Universidade da Beira Interior

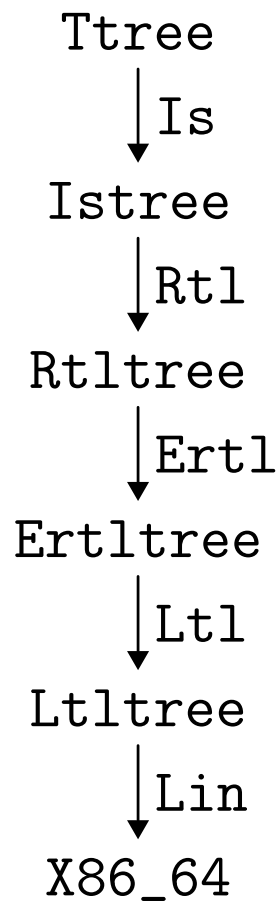
Desenho de Linguagens de Programação e de Compiladores

Simão Melo de Sousa

Aula 12 - Produção de código eficiente, parte 2

a produção de código eficaz foi estruturada em várias fases :

1. seleção de instruções
2. RTL (*Register Transfer Language*)
3. ERTL (*Explicit Register Transfer Language*)
4. LTL (*Location Transfer Language*)
   - 4.1 análise de duração de vida
   - 4.2 construção de um grafo de interferência
   - 4.3 alocação de registos por coloração de grafo
5. código linearizado (*assembly*)



tomemos como exemplo um fragmento simples da linguagem C

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

fase 1 : la seleção de instruções

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

fase 2 : RTL (*Register Transfer Language*)

```
#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
```

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

```
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
```

fase 4 : LTL (*Location Transfer Language*)

já realizamos a fase da **análise de duração de vida** *i.e.* já determinamos para cada variável (pseudo-registos ou registo físicos) em que momento o valor que elas contém pode ser utilizada no que se segue da execução

análise de duração de vida

```
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
```

vamos agora redescobrir a construção um **grafo de interferência** que exprime as restrições sobre os locais possíveis dos pseudo-registos

Definição (interferência)

*Dizemos que duas variáveis v_1 e v_2 **interferem** se não podem ser concretizadas pelo mesmo local (registo físico ou local memória)*

como a interferência não é decidível, limitamo-nos às condições suficientes

consideremos uma instrução que define uma variável v : qualquer outra variável w viva à saída desta instrução pode interferir com v

no entanto, no caso particular de uma instrução

`mov` w v

não desejamos declarar que v e w interfiram porque pode ser interessante concretizar v e w no mesmo local e assim eliminar uma ou mais instruções

adoptamos assim a definição seguinte

Definição (grafo de interferência)

*O **grafo de interferência** de uma função é um grafo não orientado cujos vértices são variáveis desta função e cujas arestas são de dois tipos : de interferência ou de preferência.*
Para cada instrução que define uma variável v e cujas variáveis vivas na saída, fora v, são $w_1, \ldots, w_n$, procedemos da seguinte forma :

- *se a instrução não é uma instrução* `mov` w v*, juntamos as n arestas de interferências $v - w_i$*
- *se se trata de uma instrução* `mov` w v*, juntamos as arestas de interferência $v - w_i$ para todos os w_i diferentes de w e juntamos a aresta de preferência $v - w$.*

(se uma arestas $v - w$ é simultaneamente de preferência e de interferência, conservamos a aresta de interferência)

o grafo de interferência pode assim ser representado em OCaml :

