École Normale Supérieure

Langages de programmation et compilation

Jean-Christophe Filliâtre

Cours 11 / 11 décembre 2008

Rappel

la production de code efficace a été découpée en plusieurs phases :

1. sélection d'instructions
2. RTL (*Register Transfer Language*)
3. ERTL (*Explicit Register Transfer Language*)
4. LTL (*Location Transfer Language*)
   1. analyse de durée de vie
   2. construction d'un graphe d'interférence
   3. allocation de registres par coloration de graphe
5. code linéarisé
6. assembleur

Rappel

on a pris en exemple un fragment simple de C

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

Rappel

phase 1 : la sélection d'instructions

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

Rappel

phase 2 : RTL (*Register Transfer Language*)

```
%2 fact(%1)
  entry : L10
  exit  : L1
  locals :
  L10 : %7 <- %1  --> L9
  L9 : %8 <- 1  --> L8
  L8 : ble %7 %8 --> L7, L6
```

```
L7 : %2 <- 1  --> L1
L6 : %3 <- %1  --> L5
L5 : %6 <- %1  --> L4
L4 : %5 <- addi -1 %6  --> L3
L3 : %4 <- call fact(%5) --> L2
L2 : %2 <- mul %3 %4  --> L1
```

Rappel

phase 3 : ERTL (*Explicit Register Transfer Language*)

```
fact(1)
  entry : L18
  locals : %10,%11,%9
  L18 : alloc_frame  --> L17
  L17 : %9 <- $ra  --> L16
  L16 : %10 <- $s0  --> L15
  L15 : %11 <- $s1  --> L14
  L14 : %1 <- $a0  --> L10
  L10 : %7 <- %1  --> L9
  L9 : %8 <- 1  --> L8
  L8 : ble %7 %8  --> L7, L6
  L7 : %2 <- 1  --> L1
  L1 : goto L24
  L24 : $v0 <- %2  --> L23
```

```
L23 : $ra <- %9  --> L22
L22 : $s0 <- %10  --> L21
L21 : $s1 <- %11  --> L20
L20 : delete_frame  --> L19
L19 : return
L6 : %3 <- %1  --> L5
L5 : %6 <- %1  --> L4
L4 : %5 <- addi -1 %6  --> L3
L3 : goto L13
L13 : $a0 <- %5  --> L12
L12 : call fact(1)  --> L11
L11 : %4 <- $v0  --> L2
L2 : %2 <- mul %3 %4  --> L1
```

Phase 4 : LTL

phase 4 : LTL (*Location Transfer Language*)

on a déjà réalisé l'**analyse de durée de vie** *i.e.* on a déterminé pour chaque variable (pseudo-registre ou registre physique) à quels moments la valeur qu'elle contient peut être utilisée dans la suite de l'exécution

on va maintenant construire un **graphe d'interférence** qui exprime les contraintes sur les emplacements possibles pour les pseudo-registres

Interférence

Définition (interférence)

On dit que deux variables v_1 et v_2 **interfèrent** *si elles ne peuvent pas être réalisées par le même emplacement (registre physique ou emplacement mémoire).*

comme l'interférence n'est pas décidable, on va se contenter de conditions suffisantes

Interférence

soit une instruction

$$v \leftarrow e$$

qui définit une variable v ; toute autre variable w vivante à la sortie de cette instruction peut interférer avec v

cependant, dans le cas particulier d'un instruction `move`

$$v \leftarrow w$$

on ne souhaite pas déclarer que v et w interfèrent car il peut être précisément intéressant de réaliser v et w par le même emplacement et d'éliminer ainsi une ou plusieurs instructions

Graphe d'interférence

on adopte donc la définition suivante

Définition (graphe d'interférence)

Le graphe d'interférence d'une fonction est un graphe non orienté dont les sommets sont les variables de cette fonction et dont les arêtes sont de deux types : interférence ou préférence.
Pour chaque instruction qui définit une variable v et dont les variables vivantes en sortie, autres que v, sont $w_1, \ldots, w_n$, on procède ainsi :

- *si l'instruction n'est pas une instruction* `move`*, on ajoute les n arêtes d'interférence $v - w_i$*
- *s'il s'agit d'une instruction* `move` *de la forme $v \leftarrow w$, on ajoute les arêtes d'interférence $v - w_i$ pour tous les w_i différents de w et on ajoute l'arête de préférence $v - w$.*

(si une arête $v - w$ est à la fois de préférence et d'interférence, on conserve uniquement l'arête d'interférence)

Code

le graphe d'interférence peut être ainsi représenté en Caml :

