École Polytechnique

CSC_52064 – Compilation

Jean-Christophe Filliâtre

compilateur optimisant (1/2)

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

on va chercher à utiliser au mieux l'assembleur x86-64, en exploitant notamment

- ses 16 registres
  - pour passer arguments et résultat
  - pour les calculs intermédiaires

- ses instructions
  - par exemple la capacité d'ajouter une constante à un registre

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

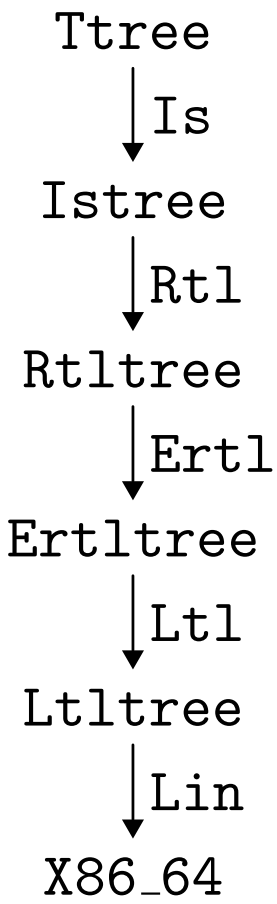
il est illusoire de chercher à produire du code efficace en une seule passe

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)

(l'architecture est celle du compilateur CompCert de Xavier Leroy
cf `http://compcert.inria.fr/`)

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.)

on l'illustre ici sur un petit fragment du langage C (celui du projet)

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
- la fonction de bibliothèque `putchar`

on fait ici le choix d'entiers 64 bits signés pour représenter (dans le code compilé) les valeurs du type `int`
ainsi, elles auront la même taille qu'un pointeur

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

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

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) {
    putchar(l->val);
    l = l->next;
  }
  return 0;
}
```

on suppose le typage effectué

en particulier, on suppose connu le type de chaque expression

note : pour mini-C, les types ne sont pas utiles pour la génération de code ; cependant,

- le typage nous apporte une forme de sûreté, comme ne pas mélanger entiers et pointeurs
- pour un fragment plus large de C, les types seraient nécessaires, par exemple pour l'arithmétique de pointeurs, le caractère signé ou non des opérations, etc.

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 explicites d'accès à la mémoire

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
- comparaison d'un registre avec une constante
- décalage des bits vers la gauche ou la droite, correspondant à une multiplication ou à une division par une puissance de deux
- utilisation de `lea` (voir TD 1)

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

attention : la sémantique des programmes doit être préservée

si un ordre d'évaluation gauche/droite était spécifié, on ne pourrait simplifier $(0 - e_1) + e_2$ en $e_2 - e_1$ que si e_1 et e_2 n'interfèrent pas

c'est le cas par exemple si e_1 et e_2 sont pures i.e. sans effet

en C, l'ordre d'évaluation n'est pas spécifié et on peut donc le faire

en arithmétique **non signée** en C, on ne pourrait pas remplacer $e + 1 < 10$ par $e < 9$ car $e + 1$ peut valoir 0 par débordement arithmétique
(si e est le plus grand entier, $e + 1 < 10$ est vrai mais $e < 9$ est faux)

en arithmétique signée, en revanche, le débordement est un comportement non défini en C (*undefined behavior*)

on peut donc faire cette simplification pour le type `int`

on ne peut remplacer $0 \times e$ par 0 que si l'expression e est sans effet

comme nos expressions incluent des appels de fonction,
déterminer si e est sans effet n'est pas décidable

mais on peut se contenter d'une approximation

$$
\begin{aligned}
\mathit{pure}(n) &= \mathit{true} \\
\mathit{pure}(x) &= \mathit{true} \\
\mathit{pure}(e_1 \texttt{ + } e_2) &= \mathit{pure}(e_1) \wedge \mathit{pure}(e_2) \\
&\vdots \\
\mathit{pure}(e_1 \texttt{ = } e_2) &= \mathit{false} \\
\mathit{pure}(f(e_1, \ldots, e_n)) &= \mathit{false} \quad \text{(on ne sait pas)}
\end{aligned}
$$

pour réaliser l'évaluation partielle, on peut utiliser des constructeurs intelligents (*smart constructors*)

un *smart constructor* se comporte comme un constructeur de la syntaxe abstraite mais il effectue d'éventuelles simplifications

exemple : pour l'addition, on se donne quelque chose comme

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

voici quelques simplifications possibles pour l'addition

$$
\begin{aligned}
\texttt{mkAdd}(n_1, n_2) &= n_1 + n_2 \\
\texttt{mkAdd}(0, e) &= e \\
\texttt{mkAdd}(e, 0) &= e \\
\texttt{mkAdd}(\texttt{addi } n_1\ e, n_2) &= \texttt{mkAdd}(n_1 + n_2, e) \\
\texttt{mkAdd}(n, e) &= \texttt{addi } n\ e \\
\texttt{mkAdd}(e, n) &= \texttt{addi } n\ e \\
\texttt{mkAdd}(e_1, e_2) &= \texttt{add } e_1\ e_2 \quad \text{sinon}
\end{aligned}
$$

bien sûr, 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

$$
\begin{aligned}
IS(e_1 + e_2) &= \texttt{mkAdd}(IS(e_1), IS(e_2)) \\
IS(e_1 - e_2) &= \texttt{mkSub}(IS(e_1), IS(e_2)) \\
IS(!\, e_1) &= \texttt{mkNot}(IS(e_1)) \\
IS(-e_1) &= \texttt{mkSub}(0, IS(e_1)) \\
&\vdots
\end{aligned}
$$

et c'est un morphisme pour les autres constructions

la sélection d'instruction introduit également des opérations explicites d'accès à la mémoire, notées `load` et `store`

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

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

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

d'où

$$\begin{aligned} IS(e_1\texttt{->}x) &= \texttt{load}\ IS(e_1)\ (n \times \texttt{wordsize}) \\ IS(e_1\texttt{->}x\ \texttt{=}\ e_2) &= \texttt{store}\ IS(e_1)\ (n \times \texttt{wordsize})\ IS(e_2) \end{aligned}$$

où n est l'indice du champ x dans la structure et $\texttt{wordsize} = 8$ (64 bits)

avec une structure telle que

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

la sélection d'instruction de l'expression

