École Polytechnique

INF564 – Compilation

Jean-Christophe Filliâtre

production de code (1/3)

l'objectif des quatre derniers cours : 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
```

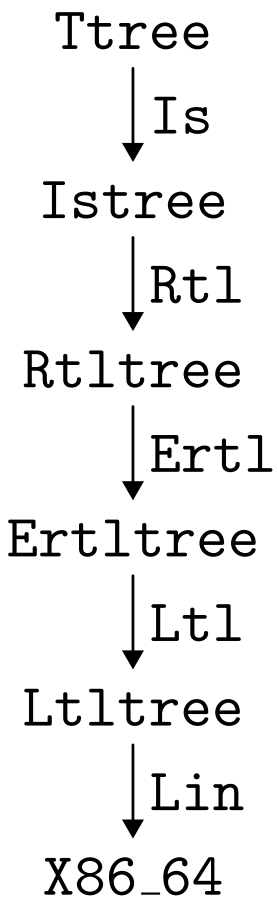
il est illusoire de chercher à produire du code efficace en une seule passe

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 `sbrk` (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 le type `int`
(donc de même taille qu'un pointeur)

mini-C (rappel)

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

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

```
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;
}
```

point de départ

on suppose le typage effectué

en particulier, on suppose connu le type de chaque expression

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
- 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

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

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

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      (* en OCaml *)
```

```
Expr mkAdd(Expr e1, Expr e2)          // en Java
```

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*

c'est à cette étape qu'il faudrait utiliser `lea` intelligemment (cf TD 1)

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))
```

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 :

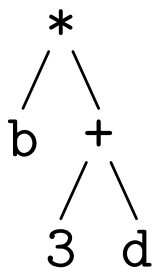
- détruire la structure arborescente des expressions et des instructions au profit d'un **graphe de flot de contrôle** (CFG en anglais), qui facilitera les phases ultérieures ; on supprime en particulier la distinction entre expressions et instructions

- 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)
```

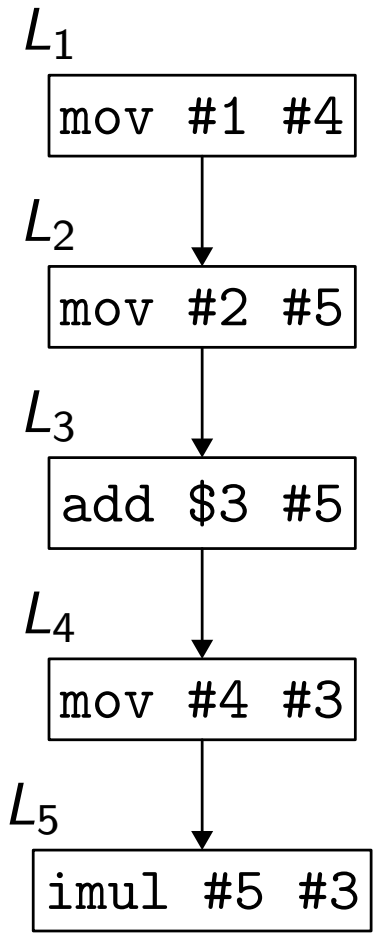
c'est-à-dire l'arbre



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

la traduction est relativement aisée

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

etc.

attention : il faut lire le code de bas en haut

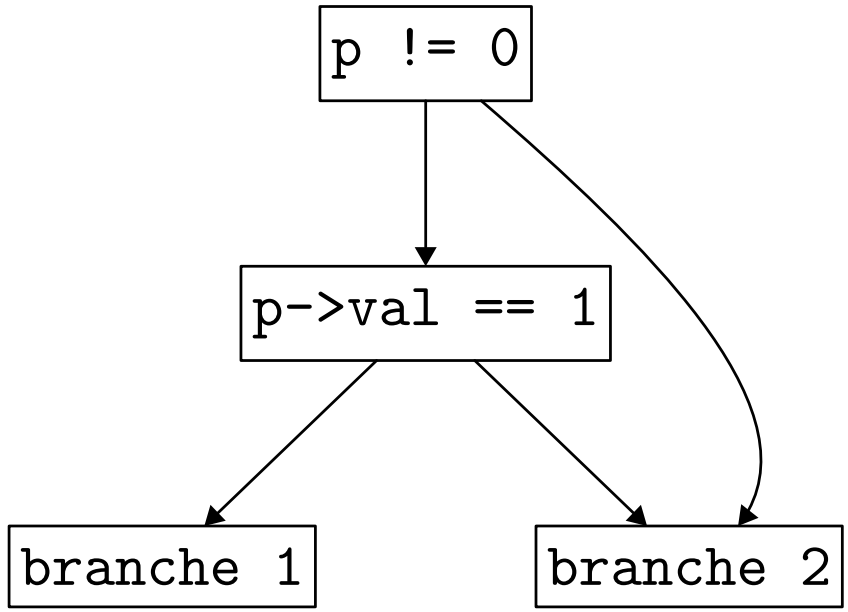
pour les variables locales, on se donne une table où chaque variable est associée à un pseudo-registre