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

type graph = edges Register.map
```

a função que o constrói é da forma

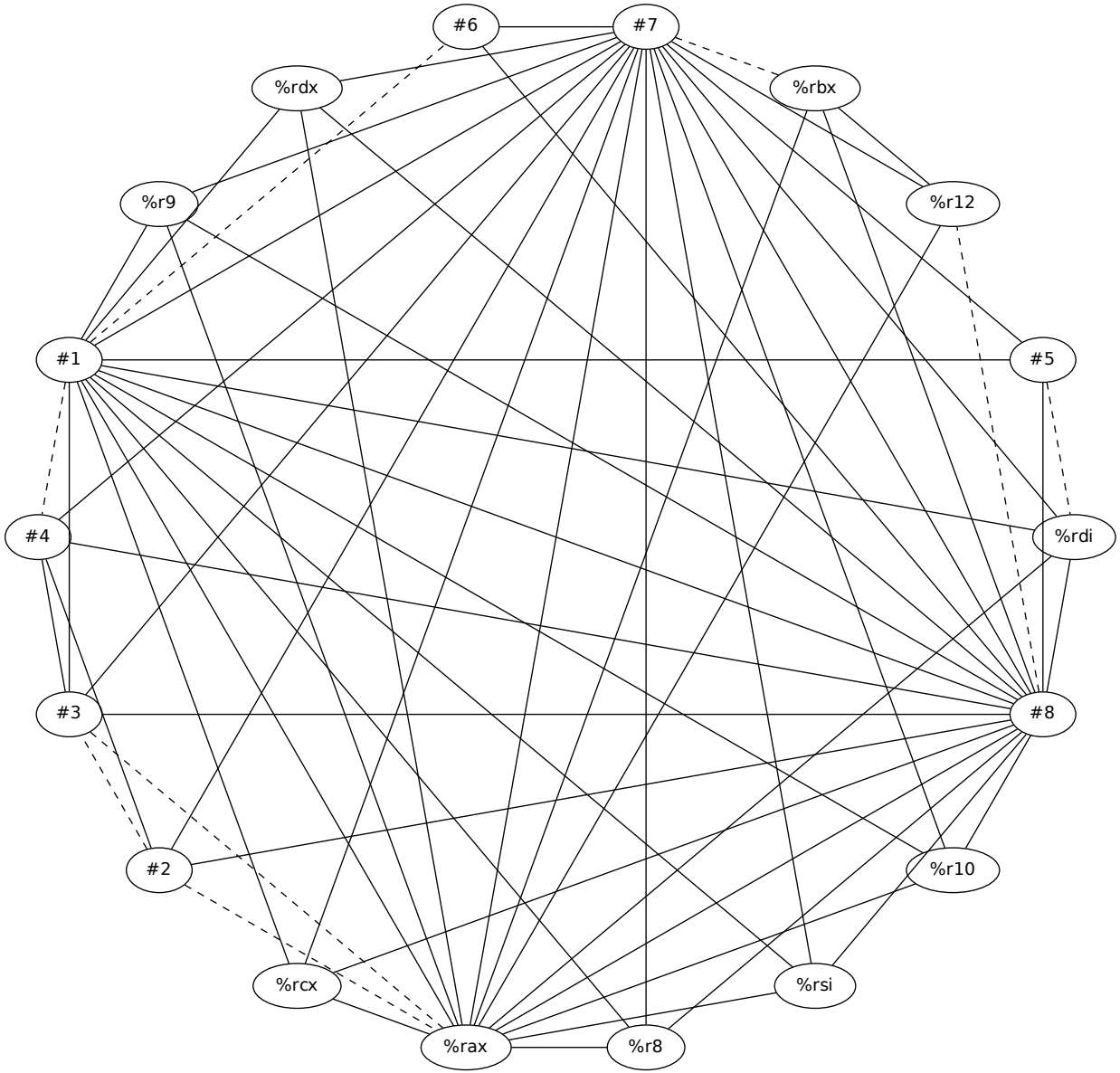
```
val make: Liveness.info Label.map -> graph
```

exemplo : factorial

obtemos o seguinte para a função `fact`

10 registos físicos + 8 pseudo-registos

arestas de preferência são pontilhadas



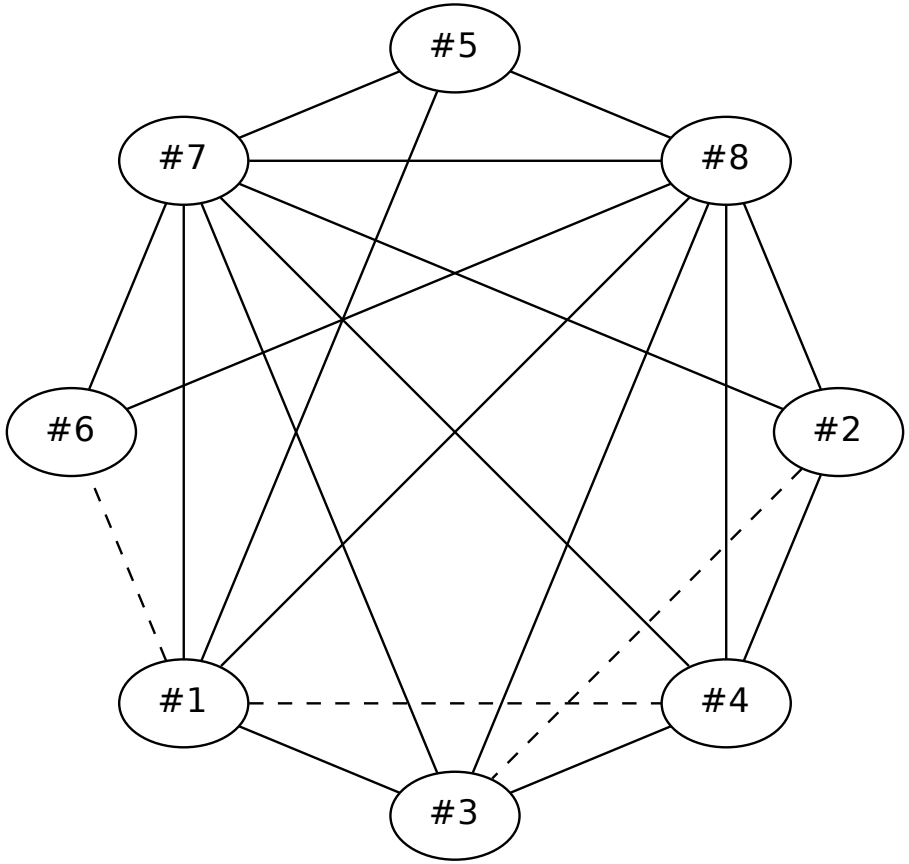
podemos então ver o problema da alocação de registos como um problema de **coloração de grafo** :

- as cores são os registos físicos
- dois vértices ligados por uma aresta de interferência não podem receber a mesma cor
- dois vértices ligados por uma aresta de preferência devem receber a mesma cor, desde que possível.

nota : há dentro do grafo vértices que são registos físicos, ou seja, vértices já com cor atribuida

observemos as cores possíveis para os pseudo-registos

	cores possíveis
#1	%r12, %rbx
#2	todas
#3	todas
#4	todas
#5	todas
#6	todas
#7	%rbx
#8	%r12



neste exemplo, vemos imediatamente que a coloração é impossível

- somente duas cores para colorir #1, #7 e #8
- estes interferem entre si

se um vértice não pode ser colorido, este corresponderá a um posicionamento na pilha ; dizer-se-á que fazemos um *spill* do pseudo-registo

mesmo no caso de uma coloração possível, determinar a coloração optima é muito custoso : é um problema NP-Completo

deveremos colorir utilizando **heurísticas**, tendo como objectivo

- uma complexidade linear ou quasi-linear
- uma boa exploração das arestas de preferência

um dos melhores algoritmos é da autoria de George e Appel (*Iterated Register Coalescing*, 1996) ; (ver livro de Appel, capítulo 11)

este algoritmo explora as ideias seguintes

seja K o número de cores (*i.e.* o número de registos físicos)