```
p->a = p->b + 2
```

donne quelque chose comme

```
store p 0 (addi 2 (load p 8))
```

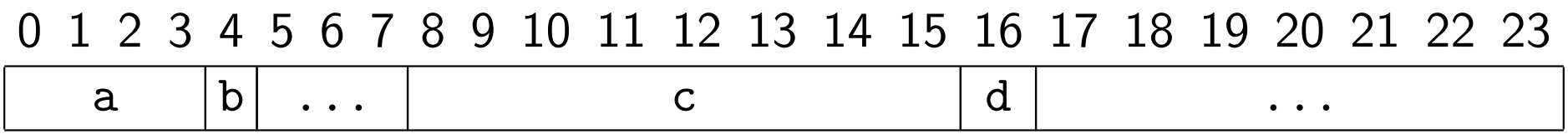
avec plus de types, on aurait des champs de longueur variable

le compilateur insère alors des octets de remplissage (*padding*) pour aligner les champs selon leur type

exemple : avec la structure

```
struct S { int a; char b; int *c; char d; };
```

on a un total de 24 octets, comme ceci



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

exemple

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

int print(struct list *l) {
  struct list *p;
  p = l;
  while (p) {
    int c;
    c = p->val;
    putchar(c);
    p = p->next;
  }
  return 0;
}
```

```
// plus besoin du type list



print(l) {
  locals p, c;
  p = l;
  while (p) {

    c = load p 0;
    putchar(c);
    p = load p 8;
  }
  return 0;
}
```

l'inévitable factorielle

```
int fact(int x) {

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

```
fact(x) {
  locals:
  if (setle x $1) return 1;
  return imul x fact(addi $-1 x);
}
```

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

objectif :

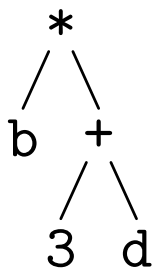
- détruire la structure arborescente des expressions et des instructions au profit d'un **graphe de flot de contrôle** (CFG en anglais), qui facilitera les phases ultérieures ; on supprime en particulier la distinction entre expressions et instructions

- 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)
```