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)

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)

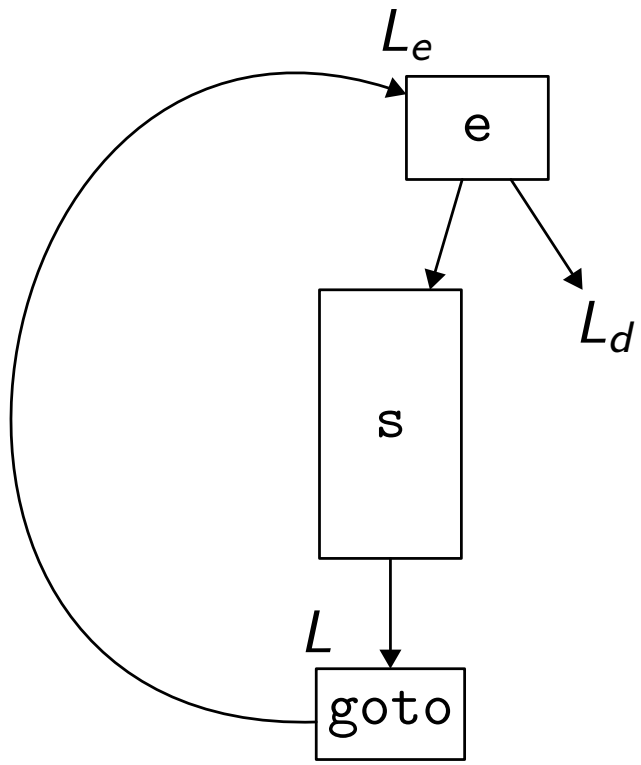
pour traduire `return`, on se donne un pseudo-registre r_{ret} pour recevoir le résultat de la fonction et une étiquette L_{ret} correspondant à la sortie de la fonction

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

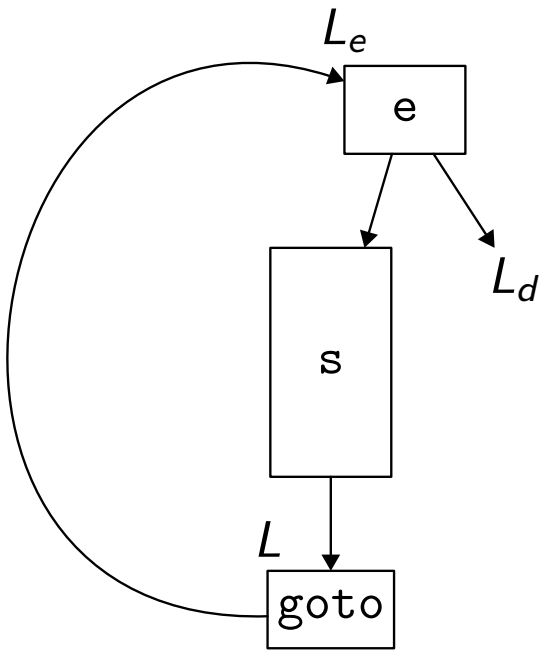
etc.

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

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



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

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

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 suite

- TD 6
  - production du code RTL
- prochain cours **lundi 24**
  - production de code efficace (2/4)

TD 6

---

production du code RTL

comme mini-C est très simple, on peut faire la sélection d'instructions et la traduction RTL en une seule étape

on peut se contenter d'une sélection d'instructions très naïve au départ (une opération C $\approx$ une opération x86-64) et l'améliorer plus tard

du code est fourni (pour OCaml et Java)

- opérations x86-64
- syntaxe abstraite de RTL
- interprète de code RTL pour tester
- affichage de code RTL pour débugger
- `main.ml` / `Main.java` pour la ligne de commande

```
./mini-c --debug --interp-rtl test.c
```

procéder incrémentalement, on ajoutant la traduction des constructions C une par une

le premier objectif doit être de traduire correctement

```
int main() {
  return 42;
}
```

ce qui doit donner quelque chose comme

```
#1 main()
  entry : L2
  exit  : L1
  locals:
  L2: mov $42 #1  --> L1
```