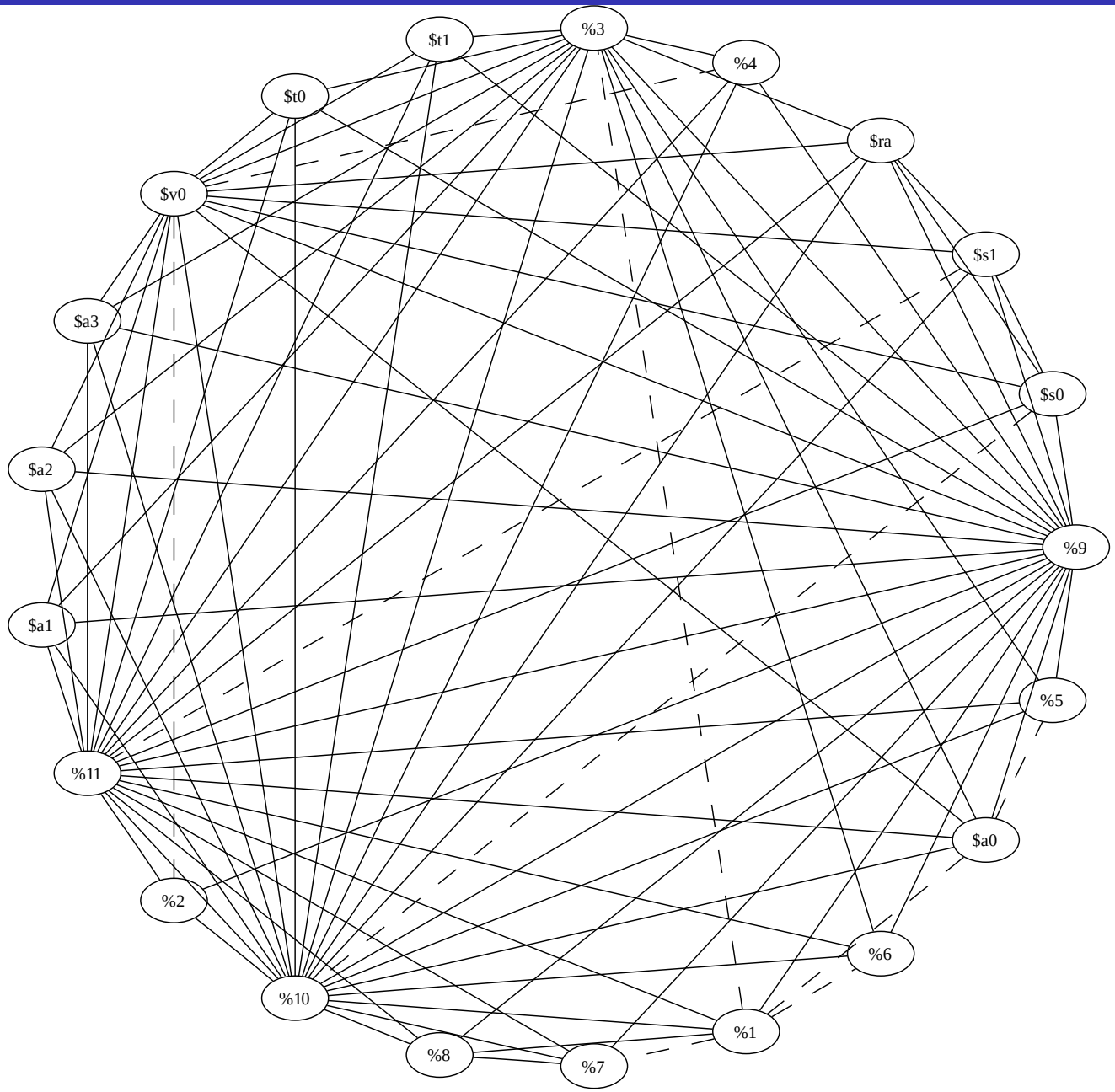
```
type arcs = { prefs : Register.set ; intfs : Register.set }

type graph = arcs Register.map
```

la fonction qui le construit est de la forme

```
  val make : Liveness.info Label.map -> graph
```

Exemple : factorielle

voici ce que l'on obtient pour la fonction `fact`



Coloriage

on peut alors voir le problème de l'allocation de registres comme un problème de **coloriage de graphe** :