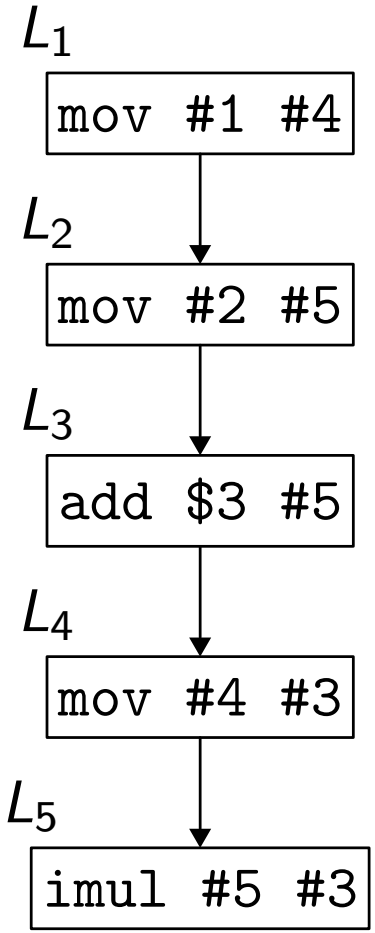
c'est-à-dire l'arbre



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

et le résultat dans #3

alors on construit le graphe



le graphe de flot de contrôle peut être représenté par un dictionnaire associant une instruction RTL à chaque étiquette

inversement, chaque instruction RTL indique quelle est l'étiquette suivante (ou les étiquettes suivantes pour une instruction de branchement)

ainsi, l'instruction RTL

$$\texttt{mov}\ n\ r\ \rightarrow L$$

signifie « charger la constante n dans le pseudo-registre r et transférer le contrôle à l'étiquette L »

`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.)
`call` $r \leftarrow f(r_1, \ldots, r_n) \to L$	
`goto` $\to L$	

on construit un graphe de flot de contrôle pour chaque fonction du programme

on construit le graphe de flot de contrôle « en partant de la fin », c'est-à-dire en connaissant à chaque instant l'étiquette désignant la suite du calcul

on traduit une expression en se donnant

- un registre r_d de destination de la valeur de l'expression
- une étiquette L_d correspondant à la suite du calcul

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

$$RTL(e,\, r_d,\, L_d)$$

la traduction est relativement aisée

$$
\begin{array}{rcll}
RTL(n, r_d, L_d) & = & \text{ajouter } L_1 : \texttt{mov } n\ r_d \ \rightarrow L_d & \text{avec } L_1 \text{ frais} \\
 & & \text{renvoyer } L_1 & \\
RTL(e_1 + e_2, r_d, L_d) & = & \text{ajouter } L_3 : \texttt{add } r_2\ r_d \ \rightarrow L_d & \text{avec } r_2, L_3 \text{ frais} \\
 & & L_2 \leftarrow RTL(e_2, r_2, L_3) & \\
 & & L_1 \leftarrow RTL(e_1, r_d, L_2) & \\
 & & \text{renvoyer } L_1 &
\end{array}
$$

etc.

attention : il faut lire le code de bas en haut

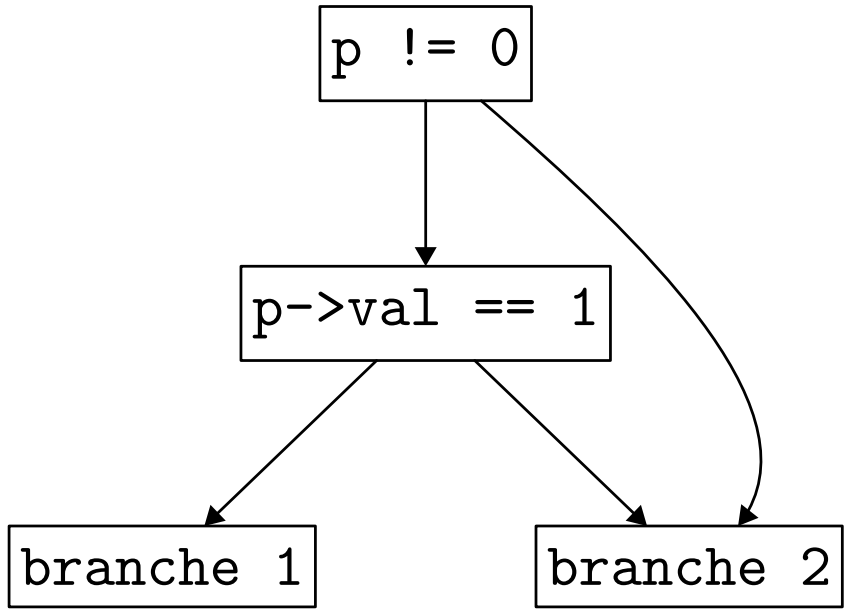
pour les variables locales, on se donne une table où chaque variable est associée à un pseudo-registre

