Universidade da Beira Interior

Desenho de Linguagens de Programação e de Compiladores

Simão Melo de Sousa

Aula 11 - Produção de código eficiente, parte 1

o objetivo deste capitulo é a produção de **código eficaz**

até aqui utilizamos muito mal as capacidades oferecidas pelo *assembly* X86-64 :

- muito poucos registos utilizados, embora haja 16 registos disponíveis
  - argumentos e variáveis locais sistematicamente na pilha
  - cálculos intermédios na pilha, igualmente

- instruções mal utilizadas
  - exemplo : nunca usamos

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

(utilizávamos um registo temporário e a pilha !)

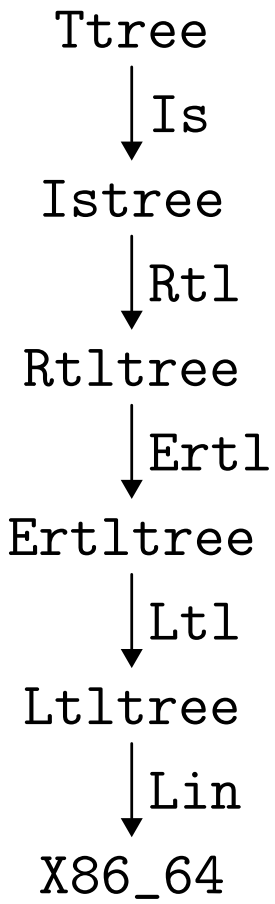
é ilusório procurar produzir código eficaz numa só fase

a produção de código será descomposta em **várias fases**

1. selecção de instruções
2. RTL (*Register Transfer Language*)
3. ERTL (*Explicit Register Transfer Language*)
4. LTL (*Location Transfer Language*)
5. código linearizado (*assembly*)

fases do compilador (código OCaml)

o ponto de partida é a árvore de sintaxe abstracta proveniente da tipagem



esta arquitectura da compilação é viável para todos os grandes paradigmas da programação (imperativa, funcional, objeto, etc.)

para a ilustrar, faremos no entanto a escolha de uma linguagem particular, neste caso um pequeno fragmento da linguagem C

consideramos um fragmento muito simples da linguagem **C** com

- inteiros (tipo `int`)
- estruturas alocadas na *heap* com `malloc` (somente apontadores sobre estas estruturas, exclusão da aritmética de apontadores)
- funções
- uma « primitiva » para mostrar um inteiro : `printf("%d\n",`e`)`

$$
\begin{array}{rcl}
E & \rightarrow & n \\
 & | & L \\
 & | & L = E \\
 & | & E\ op\ E \mid \texttt{-}\ E \mid \texttt{!}\ E \\
 & | & x(E, \ldots, E) \\
 & | & \texttt{malloc(sizeof(struct}\ x\texttt{))} \\
L & \rightarrow & x \\
 & | & E\texttt{->}x \\
op & \rightarrow & \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
 & | & \texttt{\&\&} \mid \texttt{||} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \\
D & \rightarrow & V \\
 & | & T\ x(T\ x, \ldots, T\ x)\ B \\
 & | & \texttt{struct}\ x\ \{V \ldots V\}; \\
S & \rightarrow & E\texttt{;} \\
 & | & \texttt{if}\ (E)\ S \\
 & | & \texttt{if}\ (E)\ S\ \texttt{else}\ S \\
 & | & \texttt{while}\ (E)\ S \\
 & | & \texttt{return}\ E\texttt{;} \\
 & | & \texttt{printf("\%d\textbackslash n",}E\texttt{);} \\
 & | & B \\
B & \rightarrow & \{\ V \ldots V\ S \ldots S\ \} \\
V & \rightarrow & \texttt{int}\ x, \ldots, x\texttt{;} \\
 & | & \texttt{struct}\ x\ \texttt{*}x, \ldots, \texttt{*}x\texttt{;} \\
T & \rightarrow & \texttt{int} \mid \texttt{struct}\ x\ \texttt{*} \\
P & \rightarrow & D \ldots D
\end{array}
$$

```
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) {
    printf("%d\n", l->val);
    l = l->next;
  }
  return 0;
}
```

supomos que a realização prévia da análise semântica ; a árvore resultante é a seguinte :

```
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
```

nas expressões, já distinguimos as variáveis locais e globais,
e os seus nomes são únicos

```
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          (* índice do campo *)
  | Eassign_field   of expr * int * expr
  | Eunop           of unop * expr                    (* - ! *)
  | Ebinop          of binop * expr * expr  (* + - == etc. *)
  | Ecall           of ident * expr list
  | Emalloc         of structure
```

a primeira fase é o da **seleção de instrução**

objectivo :