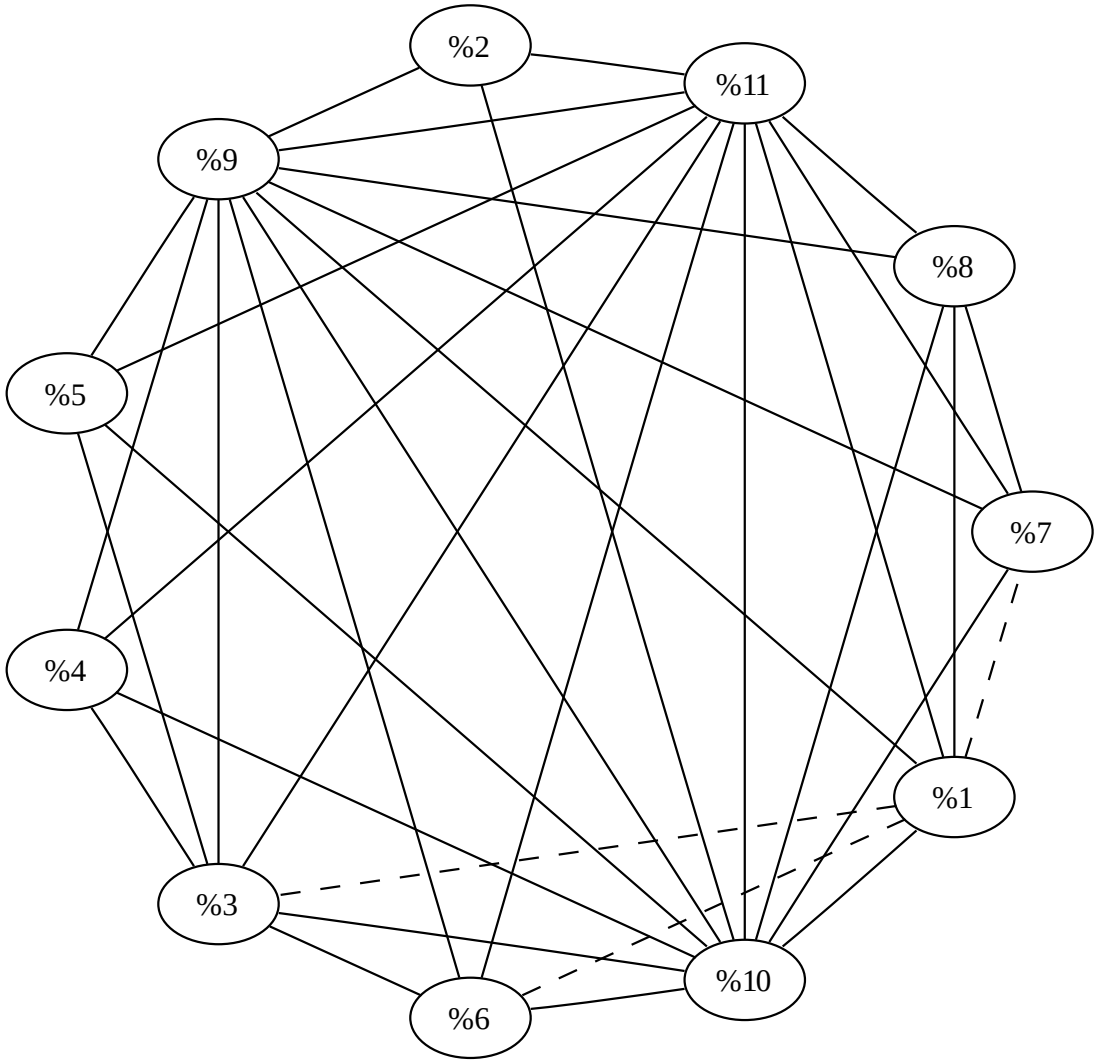
- les couleurs sont les registres physiques
- deux sommets liés par une arête d'interférence ne peuvent recevoir la même couleur
- deux sommets liés par une arête de préférence doivent recevoir la même couleur autant que possible

note : il y a dans le graphe des sommets qui sont des registres physiques, que l'on peut donc voir comme des sommets déjà coloriés

Exemple de la factorielle

si on élimine ces sommets déjà coloriés, il reste ce problème de coloriage

	couleurs possibles
%1	a_0, a_1, a_2, a_3, ra, s_0, s_1, t_0, t_1, v_0
%10	s_0
%11	s_1
%2	a_0, a_1, a_2, a_3, ra, s_0, s_1, t_0, t_1, v_0
%3	s_0, s_1
%4	a_0, a_1, a_2, a_3, ra, s_0, s_1, t_0, t_1, v_0
%5	a_0, a_1, a_2, a_3, ra, s_0, s_1, t_0, t_1, v_0
%6	a_0, a_1, a_2, a_3, ra, s_0, s_1, t_0, t_1, v_0
%7	a_0, a_1, a_2, a_3, ra, s_0, s_1, t_0, t_1, v_0
%8	a_0, a_1, a_2, a_3, ra, s_0, s_1, t_0, t_1, v_0
%9	



Difficulté

sur cet exemple, on voit tout de suite que le coloriage est impossible

- aucune couleur n'est disponible pour %9 (le pseudo-registre qui sauvegarde `$ra`)
- `$s0` et `$s1` sont les seules couleurs possibles pour %10 et %11, mais ensuite il ne reste plus de couleur pour %3

si un sommet ne peut être colorié, on décide qu'il correspondra à un emplacement de pile ; on dit qu'on *spill* le pseudo-registre

Autre difficulté

quand bien même le graphe serait effectivement coloriable, le déterminer serait trop coûteux (c'est un problème NP-complet)

on va donc colorier en utilisant des **heuristiques** et la solution de dernier recours consistant à *spiller* certains sommets