la lecture ou l'écriture d'une variable locale est alors traduite avec l'instruction `mov` (opérateur binaire)

pour traduire les opérateurs `&&` et `||` et les instructions `if` et `while`
on utilise les instructions RTL de **branchement**

exemple :

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



(les quatres blocs sont schématiques ;
ce sont en réalité des sous-graphes)

traduction d'une condition

pour traduire une condition, on se donne deux étiquettes

- une étiquette L_t correspondant à la suite du calcul dans les cas où la condition est vraie
- une autre étiquette L_f dans le cas où elle est fausse

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

traduction d'une condition

$$RTL_c(e_1 \,\&\&\, e_2, L_t, L_f) = RTL_c(e_1,\ RTL_c(e_2, L_t, L_f),\ L_f)$$

$$RTL_c(e_1 \,||\, e_2, L_t, L_f) = RTL_c(e_1,\ L_t,\ RTL_c(e_2, L_t, L_f))$$

$$RTL_c(e_1 <= e_2, L_t, L_f) =$$
ajouter L_3 : `bbranch jle` r_2 r_1 $\rightarrow L_t, L_f$
$L_2 \leftarrow RTL(e_2, r_2, L_3)$
$L_1 \leftarrow RTL(e_1, r_1, L_2)$
renvoyer L_1

$$RTL_c(e, L_t, L_f) =$$
ajouter L_2 : `ubranch jz` r $\rightarrow L_f, L_t$
$L_1 \leftarrow RTL(e, r, L_2)$
renvoyer L_1

(on peut bien entendu traiter plus de cas particuliers)

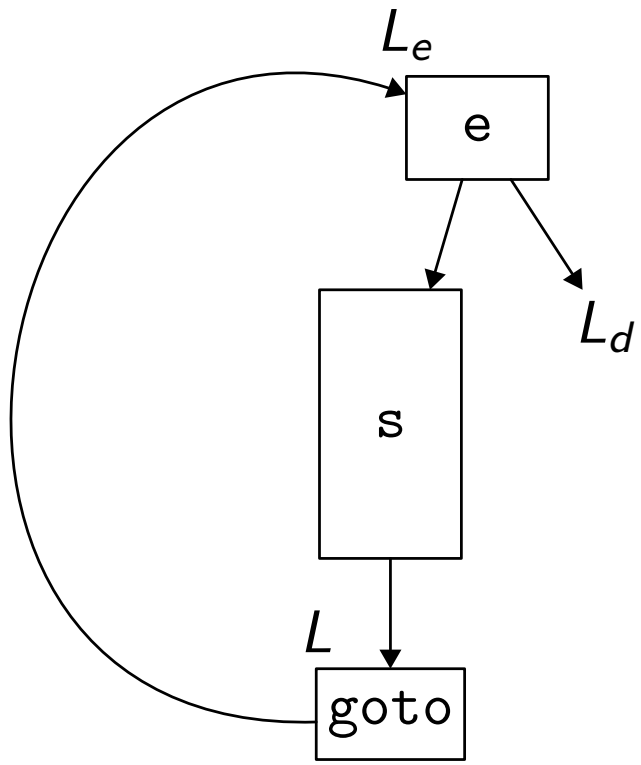
pour traduire `return`, on se donne un pseudo-registre r_{ret} pour recevoir le résultat de la fonction et une étiquette L_{ret} correspondant à la sortie de la fonction

$$
\begin{aligned}
RTL(\texttt{;}, L_d) &= \text{renvoyer } L_d \\
RTL(\texttt{return } e\texttt{;}, L_d) &= RTL(e, r_{ret}, L_{ret}) \\
RTL(\texttt{if(}e\texttt{)}s_1 \texttt{ else } s_2, L_d) &= RTL_c(e,\ RTL(s_1, L_d),\ RTL(s_2, L_d))
\end{aligned}
$$