- substituir as operações aritmética do C pelas do X86-64
- substituir o acesso aos campos de estruturas por operações `mov`

podemos ingenuamente traduzir cada operação aritmética de C por pela instrução X86-64 correspondente

no entanto X86-64 fornece instruções permitindo uma bem maior eficiência, em particular

- soma de um registo e de uma constante
- *shift* dos bits para a esquerda ou para a direita, correspondendo a uma multiplicação ou uma divisão por uma potência de dois
- comparação de um registo com uma constante

É igualmente conveniente avaliar quanto mais expressões quanto possível durante a compilação (avaliação parcial)

exemplos : podemos simplificar

- $(1 + e_1) + (2 + e_2)$ em $e_1 + e_2 + 3$
- $e + 1 < 10$ em $e < 9$
- $!(e_1 < e_2)$ em $e_1 \geq e_2$
- $0 \times e$ em 0, só se e é **pura** *i.e.* sem efeitos laterais

damo-nos novas árvores para o resultado da seleção de instrução

`Ttree.mli` (antes)

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

`Istree.mli` (após)

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

o objectivo é escrever as funções

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

as operações são agora as do X86-64

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

e elas substituem as operações do C

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

conservamos no entanto && e || por enquanto

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

para realizar a avaliação parcial, vamos utilizar o que se designa de *smart constructors*

em vez de escrever `Embinop (Madd, e1, e2)` directamente, vamos introduzir uma função

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

que efectua eventuais simplificações e que se comporta como o construtor, senão

aqui vão algumas simplificações possíveis para a soma

```
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)
```

dois aspectos são essenciais no que diz respeito a estas simplificações

- a semântica dos programas deve ficar preservada
  - exemplo : se uma ordem de avaliação esquerdo/direito está especificado, não se pode simplificar $(0 - e_1) + e_2$ em $e_2 - e_1$ se e_1 ou e_2 não é pura

- a função de simplificação deve terminar
  - é preciso encontrar uma medida positiva sobre a expressão simplificada que diminuí estritamente a cada chamada recursiva do *smart constructor*

a tradução se faz palavra a palavra

```
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)
  | ...
```

e é um morfismo para as outras construções (`Eaccess_local`, `Eaccess_global`, `Ecall`, etc.)

a seleção de instrução introduz igualmente operações explícita de acesso à memória

um endereço memória é dada por uma expressão e um *offset*
(para tirar proveito do endereçamento indirecto)

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

no nosso caso, são os acessos aos campos das estruturas que são transformadas em acesso memória

adotamos um esquema simples onde cada campo ocupa exatamente uma palavra (representamos assim o tipo `int` sobre 64 bits)

daí

```
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)
```

com aqui

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

para o resto, nada a assinalar (a seleção de instrução é um morfismo no que diz respeito as instruções do C)

aproveitamos no entanto para esquecer os tipos e agrupar o conjunto das variáveis locais ao nível da função

```
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;
}
```

a segunda fase é a transformação para a linguagem **RTL** (*Register Transfer Language*)

objectivo :

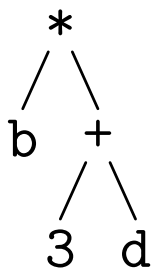
- destruir a estrutura arborescente das expressões e das instruções em favorecimento de um **grafo de fluxo de controlo**, que facilitará as fases posteriores ; removemos em particular a distinção entre expressões e instruções

- introduzir **pseudo-registos** para representar os cálculos intermédios ; estes pseudo-registos estão em números ilimitados e serão, mais tarde, transformados em registos X86-64 ou em localizações na pilha