Une heuristique très simple

voici une heuristique très simple (utilisée dans le compilateur mini-c)

1. si un sommet a une seule couleur possible qui est également une préférence, on la prend
2. sinon, si un sommet a une seule couleur possible, on la prend
3. sinon, si un sommet a une préférence dont la couleur est connue, on prend cette couleur si possible
4. sinon, si un sommet a encore une couleur possible, on la prend
5. sinon, on *spill* un sommet quelconque

Et les pseudo-registres *spillés* ?

que fait-on des pseudo-registres *spillés* ?

on leur associe des emplacements sur la pile, dans la zone basse du tableau d'activation (en dessous des paramètres)

plusieurs pseudo-registres peuvent occuper le même emplacement de pile, s'ils n'interfèrent pas $\Rightarrow$ comment utiliser au mieux l'espace de pile ?

c'est de nouveau un problème de coloriage de graphe, mais cette fois avec une infinité de couleurs possibles (chaque couleur correspondant à un emplacement de pile différent)

là encore, on emploie des heuristiques, pour limiter l'espace utilisé mais aussi pour (tenter de) satisfaire les préférences

En pratique

en pratique, on utilise des techniques bien plus sophistiquées (mais qui restent de complexité linéaire ou quasi-linéaire), notamment pour bien exploiter les arêtes de préférence

l'un des meilleurs algorithmes est dû à George et Appel (*Iterated Register Coalescing*, 1996) ; cf Appel, chapitre 11

cet algorithme exploite les idées suivantes

Simplification

soit K le nombre de couleurs (*i.e.* le nombre de registres physiques)

une première idée, due à Kempe (1879 !), est la suivante : si un sommet a un degré $< K$, alors on peut le retirer du graphe, colorier le reste, et on sera ensuite assuré de pouvoir lui donner une couleur ; cette étape est appelée **simplification**

on répète cette opération autant que possible (retirer un sommet diminue le degré d'autres sommets et peut donc produire de nouveaux candidats à la simplification) ; les sommets retirés sont donc stockés sur une pile

Spill

lorsqu'il ne reste que des sommets de degré $\geq K$, on en choisit un comme **candidat au spilling** (*potential spill*) ; il est alors retiré du graphe, mis sur la pile et le processus de simplification peut reprendre

on choisit de préférence un sommet qui

- est peu utilisé (les accès à la mémoire coûtent cher)
- a un fort degré (pour favoriser de futures simplifications)

Sélection

lorsque le graphe est vide, on commence le processus de coloration, appelé **sélection**

on dépile les sommets un à un et pour chacun

- s'il s'agit d'un sommet de faible degré, on est assuré de lui trouver une couleur
- s'il s'agit d'un sommet de fort degré, c'est-à-dire d'un candidat au spilling, alors
  - soit il peut être tout de même colorié car ses voisins utilisent moins de K couleurs ; on parle de **coloriage optimiste**
  - soit il ne peut être colorié et doit être effectivement spillé (on parle d'*actual spill*)

Spill effectif

lorsqu'un pseudo-registre se retrouve effectivement spillé, il faut transformer le programme pour aller chercher en mémoire à chaque utilisation de ce pseudo-registre et pour écrire en mémoire à chaque définition de ce pseudo-registre

mais pour cela il faut des registres physiques !

on introduit donc de nouveaux temporaires, on modifie le graphe d'interférence et on recommence tout ; heureusement, cela converge très rapidement en pratique (2 ou 3 étapes seulement)

Coalescence

enfin, il convient d'utiliser au mieux les arêtes de préférence

pour cela, on utilise une technique appelée **coalescence** (*coalescing*) qui consiste à fusionner deux sommets du graphe ; comme cela peut augmenter le degré du sommet résultant, on ajoute un critère suffisant pour ne pas détériorer la k-colorabilité

cette phase s'insère après la simplification et déclenche une nouvelle simplification en cas de coalescence

pour une description complète de l'algorithme, et son pseudo-code, voir le chapitre 11 du Appel

Code

le résultat de l'allocation de registres a la forme suivante

```
type color = Spilled of int | Reg of Register.t

type coloring = color Register.map
```

la fonction d'allocation se présente alors ainsi

```
  val alloc_registers : Interference.graph -> coloring
```

Exemple

pour la factorielle, on obtient l'allocation de registres suivante

```
  %1 -> $a0
 %10 -> $s0
 %11 -> $s1
  %2 -> $v0
  %3 -> stack 4
  %4 -> $v0
  %5 -> $a0
  %6 -> $a0
  %7 -> $a0
  %8 -> $a1
  %9 -> stack 0
```

