e exitl

  | Istree.Sif (e, s1, s2) ->
      condition e
        (stmt retr s1 exitl destl)
        (stmt retr s2 exitl destl)
```

etc.

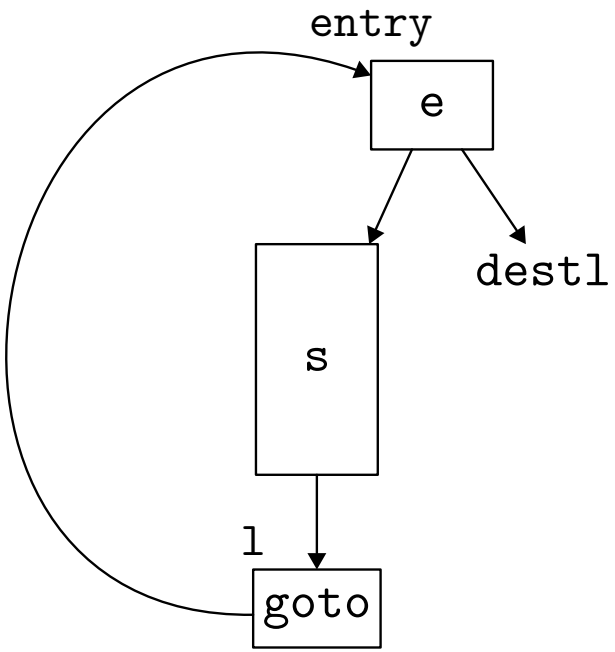
pour une boucle `while`, on crée un cycle dans le graphe de flot de contrôle

```
while (e) {
  ...s...
}
```



boucle

```
let rec stmt retr s exitl destl = match s with
  | ...
  | Istree.Swhile (e, s) ->
      let l = Label.fresh () in
      let entry = condition e (stmt retr s exitl l) destl in
      graph := Label.M.add l (Egoto entry) !graph;
      entry
```

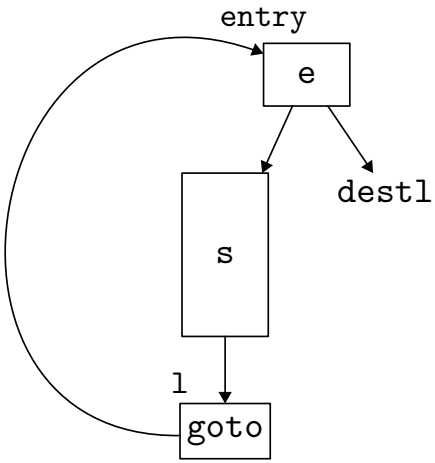


on peut aussi l'écrire comme un opérateur de point fixe

```
val loop: (label -> label) -> label
```

par exemple ainsi

```
let loop f =
  let l = Label.fresh () in
  let entry = f l in
  graph := Label.M.add l (Egoto entry) !graph;
  entry
```



et l'utiliser ainsi

```
| Istree.Swhile (e, s) ->
    loop (fun l -> condition e (stmt retr s exitl l) destl)
```

les paramètres d'une fonction et son résultat sont maintenant dans des pseudo-registres

```
type deffun = {
  fun_name    : ident;
  fun_formals: register list;
  fun_result  : register;
  fun_locals  : Register.set;
  fun_entry   : label;
  fun_exit    : label;
  fun_body    : instr Label.map;
}
```

de même pour l'appel

```
type instr =
  ...
  | Ecall of register * ident * register list * label
```

traduction d'une fonction

la traduction d'une fonction se compose des étapes suivantes

1. on alloue des pseudo-registres frais pour ses arguments, son résultat et ses variables locales
2. on part d'un graphe vide
3. on crée une étiquette fraîche pour la *sortie* de la fonction
4. on traduit le corps de la fonction dans RTL et le résultat est l'étiquette d'*entrée* de la fonction

considérons l'inévitable factorielle

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

on obtient

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

on suppose que le module `Register` décrit aussi les registres physiques

```
type t
...
val parameters: t list  (* pour les n premiers arguments *)
val rax: t              (* pour résultat, division *)
val callee_saved: t list
val rdi: t              (* pour malloc et printf *)
```

dans RTL, on avait

```
  | Ecall of register * ident * register list * label
```

dans ERTL, on a maintenant