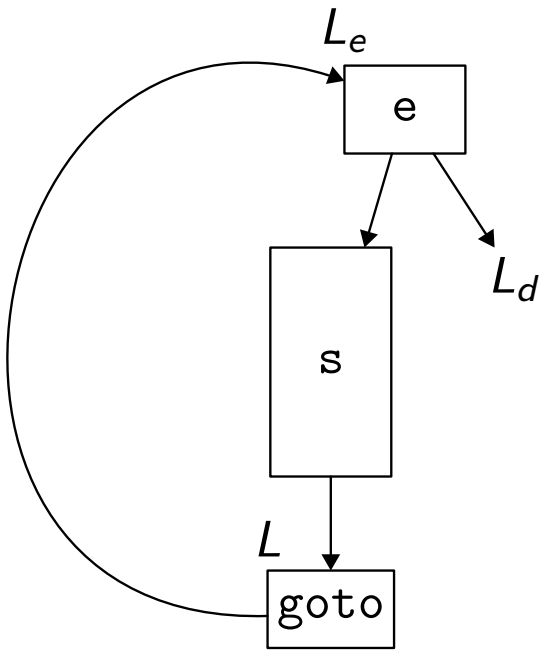
etc.

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

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



$$RTL(\texttt{while}(e)\,s, L_d) \quad = \quad L_e \leftarrow RTL_c(e, RTL(s, L),\ L_d)$$
ajouter L : goto L_e
renvoyer L_e



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

```
#3 f(#1, #2) { ... }
```

de même que les paramètres effectifs dans un appel

```
call #4 <- f(#5, #6)
```

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

toujours avec la factorielle

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

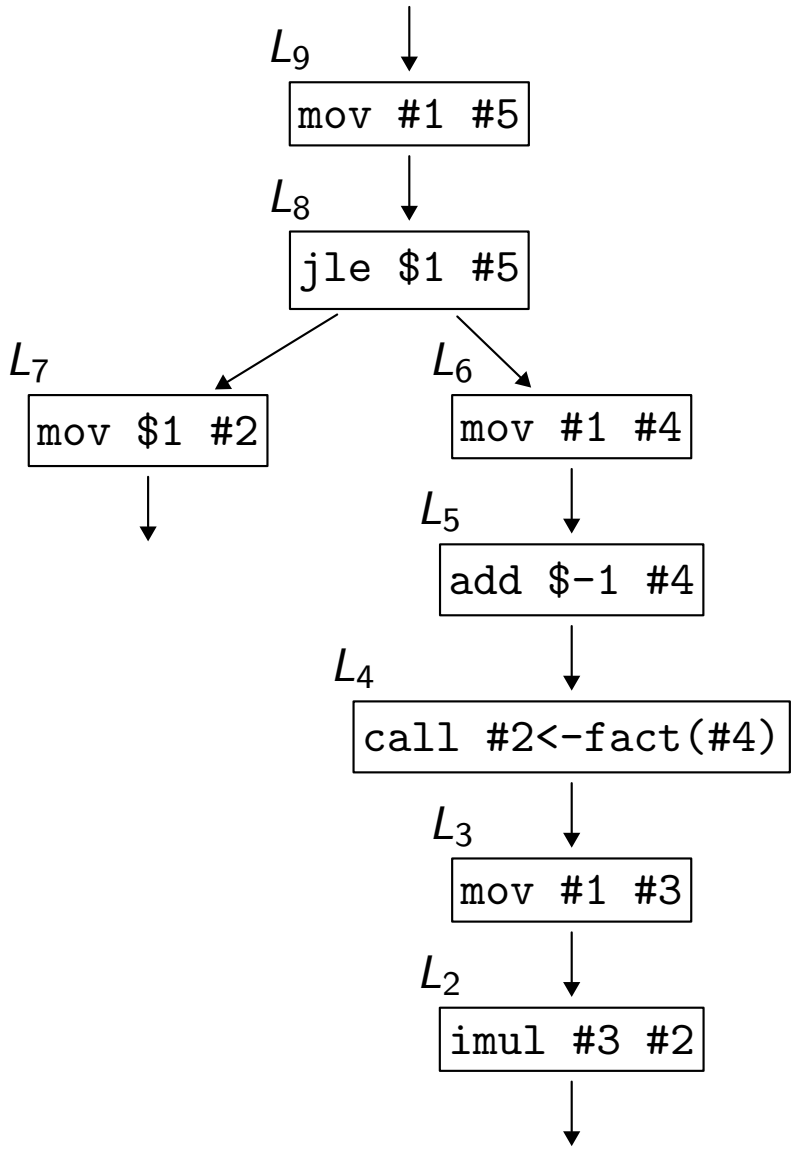
on obtient

```
#2 fact(#1)
  entry : L9
  exit  : L1
  locals:
  L9: mov #1 #5              --> L8
  L8: jle $1 #5        --> L7, L6
  L7: mov $1 #2              --> L1
  L6: mov #1 #4              --> L5
  L5: add $-1 #4             --> L4
  L4: call #2<-fact(#4)      --> L3
  L3: mov #1 #3              --> L2
  L2: imul #3 #2             --> L1
```