Exemple

ce qui *donnerait* le code suivant

```
fact(1)
  entry : L18
  locals : %10,%11,%9
  L18 : alloc_frame  --> L17
  L17 : stack(0) <- $ra --> L16
  L16 : $s0 <- $s0  --> L15
  L15 : $s1 <- $s1  --> L14
  L14 : $a0 <- $a0  --> L10
  L10 : $a0 <- $a0  --> L9
  L9 : $a1 <- 1  --> L8
  L8 : ble $a0 $a1  --> L7, L6
  L7 : $v0 <- 1  --> L1
  L1 : goto L24
  L24 : $v0 <- $v0  --> L23
```

```
L23 : $ra <- stack(0) --> L22
L22 : $s0 <- $s0  --> L21
L21 : $s1 <- $s1  --> L20
L20 : delete_frame  --> L19
L19 : return
L6 : stack(4) <- $a0  --> L5
L5 : $a0 <- $a0  --> L4
L4 : $a0 <- addi -1 $a0 --> L3
L3 : goto L13
L13 : $a0 <- $a0  --> L12
L12 : call fact  --> L11
L11 : $v0 <- $v0  --> L2
L2 : $v0 <- mul stack(4) $v0
      --> L1
```

Remarques

comme on le constate, de nombreuses instructions de la forme

$$v \leftarrow v$$

peuvent être éliminées ; c'était l'intérêt des arêtes de préférence

à l'inverse, on constate que des instructions comme

$$\texttt{\$v0} \leftarrow \texttt{mul stack(0) \$v0}$$

apparaissent, qui n'ont pas d'équivalent MIPS ; que faire ?

ces deux problèmes vont être traités dans la traduction vers LTL

Le langage LTL

la plupart des instructions LTL sont les mêmes que celles de ERTL, si ce n'est que les registres sont maintenant tous des registres physiques

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

seules `Eget_stack_param` et `Eset_stack_param` disparaissent, au profit de deux instructions plus générales manipulant la pile sur la base de `$sp`

```
  | Eget_stack of register * int * label
  | Eset_stack of register * int * label
```

Traduction ERTL vers LTL

on traduit chaque instruction ERTL à l'aide d'une fonction qui prend en argument le coloriage du graphe d'une part, et la taille du tableau d'activation d'autre part (qui est maintenant connue pour chaque fonction)

```
let instr colors frame_size = function
  | ...
```

Traduction ERTL vers LTL

considérons la toute première instruction, qui charge la constante `n` dans la variable `r` :

```
let instr colors frame_size = function
  | Ertltree.Econst (r, n, l) ->
      ?
```

de deux choses l'une

- soit `r` est directement un registre physique `h` ou un pseudo-registre réalisé par un registre physique `h` et la traduction est alors immédiate

  ```
  Econst (h, n, l)
  ```

- soit `r` est un pseudo-registre spillé, et la traduction est alors plus complexe : il faut charger `n` dans un registre physique, puis utiliser ce registre physique pour écrire en mémoire
  problème : quel registre physique utiliser ?

Registres temporaires

en adopte ici une solution simple : deux registres particuliers seront utilisés comme registres temporaires pour ces transferts avec la mémoire, et ne seront pas utilisés par ailleurs
(en l'occurrence on choisit ici d'utiliser `$v1` et `$fp`)

en pratique, on n'a pas nécessairement le loisir de gâcher ainsi deux registres ; deux solutions au moins

- un jeu d'instructions complexe (CISC) peut proposer plusieurs modes d'adressage, en particulier des opérations arithmétiques manipulant la mémoire

```
        add [ebp-8], ecx
```

- on peut modifier le graphe d'interférence et relancer une allocation de registres pour déterminer un registre libre pour le transfert (cf plus haut)

Traduction ERTL vers LTL

on se donne donc les deux registres temporaires

```
let tmp1, tmp2 = "$v1", "$fp"
```

pour écrire dans la variable `r`, on se donne une fonction `write`, qui prend également en arguments le coloriage `c` et l'étiquette où il faut aller après l'écriture ; elle renvoie le registre physique dans lequel il faut effectivement écrire et l'étiquette où il faut effectivement aller ensuite

```
let write c r l = match lookup c r with
  | Reg hr -> hr, l
  | Spilled n -> tmp1, generate (Eset_stack (tmp1, n, l))
```

Traduction ERTL vers LTL

on peut maintenant traduire facilement de ERTL vers LTL

```
let instr c frame_size = function
  | Ertltree.Econst (r, n, l) ->
      let hr, l = write c r l in
      Econst (hr, n, l)
  | Ertltree.Eaccess_global (r, x, l) ->
      let hwr, l = write c r l in
      Eaccess_global (hwr, x, l)
  | ...
```

Traduction ERTL vers LTL

inversement, on se donne une fonction `read1` pour lire le contenu d'une variable

```
let read1 c r f = match lookup c r with
  | Reg hr -> f hr
  | Spilled n -> Eget_stack (tmp1, n, generate (f tmp1))
```

et on l'utilise ainsi

```
let instr c frame_size = function
  | ...
  | Ertltree.Eassign_global (r, x, l) ->
      read1 c r (fun hwr -> Eassign_global (hwr, x, l))
```