uma primeira ideia, da autoria de Kempe (1879 !), é a seguinte : se um vértice tem um grau $< K$, então podemos retirá-lo do grafo, colorir o resto, e teremos em seguida seguramente uma cor para lhe atribuir ; esta etapa é chamada de **simplificação**

os vértices retirados são assim colocados numa pilha

retirar um vértice diminuí o grau dos outros vértices e pode assim produzir novos candidatos para a simplificação

quando só restam vértices de grau $\geq K$, escolhemos um como **candidato ao spilling** (*potential spill*) ; este é retirado do grafo, colocado na pilha e o processo de simplificação pode retomar

escolhemos de preferência um vértice que

- é pouco utilizado (os acessos à memória custam caro)
- tem um grau alto (para favorecer simplificações futuras)

quando o grafo é vazio, começamos o processo de coloração, chamado de **seleção**

tiramos da pilha os vértices, um a um e para cada um deles

- se se trata de um vértice com grau baixo, estamos com condições de lhe atribuir uma cor
- se se trata de um vértice com grau alto, isto é candidato ao spilling, então
  - ou pode ser coorido, porque os seus vizinhos utilizam menos de K cores ; fala-se de **coloração optimista**
  - ou não pode ser colorido e deve ser efectivamente alvo de *spilling* (fala-se de *actual spill*)

finalmente, convém utilizar da melhor forma as arestas de preferência

para isso, utilizamos uma técnica chamada **coalescência** (*coalescing*) que consiste em fundir dois vértices do grafo

assim podemos aumentar o grau do vértice resultante, juntamos um critério suficiente para não deteriorar a K-colorabilidade

Definição (critério de George)

dois vértices v_1 e v_2 podem fundir
se todo o vizinho de v_1 de grau $\geq K$ é igualmente vizinho de v_2.

pseudo-código do algoritmo de George-Appel

escreve-se naturalmente de forma recursiva