consideremos a expressão C

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

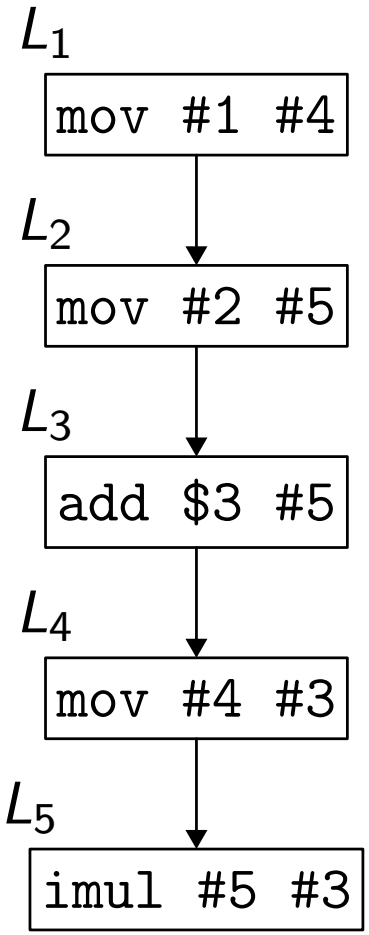
isto é a árvore



supomos que b e d estão nos pseudo-registos #1 e #2

e o resultado está em #3

então construímos o grafo



damo-nos um módulo `Register` para os pseudo-registos

```
type t

val fresh: unit -> t

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

damo-nos igualmente um módulo `Label` para as etiquetas representando os vértices do grafo de controlo

```
type t

val fresh: unit -> t

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

um grafo é assim simplesmente um dicionário associando uma instrução RTL a cada etiqueta

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

de forma inversa, cada instrução RTL indica qual é (ou quais são) a(s) etiqueta(s) seguinte(s) ; assim

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

a instrução `Econst (`*n*`, `*r*`, `*l*`)` significa « carregar o valor *n* no pseudo-registo *r* e transferir o controlo para a etiqueta *l* »

da mesma forma, encontramos o acesso às variáveis globais (ainda representadas pelos identificadores), os acessos à memória, `malloc` e `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
```

finalmente, as operações aritméticas manipulam agora os pseudo-registos

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

para construir o grafo de fluxo de controlo arquivamo-lo (temporariamente) numa referência

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

e damo-nos uma função para juntar uma aresta no grafo

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

traduzimos as expressões graças a uma função

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

que toma em argumento

- o registo destinatário do valor da expressão
- a expressão por traduzir
- a etiqueta de saída *i.e.* correspondendo à continuação do cálculo

e retorna a etiqueta de entrada do cálculo desta expressão

construímos assim o grafo partindo do fim de cada função

a tradução é relativamente fácil

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

quando for necessário, introduzimos pseudo-registos *frescos*

```
  | Istree.Embinop (op, e1, e2) ->
      let tmp1 = Register.fresh () in
      let tmp2 = Register.fresh () in
      expr tmp1 e1 (
      expr tmp2 e2 (
      move tmp1 destr (generate (
      Embinop (op, tmp2, destr, destl)))))
```

para as variáveis locais, damo-nos uma tabela indicando qual pseudo-registo está associado a cada variável

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

onde `Mmov` é a operação mov de X86-64

etc.

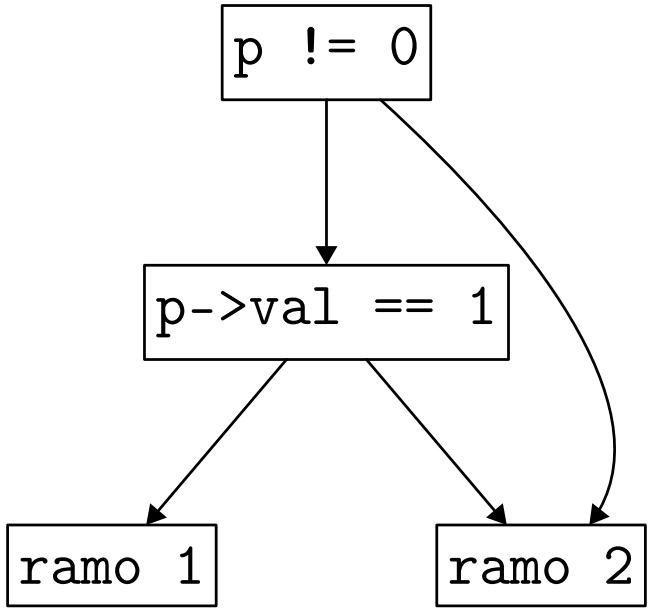
para traduzir os operadores `&&` e `||` e as instruções `if` e `while` são necessários instruções RTL de **salto** (*branching*)

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

com

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

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

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



(os quarto blocos são esquemáticos ; estes se descompõem em realidade em sub-grafos)

para traduzir uma condição, definimos uma função

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

as duas etiquetas passadas em argumento correspondem à sequência dos cálculos nos casos onde a condição é respectivamente verdade e falsa

retornamos a etiqueta de entrada da avaliação da condição