toujours avec la factorielle

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

on obtient



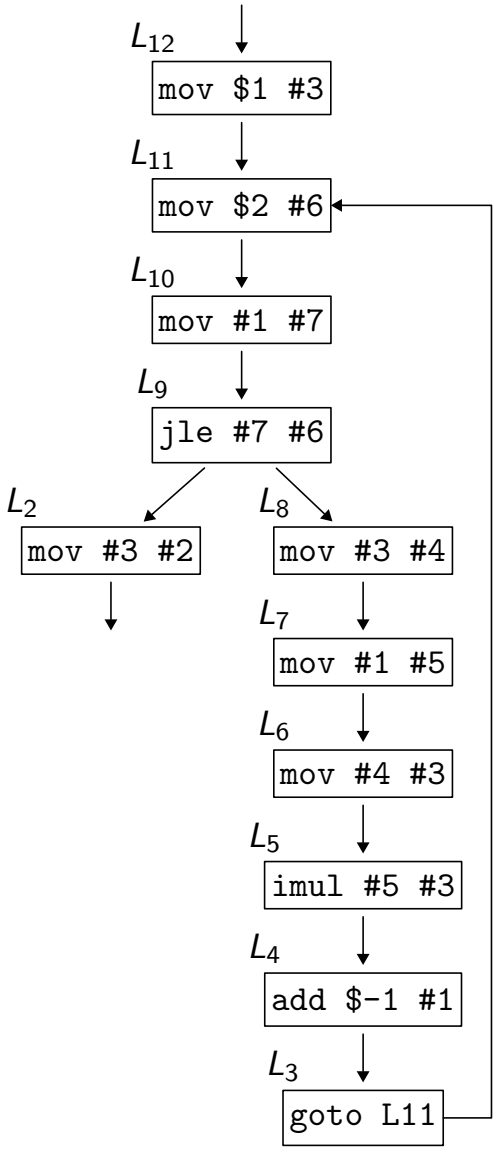
avec une boucle

```
int loop(int x) {
  int r;
  r = 1;
  while (2 <= x) {
    r = r * x;
    x = x - 1;
  }
  return r;
}
```

```
#2 loop(#1)
  entry : L12
  exit  : L1
  locals: #3
  L12: mov $1 #3   --> L11
  L11: mov $2 #6   --> L10
  L10: mov #1 #7   --> L9
  L9 : jle #7 #6   --> L8, L2
  L8 : mov #3 #4   --> L7
  L7 : mov #1 #5   --> L6
  L6 : mov #4 #3   --> L5
  L5 : imul #5 #3 --> L4
  L4 : add $-1 #1 --> L3
  L3 : goto L11
  L2 : mov #3 #2   --> L1
```

avec une boucle

```
int loop(int x) {
  int r;
  r = 1;
  while (2 <= x) {
    r = r * x;
    x = x - 1;
  }
  return r;
}
```