Traduction ERTL vers LTL

on se donne de même une fonction `read2` pour lire le contenu de deux variables

et on l'utilise ainsi

```
| Ertltree.Embinop (r1, op, r2, r3, l) ->
   read2 c r2 r3 (fun hw2 hw3 ->
   let hw1, l = write c r1 l in Embinop (hw1, op, hw2, hw3, l))
```

L'opération move

on applique un traitement spécial à l'instruction Mmove

```
| Ertltree.Emunop (r1, Mmove, r2, l) ->
   begin match lookup c r1, lookup c r2 with
     | w1, w2 when w1 = w2 ->
        Egoto l
     | Reg hr1, Reg hr2 ->
        Emunop (hr1, Mmove, hr2, l)
     | Reg hr1, Spilled ofs2 ->
        Eget_stack (hr1, ofs2, l)
     | Spilled ofs1, Reg hr2 ->
        Eset_stack (hr2, ofs1, l)
     | Spilled ofs1, Spilled ofs2 ->
        Eget_stack (tmp1, ofs2, generate (
        Eset_stack (tmp1, ofs1, l)))
   end
```

Paramètres sur la pile

maintenant que l'on connaît la taille du tableau d'activation, on peut traduire `Eget_stack_param` en terme d'accès par rapport à `$sp`

```
  | Ertltree.Eget_stack_param (r, n, l) ->
     let hwr, l = write c r l in
     Eget_stack (hwr, frame_size + n, l)
```

en revanche, `Eset_stack_param` ne change pas

```
  | Ertltree.Eset_stack_param (r, n, l) ->
     read1 c r (fun hwr -> Eset_stack (hwr, n, l))
```

Tableau d'activation

on peut enfin traduire `Ealloc_frame` et `Edelete_frame` en terme de manipulation de `$sp`

```
| Ertltree.Ealloc_frame l
| Ertltree.Edelete_frame l when frame_size = 0 ->
    Egoto l
| Ertltree.Ealloc_frame l ->
    Emunop (Register.sp, Maddi (- frame_size), Register.sp, l)
| Ertltree.Edelete_frame l ->
    Emunop (Register.sp, Maddi frame_size, Register.sp, l)
```

Traduction ERTL vers LTL

on n'a plus qu'à assembler tous les morceaux

```
let deffun f =
  let ln = Liveness.analyze f.Ertltree.fun_body in
  let ig = Interference.make ln in
  let c, nlocals = Coloring.find ig in
  let n_stack_params =
    max 0 (f.Ertltree.fun_formals - List.length Register.parameters)
  in
  let frame_size = word_size * (nlocals + n_stack_params) in
  graph := Label.M.empty;
  Label.M.iter (fun l i ->
    let i = instr c frame_size i in graph := Label.M.add l i !graph)
    f.Ertltree.fun_body;
  { fun_name = f.Ertltree.fun_name;
    fun_entry = f.Ertltree.fun_entry;
    fun_body = !graph; }
```

Exemple

pour la factorielle, on obtient le code LTL suivant

```
fact__1()
  entry : L18
  L18 : $sp <- addi -8 $sp  --> L17
  L17 : stack(0) <- $ra  --> L16
  L16 : goto L15
  L15 : goto L14
  L14 : goto L10
  L10 : goto L9
  L9 : $a1 <- 1  --> L8
  L8 : ble $a0 $a1  --> L7, L6
  L7 : $v0 <- 1  --> L1
  L1 : goto L24
  L24 : goto L23
  L23 : $ra <- stack(0)  --> L22
  L22 : goto L21
  L21 : goto L20
  L20 : $sp <- addi 8 $sp  --> L19
  L19 : return
  L6 : stack(4) <- $a0  --> L5
  L5 : goto L4
  L4 : $a0 <- addi -1 $a0  --> L3
  L3 : goto L13
  L13 : goto L12
  L12 : call fact__1  --> L11
  L11 : goto L2
  L2 : $v1 <- stack(4)  --> L25
  L25 : $v0 <- mul $v1 $v0  --> L1
```

Phase 5 : linéarisation

il reste une dernière étape : le code est toujours sous la forme d'un **graphe de flot de contrôle** et l'objectif est de produire du **code assembleur linéaire**

plus précisément : les instructions de branchement de LTL contiennent deux étiquettes, l'une en cas de test positif et l'autre en cas de test négatif, alors que les instructions de branchement de l'assembleur contiennent une unique étiquette pour le cas positif et l'exécution se poursuit sur l'instruction suivante en cas de test négatif

note : on va produire directement de l'assembleur MIPS dans cette phase et donc omettre la phase 6 (production d'assembleur) ; on pourrait cependant distinguer les deux dans un contexte plus complexe

Linéarisation

la linéarisation consiste à parcourir le graphe de flot de contrôle et à produire le code MIPS tout en notant dans une table les étiquettes déjà visitées

lors d'un branchement, on s'efforce autant que possible de produire le code assembleur naturel si la partie du code correspondant à un test négatif n'a pas encore été visitée ; dans le pire des cas, on utilise un branchement inconditionnel