```
  | Ecall of ident * int * label
```

i.e. il ne reste que le nom de la fonction à appeler, car de nouvelles instructions vont être insérées pour charger les arguments dans des registres et sur la pile, et pour récupérer le résultat dans `%rax`
(on conserve néanmoins le nombre de paramètres passés dans des registres, qui sera utilisé par la phase 4)

les instructions `Emalloc` et `Eprintf` disparaissent

les autres instructions de RTL sont inchangées

en revanche, de nouvelles instructions apparaissent :

- pour allouer et désallouer le tableau d'activation

```
| Ealloc_frame  of label
| Edelete_frame of label
```

(note : on ne connaît pas encore sa taille)

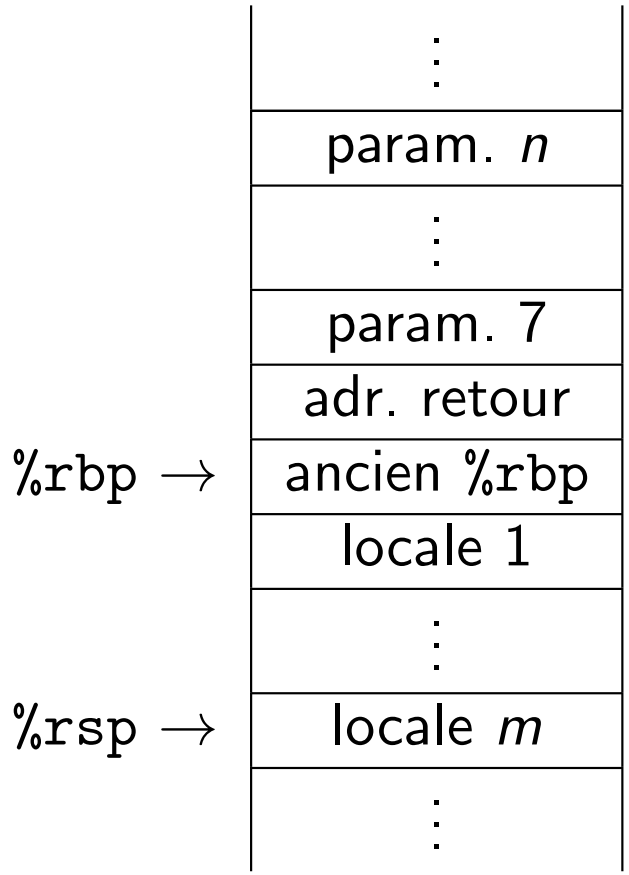
- pour lire / écrire les paramètres sur la pile

```
| Eget_param  of int * register * label
| Epush_param of register * label
```

- le retour est maintenant explicite

```
| Ereturn
```

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

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 arguments 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
- à chaque appel, pour
  - copier les pseudo-registres contenant les arguments dans `%rdi`, ... et sur la pile avant l'appel
  - copier `%rax` dans le pseudo-registre contenant le résultat après l'appel

on traduit les instructions de RTL vers ERTL avec une fonction

```
val instr: Rtltree.instr -> Ertltree.instr
```

peu de changement, si ce n'est les appels, c'est-à-dire les instructions `Ecall`, `Emalloc` et `Eprintf`, et la division

malloc et printf

```
| Rtltree.Emalloc (r, n, l) ->
    Econst  (n, Register.rdi,          generate (
    Ecall   ("malloc", 1,              generate (
    Embinop (Mmov, Register.rax, r,    l)))))
```

```
| Rtltree.Eprintf (r, l) ->
    Embinop (Mmov, r, Register.rdi,    generate (
    Ecall   ("print_int", 1,           l)))
```

dividende et quotient sont dans `%rax`

```
| Rtltree.Embinop (Mdiv, r1, r2, l) ->
    Embinop (Mmov, r2, Register.rax,    generate (
    Embinop (Mdiv, r1, Register.rax,    generate (
    Embinop (Mmov, Register.rax, r2,    l)))))
```

(attention au sens : on divise ici `r2` par `r1`)

en RTL, l'appel se présente sous la forme

```
  | Rtltree.Ecall (r, x, rl, l) ->
```

où `r` est le pseudo-registre qui reçoit le résultat, `x` le nom de la fonction et `rl` la liste des pseudo-registres contenant les arguments

on commence par associer les premiers paramètres aux registres physiques de `Register.parameters` :