```
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)))
```

(podemos, claro, tratar mais casos particulares do que os que aqui estão)

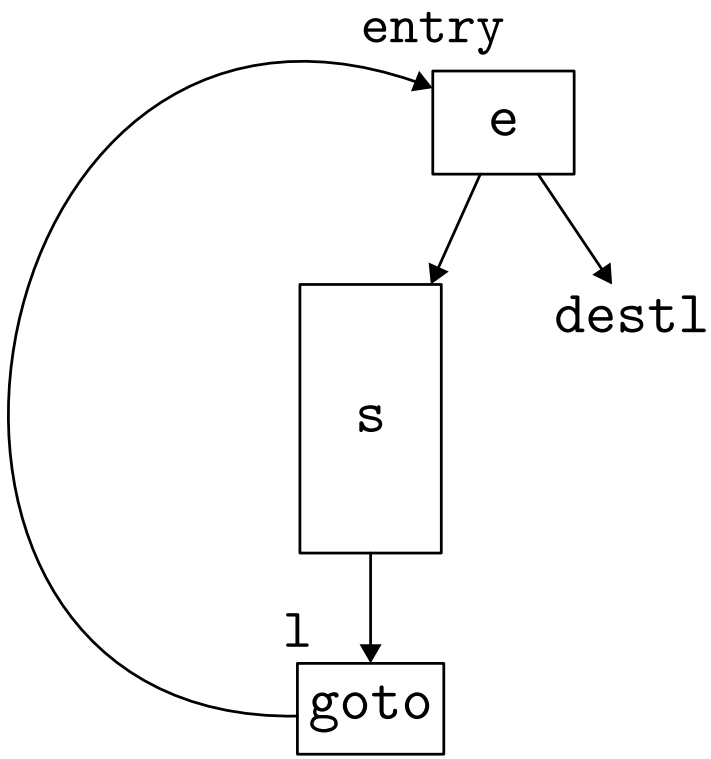
para traduzir `return`, damo-nos um pseudo-registo `retr` para receber o resultado da função e de uma etiqueta `exitl` correspondendo à saída da função ; senão damo-nos uma etiqueta `destl` que corresponde à continuação do cálculo

```
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)
```

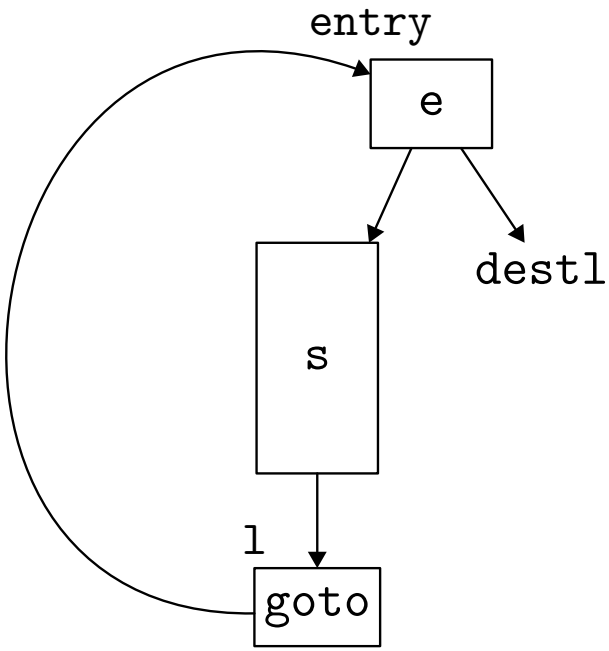
etc.

para um ciclo `while`, criamos um ciclo no grafo de fluxo de controlo

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



```
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
```

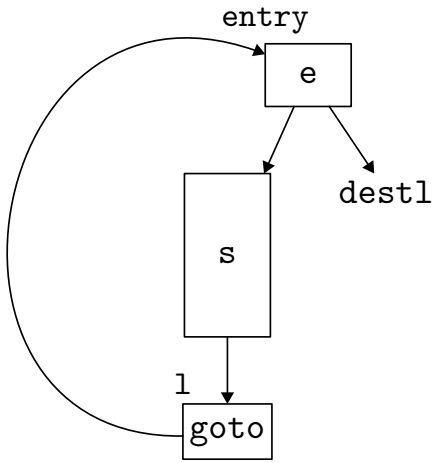


podemos assim escrevê-lo como um operador de ponto fixo

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

por exemplo

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



e utilizá-lo da seguinte forma

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

os parâmetros de uma função e o seu resultado estão agora nos pseudo-registos