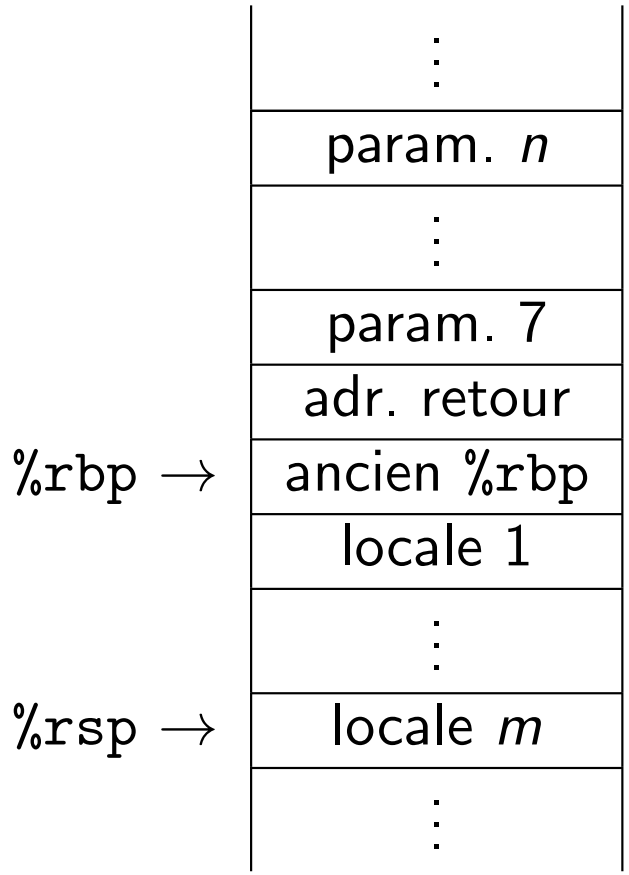


la troisième phase est la transformation de RTL vers le langage **ERTL** (*Explicit Register Transfer Language*)

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 `malloc` 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 ces 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

```
L4: #2 <- call fact(#4) --> L3
```

est traduit en ERTL par

```
L4 : goto           --> L12
L12: mov #4 %rdi    --> L11
L11: call fact(1)   --> L10
L10: mov %rax #2    --> L3
```

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    : ...
```

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

RTL

```
#2 fact(#1)
  entry : L9
  exit  : L1
  locals:




  L9: mov #1 #5          --> L8
  ...
```

ERTL

```
fact(1)
  entry : L16

  locals: #6, #7
  L16: alloc_frame   --> L15
  L15: mov %rbx #6   --> L14
  L14: mov %r12 #7   --> L13
  L13: mov %rdi #1   --> L9
  L9: mov #1 #5      --> L8
  ...
```

on suppose ici que les seuls registres *callee-saved* sont `%rbx` et `%r12` (en pratique, on a également `%r13`, `%r14`, `%r15`)

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 : L9
  exit  : L1
  locals:
  ...
  L7: mov $1 #2      --> L1
  ...
  L2: imul #3 #2     --> L1
```

ERTL

```
fact(1)
  entry : L16

  locals: #6, #7
  ...
  L7: mov $1 #2      --> L1
  ...
  L2: imul #3 #2     --> L1
  L1: goto           --> L21
  L21: mov #2 %rax   --> L20
  L20: mov #6 %rbx   --> L19
  L19: mov #7 %r12   --> L18
  L18: delete_frame --> L17
  L17: return
```

au total, on obtient le code ERTL suivant :

```
fact(1)
  entry : L16
  locals: #6,#7
  L16: alloc_frame  --> L15
  L15: mov %rbx #6  --> L14
  L14: mov %r12 #7  --> L13
  L13: mov %rdi #1  --> L9
  L9: mov #1 #5     --> L8
  L8: jle $1 #5  --> L7, L6
  L7: mov $1 #2     --> L1
  L1: goto          --> L21
  L21: mov #2 %rax  --> L20
```

```
  L20: mov #6 %rbx  --> L19
  L19: mov #7 %r12  --> L18
  L18: delete_frame --> L17
  L17: return
  L6: mov #1 #4     --> L5
  L5: add $-1 #4    --> L4
  L4: goto          --> L12
  L12: mov #4 %rdi  --> L11
  L11: call fact(1) --> L10
  L10: mov %rax #2  --> L3
  L3: mov #1 #3     --> L2
  L2: imul #3 #2    --> L1
```

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

  si par exemple on réalise #7 par `%r12`, on supprime tout simplement les deux instructions `L14` et `L19`