Linéarisation

le code MIPS est produit séquentiellement à l'aide d'une fonction

```
val emit : Label.t -> Mips.instruction -> unit
```

Linéarisation

on utilise deux tables

une première pour les étiquettes déjà visitées

```
let visited = Hashtbl.create 17
```

et une seconde pour les étiquettes qui devront rester dans le code assembleur (on ne le sait pas au moment même où une instruction assembleur est produite)

```
let labels = Hashtbl.create 17
let need_label l = Hashtbl.add labels l ()
```

Linéarisation

la linéarisation est effectuée par deux fonctions mutuellement récursives

- `lin` produit le code à partir d'une étiquette donnée, s'il n'a pas déjà été produit, et une instruction de saut vers cette étiquette sinon

```
val lin : instr Label.map -> Label.t -> unit
```

- `instr` produit le code à partir d'une étiquette et de l'instruction correspondante, sans condition

```
val instr : instr Label.map -> Label.t -> instr -> unit
```

Linéarisation

la fonction `lin` est un simple parcours de graphe

si l'instruction n'a pas déjà été visitée, on la marque comme visitée et on appelle `instr`

```
let rec lin g l =
  if not (Hashtbl.mem visited l) then begin
    Hashtbl.add visited l () ;
    instr g l (Label.M.find l g)
```

sinon on marque son étiquette comme requise dans le code assembleur et on produit un saut inconditionnel vers cette étiquette

```
   end else begin
    need_label l ;
    emit (Label.fresh ()) (B l)
  end
```

Linéarisation

la fonction `instr` produit effectivement le code MIPS et rappelle récursivement `lin` sur l'étiquette suivante

```
and instr g l = function
  | Econst (r, n, l1) ->
      emit l (Li (r, n)) ; lin g l1
  | Eaccess_global (r, x, l1) ->
      emit l (Lw (r, Alab x)) ; lin g l1
  | ...
```

Branchements

le cas intéressant est celui d'un branchement (on considère ici `Emubranch` ; c'est identique pour `Embbranch`)

on considère d'abord le cas favorable où le code correspondant à un test négatif n'a pas encore été produit

```
  | Emubranch (br, r, lt, lf)
    when not (Hashtbl.mem visited lf) ->
      need_label lt ;
      emit l (ubranch br r lt) ;
      lin g lf ;
      lin g lt
```

(où `ubranch` est la fonction qui produit l'instruction MIPS de branchement)

Branchements

sinon, il est possible que le code correspondant à un test positif n'ait pas encore été produit et on peut alors avantageusement **inverser la condition** de branchement

```
  | Emubranch (br, r, lt, lf)
    when not (Hashtbl.mem visited lt) ->
      instr g l (Emubranch (inv_ubranch br, r, lf, lt))
```

où

```
let inv_ubranch = function
  | Mbeqz -> Mbnez
  | Mbnez -> Mbeqz
  | ...
```

Branchements

enfin, dans le cas où le code correspondant aux deux branches a déjà été produit, on n'a pas d'autre recours que de produire un branchement inconditionnel

```
  | Emubranch (br, r, lt, lf) ->
      need_label lt ; need_label lf ;
      emit l (ubranch br r lt) ;
      emit l (B lf)
```

note : on peut essayer d'estimer la condition qui sera vraie le plus souvent

goto

le code contient de nombreux `goto` (boucles `while` dans la phase RTL, insertion de code dans la phase ERTL, suppression d'instructions `move` dans la phase LTL)

on élimine ici les `goto` lorsque c'est possible

```
  | Egoto l1 ->
      if Hashtbl.mem visited l1 then begin
        need_label l1;
        emit l (B l1)
      end else begin
        emit l Nop;
        lin g l1
      end
```

On rassemble tout

le programme principal enchaîne toutes les phases de la compilation

```
let f = open_in file in
let buf = Lexing.from_channel f in
let p = Parser.file Lexer.token buf in
close_in f ;
let p = Typing.program p in
let p = Is.program p in
let p = Rtl.program p in
let p = Ertl.program p in
let p = Ltl.program p in
let code = Lin.program p in
let c = open_out (Filename.chop_suffix file ".c" ^ ".s") in
let fmt = formatter_of_out_channel c in
Mips.print_program fmt code ;
close_out c
```

et voilà !