```
let assoc_formals formals =
  let rec assoc = function
    | [],      _       -> [], []
    | rl,      []      -> [], rl
    | r :: rl, p :: pl -> let a, rl = assoc (rl, pl) in
                          (r, p) :: a, rl
  in
  assoc (formals, Register.parameters)
```

traduction de l'appel

on se donne

```
let move src dst l = generate (Embinop (Mmov, src, dst, l))
let push_param r l = generate (Epush_param (r, l))
let pop n l =
  if n = 0 then l else generate (Emunop (Maddi n, rsp, l))
```

l'appel se traduit alors ainsi (il faut lire le code « à l'envers »)

```
| Rtltree.Ecall (r, x, rl, l) ->
    let frl, fsl = assoc_formals rl in
    let n = List.length frl in
    let l = pop (word_size * List.length fsl) l in
    let l = generate (Ecall (x, n, move rax r l)) in
    let l = List.fold_left (fun l t -> push_param t l) l fsl in
    let l = List.fold_right (fun (t, r) l -> move t r l) frl l in
    Egoto l
```

le code RTL

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

est traduit en ERTL par

```
L5 : goto           --> L13
L13: mov #5 %rdi    --> L12
L12: call fact(1)   --> L11
L11: mov %rax #3    --> L4
```

il reste à traduire chaque fonction

RTL

```
type deffun = {
  fun_name    : ident;
  fun_formals: register list;
  fun_result  : register;
  fun_locals  : Register.set;
  fun_entry   : label;
  fun_exit    : label;
  fun_body    : instr Label.map;
}
```

ERTL

```
type deffun = {
  fun_name    : ident;
  fun_formals: int; (* nb *)

  fun_locals  : Register.set;
  fun_entry   : label;

  fun_body    : instr Label.map;
}
```

on associe un pseudo-registre à chaque registre physique qui doit être sauvegardé *i.e.* les registres de la liste `callee_saved`

```
let deffun f =
  graph := ...on traduit chaque instruction...
  let savers =
    List.map (fun r -> Register.fresh(), r) callee_saved in
  let entry = fun_entry savers f.Rtltree.fun_formals
                                 f.Rtltree.fun_entry in
  fun_exit savers f.Rtltree.fun_result f.Rtltree.fun_exit;
  { fun_name = f.Rtltree.fun_name;
    ...
    fun_body = !graph; }
```

à l'entrée de la fonction, il faut

- allouer le tableau d'activation avec `Ealloc_frame`
- sauvegarder les registres (liste `savers`)
- copier les arguments dans leurs pseudo-registres (`formals`)

```
let fun_entry savers formals entry =
  let frl, fsl = assoc_formals formals in
  let ofs = ref word_size in
  let l = List.fold_left
    (fun l t -> ofs := !ofs + word_size; get_param t !ofs l)
    entry fsl in
  let l = List.fold_right (fun (t, r) l -> move r t l) frl l in
  let l = List.fold_right (fun (t, r) l -> move r t l) savers l in
  generate (Ealloc_frame l)
```

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

```
let fun_exit savers retr exitl =
  let l = generate (Edelete_frame (generate Ereturn)) in
  let l = List.fold_right (fun (t, r) l -> move t r l) savers l in
  let l = move retr Register.rax l in
  graph := Label.M.add exitl (Egoto l) !graph
```

exemple : 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
```

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

déception

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

  ainsi, si 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 (cf cours 9)

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 notamment 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