```
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;
}
```

idem para a chamada

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

a tradução duma função descompõe-se nas etapas seguintes

1. alocamos pseudo-registos *frescos* para os seus argumentos, o seu resultado, e as suas variáveis locais
2. partimos de um grafo vazio
3. criamos uma etiqueta fresca para a *saída* da função
4. traduzimos o corpo da função no RTL e o resultado é a etiqueta de *entrada* da função

consideremos a inevitável função factorial

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

obtemos

```
#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
```

a terceira fase é a transformação de RTL para a linguagem **ERTL** (*Explicit Register Transfer Language*)

objetivo : explicitar as **convenções de chamadas**, ou seja aqui

- os seis primeiros argumentos são passados via `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` e os seguintes na pilha
- o resultado é retornado via `%rax`
- os registos *callee-saved* devem ser salvaguardados pelo *callee* (`%rbx`, `%r12`, `%r13`, `%r14`, `%r15`, `%rbp`)
- a divisão `idivq` impõe o dividende e o quociente em `%rax`
- `malloc` e `printf` são funções de biblioteca, com argumento em `%rdi` e resultado em `%rax`

supomos que o módulo `Register` descreve igualmente os registos físicos

```
type t
...
val parameters: t list  (* para os n primeiros argumentos *)
val rax: t              (* para o resultado da divisão *)
val callee_saved: t list
val rdi: t              (* para malloc e printf *)
```

em RTL, tínhamos

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

em ERTL, temos agora

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

i.e. só resta o nome da função por chamar, porque novas instruções vão ser inseridas para carregar os argumentos nos registos e na pilha, e para recuperar o resultado em `%rax`
(conservamos no entanto o número de parâmetros passados nos registos , que será utilizado na fase 4)

as instruções `Emalloc` e `Eprintf` desaparecem

as outras instruções de RTL permaneçam inalteradas

mas no entanto, novas instruções aparecem :

- para alocar e desalocar a tabela de activação

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

(nota : não conhecemos ainda o seu tamanho)

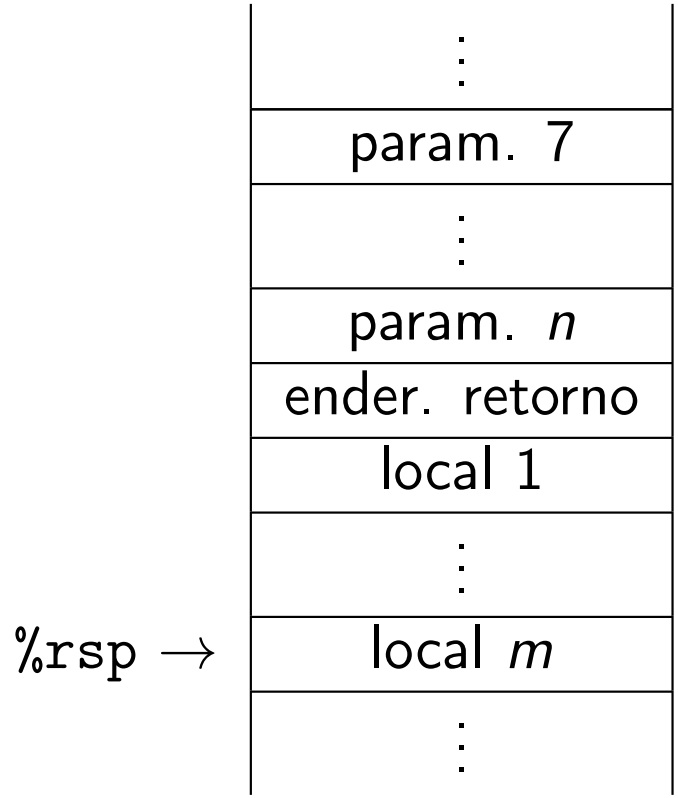
- para ler / escrever os parâmetros na pilha

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

- o retorno é agora explícito

```
| Ereturn
```

a tabela de activação organiza-se da seguinte forma :



a zona das m variáveis locais conterá todos os pseudo-registos que não poderão serem arquivados nos registos físicos ; é a fase de alocação de registos (fase 4) que determinará m

não mudamos a estrutura do grafo de controlo ; limitarmo-nos em inserir **novas instruções**

- no início de cada função, para
  - alocar a tabela de activação
  - salvaguardar os registos *callee-saved*
  - copiar os argumentos nos pseudo-registos correspondentes
- no fim de cada função, para
  - copiar o pseudo-registos contendo o resultado em `%rax`
  - restaurar os registos *callee-saved*
  - desalocar a tabela de activação
- a cada chamada, para
  - copiar pseudo-registos contendo os argumentos em `%rdi`, `...` e na pilha antes da chamada
  - copiar `%rax` para o pseudo-registo contendo o resultado após a chamada

traduzimos as instruções de RTL para ERTL com uma função

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

poucas alterações, com a exceção das chamada, isto é s instruções `Ecall`, `Emalloc`, `Eprintf` e a divisão