Factorielle

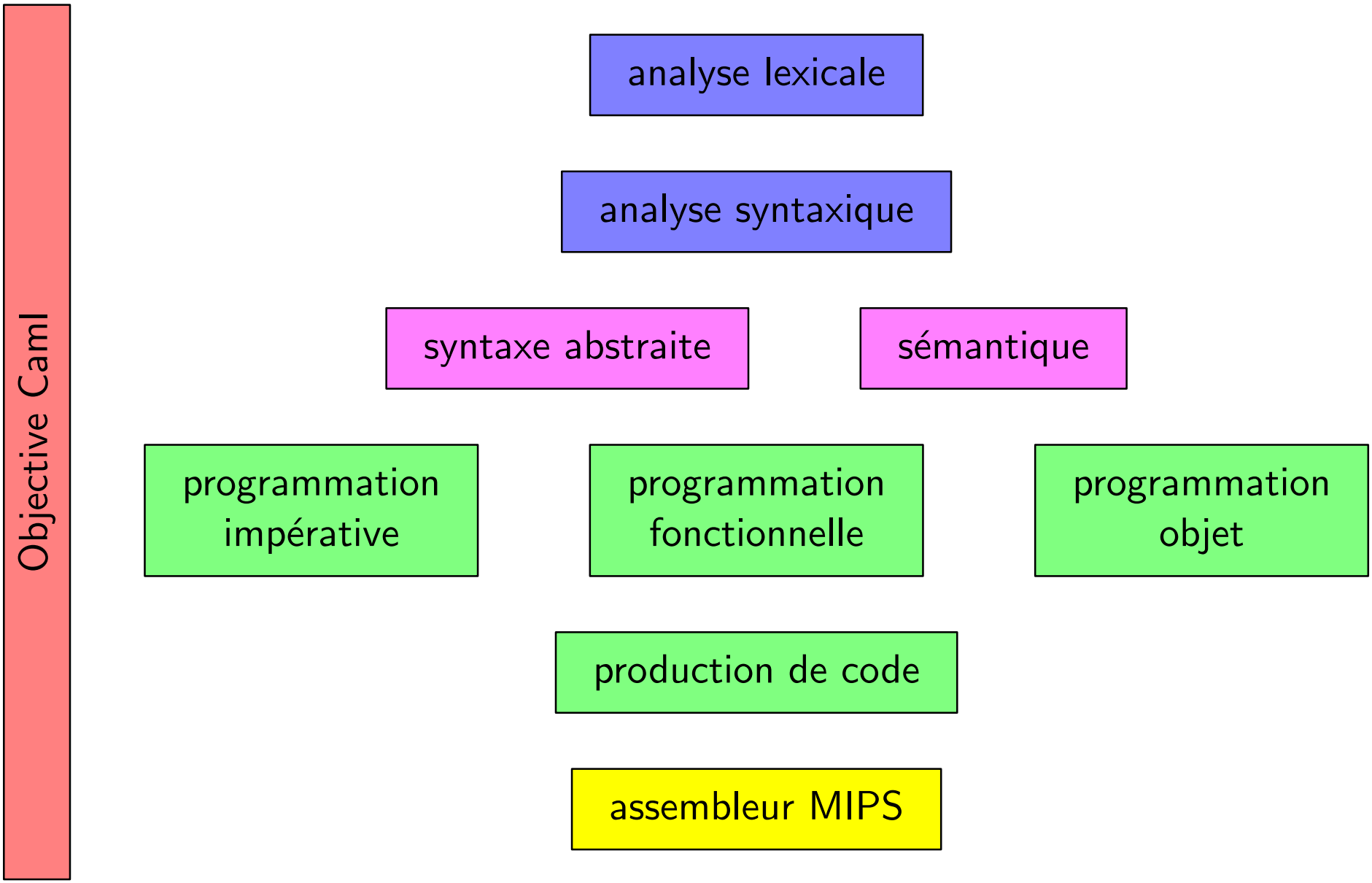
```
fact__1:
        add   $sp, $sp, -8
        sw    $ra, 0($sp)
        li    $a1, 1
        ble   $a0, $a1, L27
        sw    $a0, 4($sp)
        add   $a0, $a0, -1
        jal   fact__1
        lw    $v1, 4($sp)
        mul   $v0, $v1, $v0
L21:
        lw    $ra, 0($sp)
        add   $sp, $sp, 8
        jr    $ra
L27:
        li    $v0, 1
        b     L21
```

Taille de mini-C

	lignes de code
parsing	224
typage	169
sélection d'instruction	97
RTL	122
ERTL	103
LTL	113
durée de vie	91
interférence	46
coloriage	73
linéarisation	111
divers	52
total	**1201**

Le dragon est mort

Récapitulation



Mais aussi...

systèmes de modules
SSA
sous-expressions communes
transformation de programmes
interprétation abstraite
analyse d'alias
dépliage de boucles
analyse interprocédurale
optimisation à lucarne
pipeline
mémoire cache
programmation logique
programmation dynamique
ordonnancement des instructions
etc.

La suite

la semaine prochaine

- TD 11 le mercredi 17 décembre
  - aide au projet

ensuite

- TD 12 le mercredi 7 janvier
  - aide au projet

- projet à rendre pour le jeudi 15 janvier

- examen le jeudi 22 janvier, de 14h à 17h, en salle U/V