- 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

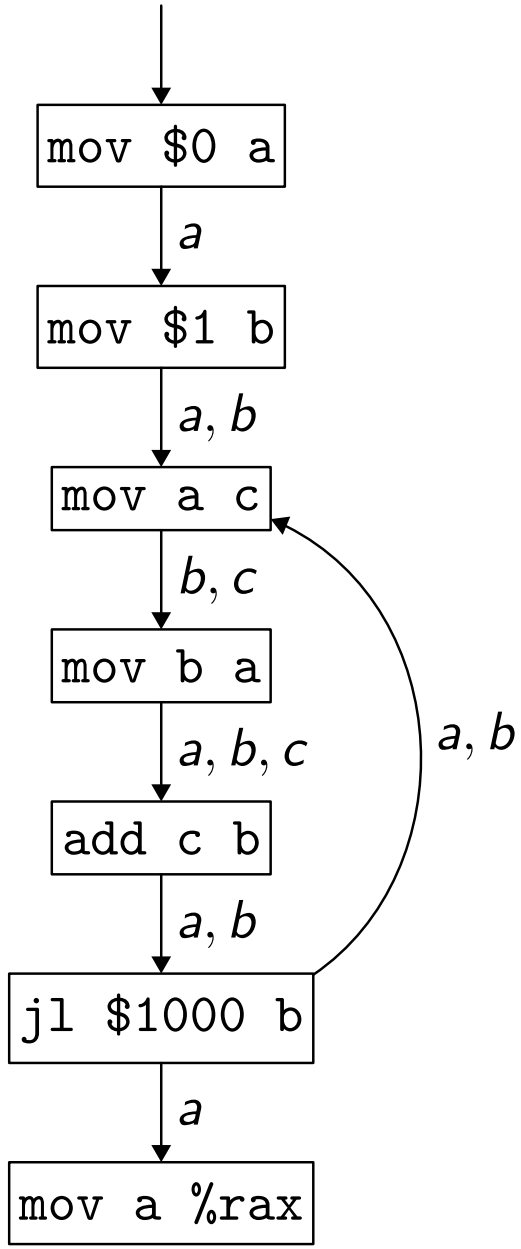
Définition (variable vivante)

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

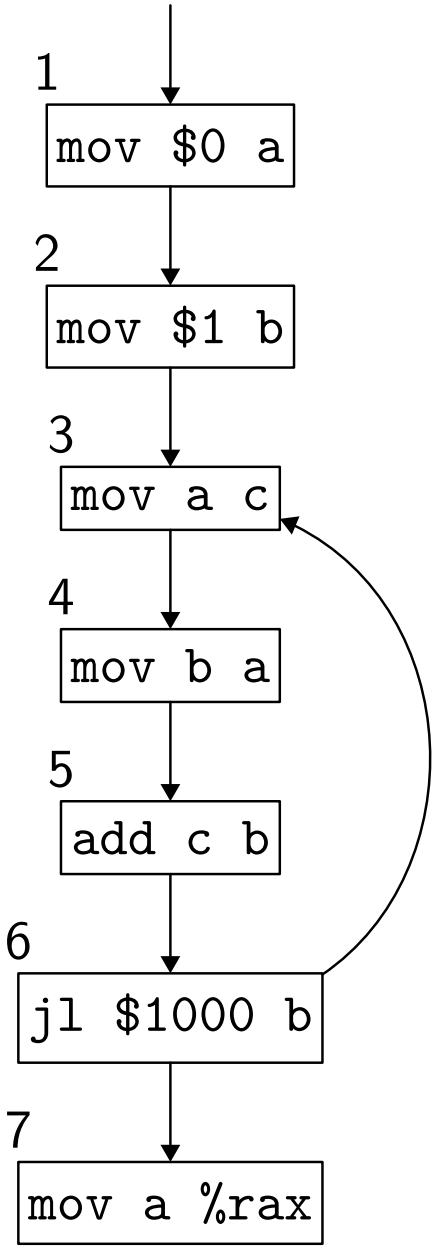
- *$in(l)$ l'ensemble des variables vivantes sur l'ensemble des arêtes arrivant sur l, et*
- *$out(l)$ l'ensemble des variables vivantes sur l'ensemble des arêtes sortant de l.*

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

sur l'exemple de la factorielle

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

la suite

- TD 7
  - projet Mini Go
- cours 8
  - compilateur optimisant 2/2