```
| 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 e quociente estão em `%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)))))
```

(cuidado com a ordem : dividimos aqui `r2` por `r1`)

em RTL, a chamada apresenta-se na forma

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

onde `r` é o pseudo-registo que recebe o resultado, `x` o nome da função e `rl` a lista dos pseudo-registos contendo os argumentos

começamos por associar os primeiros parâmetros com os registos físicos 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)
```

damo-nos

```
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))
```

a chamada traduz-se então da seguinte forma (é necessário ler o código « ao contrário »)

```
| 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_right (fun t l -> push_param t l) fsl l in
    let l = List.fold_right (fun (t, r) l -> move t r l) frl l in
    Egoto l
```

o código RTL

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

é traduzido em ERTL como

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

resta-nos traduzir cada função

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;
}
```

associamos um pseudo-registo a cada registo físico que deve ser salvaguardado *i.e.* os registos da lista `callee_saved`

```
let deffun f =
  graph := ...traduzimos cada instrução...
  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 entrada da função, é necessário

- alocar a tabela de activação com `Ealloc_frame`
- salvaguardar os registos (lista `savers`)
- copiar os argumentos nos pseudo-registos (`formals`)

```
let fun_entry savers formals entry =
  let frl, fsl = assoc_formals formals in
  let ofs = ref 0 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)
```

(nota : o *offset* de `get_param` é calculado relativamente ao topo da tabela de activação por enquanto, porque não conhecemos ainda o seu tamanho)

à saída da função, é necessário

- copiar o pseudo-registo contendo o resultado para `%rax`
- restaurar os registos salvaguardados
- desalocar a tabela de activação

```
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
```

```
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
```

(supomos aqui que os registos %rbx e %r12 são os únicos *callee-saved*)

ainda é longe de ser o que imaginávamos ser um bom código X86-64 para a factorial

neste ponto é preciso perceber que

- a alocação de registo (fase 4) terá por tarefa associar registos físicos aos pseudo-registos por forma a limitar o uso da pilha, mas também de remover certas instruções

  de facto, se associamos `#8` a `%r12`, removemos simplesmente as duas instruções `L16` e `L21`

- o código ainda não está organizado linearmente (o grafo está organizado e mostrado de forma arbitrária) ; será tarefa para a fase 5, que tratará de minimizar os saltos

é ao nível da tradução de RTL $\rightarrow$ ERTL que se deve realizar a optimização das **chamadas terminais** (se assim for objectivo)

de facto, as instruções por produzir não são as mesmas, e esta mudança terá influência na fase seguinte (de alocação de registos)

há uma dificuldade, no entanto, se a função chamada por uma chamada terminal não tem o mesmo número de argumento passados na pilha ou de variáveis locais, porque a tabela de activação deve ser alterada

duas soluções pelo menos

- limitar a optimização das chamadas terminais aos casos em que a tabela de activação não fica modificada : é o caso se se trata de uma chamada terminal de uma função recursiva de ela própria
- o *caller* altera a tabela de activação e transfere o controlo ao *callee* *depois* da instrução de criação da sua tabela de activação

a quarta fase é a tradução de ERTL para **LTL** (*Location Transfer Language*)

trata-se de fazer desaparecer os pseudo-registos em benefício de

- registos físicos, preferencialmente
- espaço na pilhas, senão

é o que chamamos a fase da **alocação de registos**

a alocação de registos é uma fase complexa, que iremos descompor ela própria em várias fases

1. análise de **duração de vida**
   - trata-se de determinar em quais momentos precisos o valor de um pseudo-registo é necessário para a continuação do cálculo

2. construção de um **grafo de interferência**
   - trata-se de um grafo que indica quais são os pseudo-registos que não podem ficar fisicamente no mesmo local (codificado pelo mesmo registos físico)