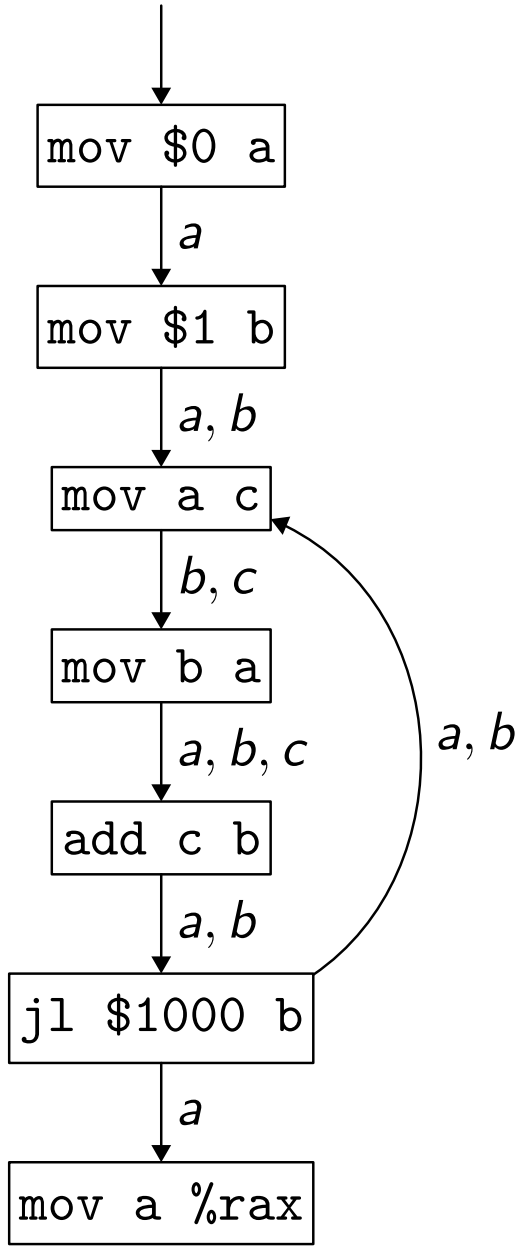
Définition (variable vivante)

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

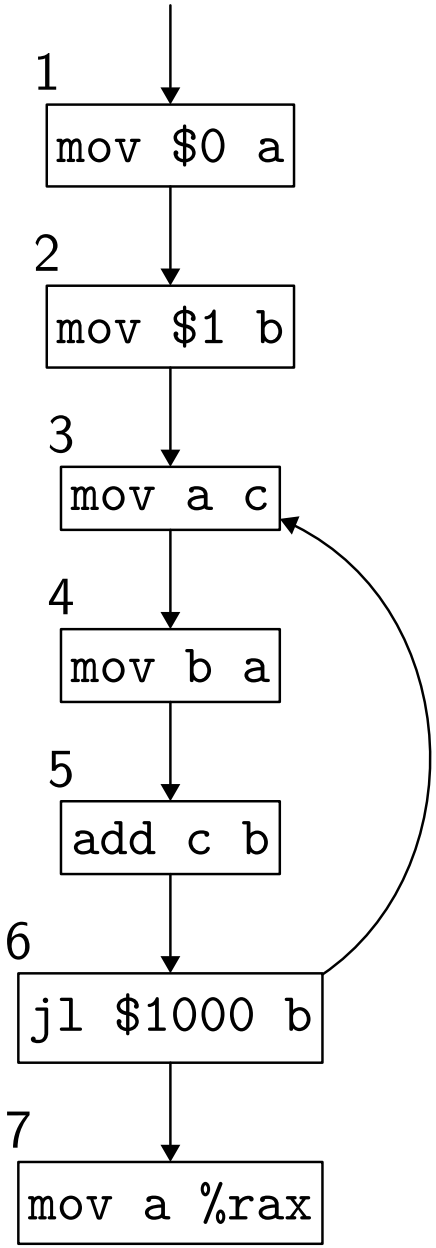
- *$in(l)$ l'ensemble des variables vivantes sur l'ensemble des arêtes arrivant sur l, et*
- *$out(l)$ l'ensemble des variables vivantes sur l'ensemble des arêtes sortant de l.*

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 6)



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

	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
```

calcul de *def* et *use*

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

exemples

```
let def_use = function
  | Econst (_,r,_)          -> [r], []
  | Eassign_global (r,_,_) -> [],  [r]
  | Emunop (_,r,_)          -> [r], [r]
  | Egoto _                 -> [],  []
  | ...
```

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

pour un appel, on exprime que les $min(6, n)$ premiers registres de la liste `parameters` vont être utilisés, et que tous les registres *caller-saved* peuvent être écrasés par l'appel

```
  | Ecall (_,n,_) ->
      caller_saved, prefix n parameters
```

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

```
  | Ereturn ->
      [], rax :: 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
```

la suite

- TD 11
  - coloriage de graphe
- prochain cours
  - compilateur optimisant, partie 2