3. alocação de registo realizada graças a uma **coloração de grafo**
   - é a atribuição propriamente dos registos físicos e dos locais na pilha aos pseudo-registos

nesta sequência chamamos de *variável* um pseudo-registo ou um registo físico

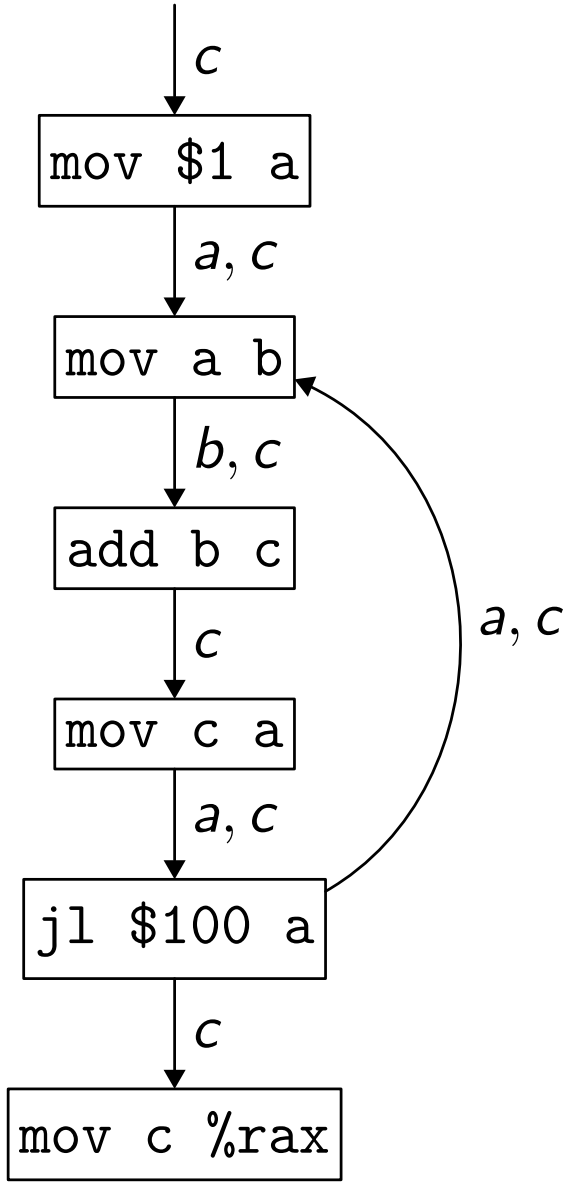
Definição (variável viva)

Num ponto de programa, uma variável é dita ***viva*** *se o valor que ela contém pode ser utilizada na sequência da execução*

dizemos aqui « pode ser utilizada » porque a propriedade « está a ser utilizada » não é decidível ; limitamo-nos a uma aproximação correcta

associemos as variáveis vivas às *arestas* do grafo de fluxo de controlo

```
    mov $1 a
L1: mov a b
    add b c
    mov c a
    jl  $100 a L1
    mov c %rax
```



a noção de variáveis vivas deduz-se das **definições** e das **utilizações** das variáveis realizadas por cada instrução

Definição

Para uma instrução l do grafo de fluxo de controlo, notemos

- *$def(l)$ o conjunto das variáveis definidas por esta instrução e*
- *$use(l)$ o conjunto da variáveis utilizadas por esta instrução.*

exemplo : para a instrução add r_1 r_2 temos

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

para calcular as variáveis vivas, é cómodo associá-las não as arestas, mas sim aos *nodos* do grafo de fluxo de controlo, isto é, a cada instrução

mas é necessário então distinguir as variáveis **vivas à entrada** de uma instrução e as variáveis **vivas à saída**

Definição

Para uma instrução l do grafo de fluxo de controlo, notemos

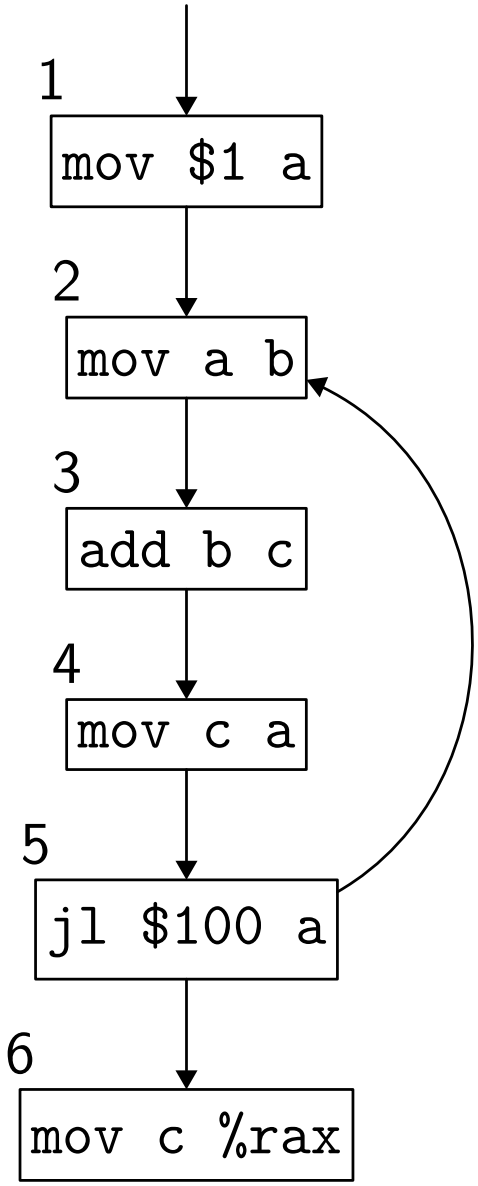
- *$in(l)$ o conjunto das variáveis vivas no conjunto das arestas que chegam a l, e*
- *$out(l)$ o conjunto das variáveis vivas sobre o conjunto das arestas que saem de l.*

as equações que definem $in(l)$ e $out(l)$ são as seguintes

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

tratam-se de equações recursivas cuja menor solução é a solução que nos interessa

estamos perante uma função monótona sobre um domínio finito e podemos assim aplicar o teorema de Tarski (visto nas aulas de CF, mas também nas aulas sobre a *framework* monótona de análise estática que se seguem)



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

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

obtemos o ponto fixo após 7 iterações

supondo que o grafo de fluxo de controlo contendo N nodos e N variáveis, um calculo ingénuo terá uma complexidade de $O(N^4)$ no pior caso

podemos melhorar a eficácia do cálculo de várias formas

- calculando na « ordem inversa » do grafo de fluxo de controlo, e calculando *out* antes de *in* (no exemplo anterior, a convergência é atingida em 3 iterações no lugar de 7)
- fusionando os nodos que só tem um único predecessor e um único sucessor com estes últimos (*basic blocks*)
- utilizando um algoritmo mais súbtil que só calcula novamente os valores de *in* e *out* que podem ter mudado ; é o algoritmo de Kildall

ideia : se $in(l)$ muda, então é necessário fazer novamente o cálculo para os predecessores de l unicamente

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

donde o algoritmo seguinte

```
seja  WS um conjunto contendo todos os nodos
enquanto WS não for vazio
    extrair um nodo l de WS
    old_in <- in(l)
    out(l) <- ...
    in(l)  <- ...
    se in(l) difere de old_in(l) então
        juntar todos os predecessores de l em WS
```

o cálculo dos conjuntos *def*(*l*) (definições) e *use*(*l*) (utilizações) é imediata para a maior parte das instruções

exemplos

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

a situação para as chamadas é um pouco mais súbtil

para uma chamada, expressamos que os $min(6, n)$ primeiros registos da lista `parameters` vão ser utilizados, e que todos os registos *caller-saved* podem ser sobrepostos pela chamada

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

finalmente, para `return`, `%rax` e todos os registos *callee-saved* serão utilizados

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

reconsideremos o exemplo da factorial e a sua forma ERTL

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

exemplo da factorial

```
L17: alloc_frame -> L16  in = %r12,%rbx,%rdi  out = %r12,%rbx,%rdi
L16: mov %rbx #7 -> L15  in = %r12,%rbx,%rdi  out = %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
```

conclusão

- prática laboratorial desta semana:
  - coloração de grafos
- próxima aula
  - segunda parte do capítulo sobre a produção de código eficaz

Para além dos agradecimentos habituais ao Jean-Christophe Filliâtre, por "transitividade" agradece-se também ao François Pottier e ao Xavier Leroy que influenciaram tanto a arquitectura do compilador aqui descrito (baseado no CompCert do X. Leroy) como também as notas apresentadas (inspiradas também das aulas de F. Pottier)

estes acetatos resultam essencialmente de uma adaptação do material pedagógico gentilmente cedido pelo Jean-Christophe Filliâtre (link1, link2)

adicionalmente poderá consultar as obras seguintes

- **Modern Compilers: Principles, Techniques, and Tools**, Alfred V. Aho, Monica S. Lam, Ravi Sethi et Jeffrey D. Ullman
- **Types and Programming Languages**, Benjamin C. Pierce
- **Modern Compiler Implementation**, Andrew W. Appel (3 versões: ML, C, Java)