```
let rec simplify g =
  ...
and coalesce g =
  ...
and freeze g =
  ...
and spill g =
  ...
and select g v =
  ...
```

nota : a pilha dos vértices por colorir é assim implícita

```
let rec simplify g =
  if existe um vertice v sem arestas de preferência
     de grau minimal  e < K
  then
    select g v
  else
    coalesce g
```

```
and coalesce g =
  if existe uma aresta de preferência v1-v2
     que satisfaz o critério de George
  then
    g <- fundir g v1 v2
    c <- simplify g
    c[v1] <- c[v2]
    retornar c
  else
    freeze g
```

```
and freeze g =
  if existe um vértice de grau minimal < K
  then
    g <- esquecer as arestas de preferência de v
    simplify g
  else
    spill g
```

```
and spill g =
  if g está vazio
  then
    retornar a coloração vazia
  else
    escolher um vértice v de custo minimal
    select g v
```

podemos tomar, por exemplo

$$custo(v) = \frac{número\ de\ utilização\ de\ v}{grau\ de\ v}$$

```
and select g v =
  remover o vértice v de g
  c <- simplify g
  if existe uma cor r possível para v
  then
    c[v] <- r
  else
    c[v] <- spill
  retornar c
```

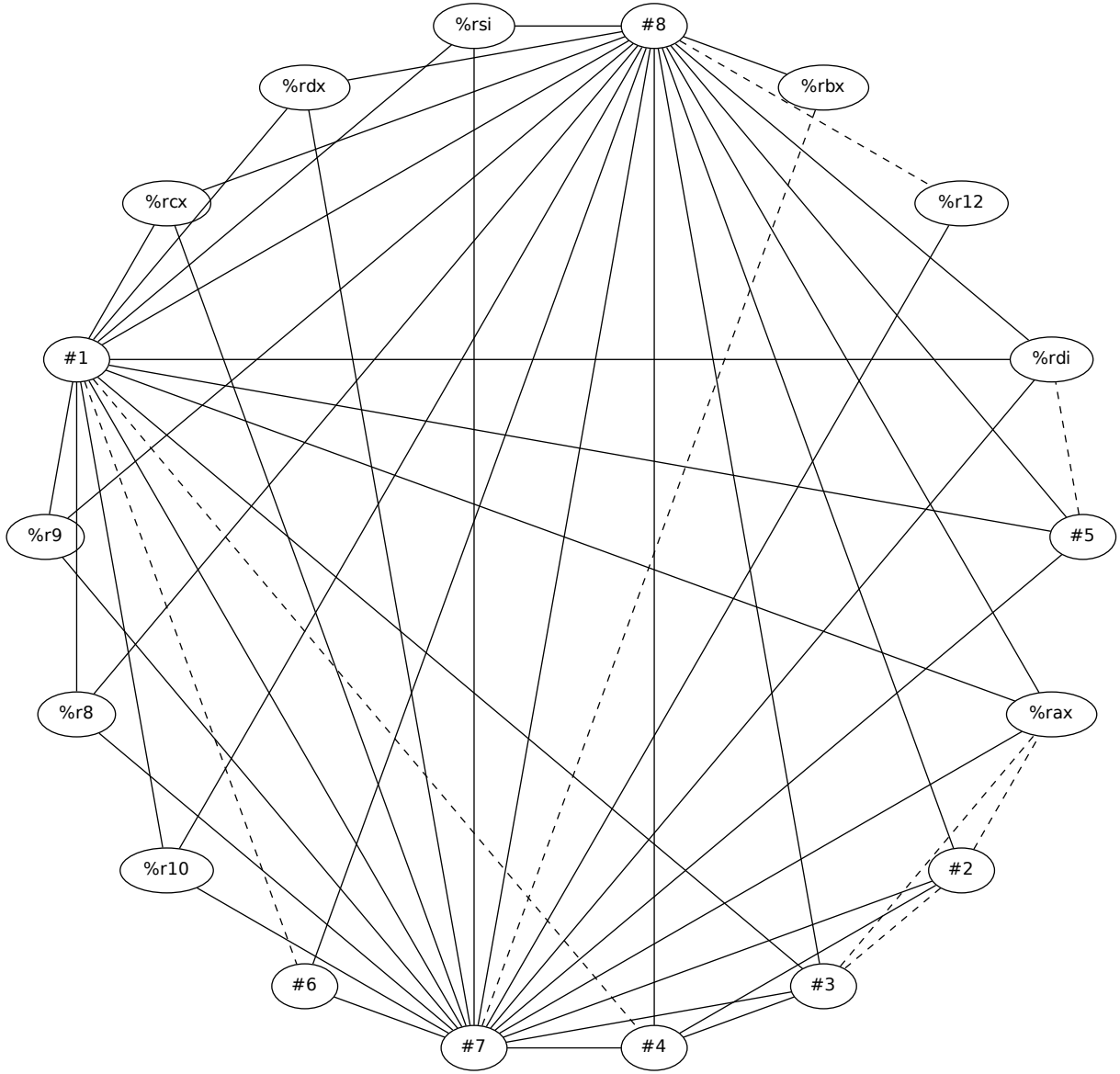
exemplo

1.

`simplify g` $\rightarrow$

`coalesce g` $\rightarrow$

seleciona #2- - -#3



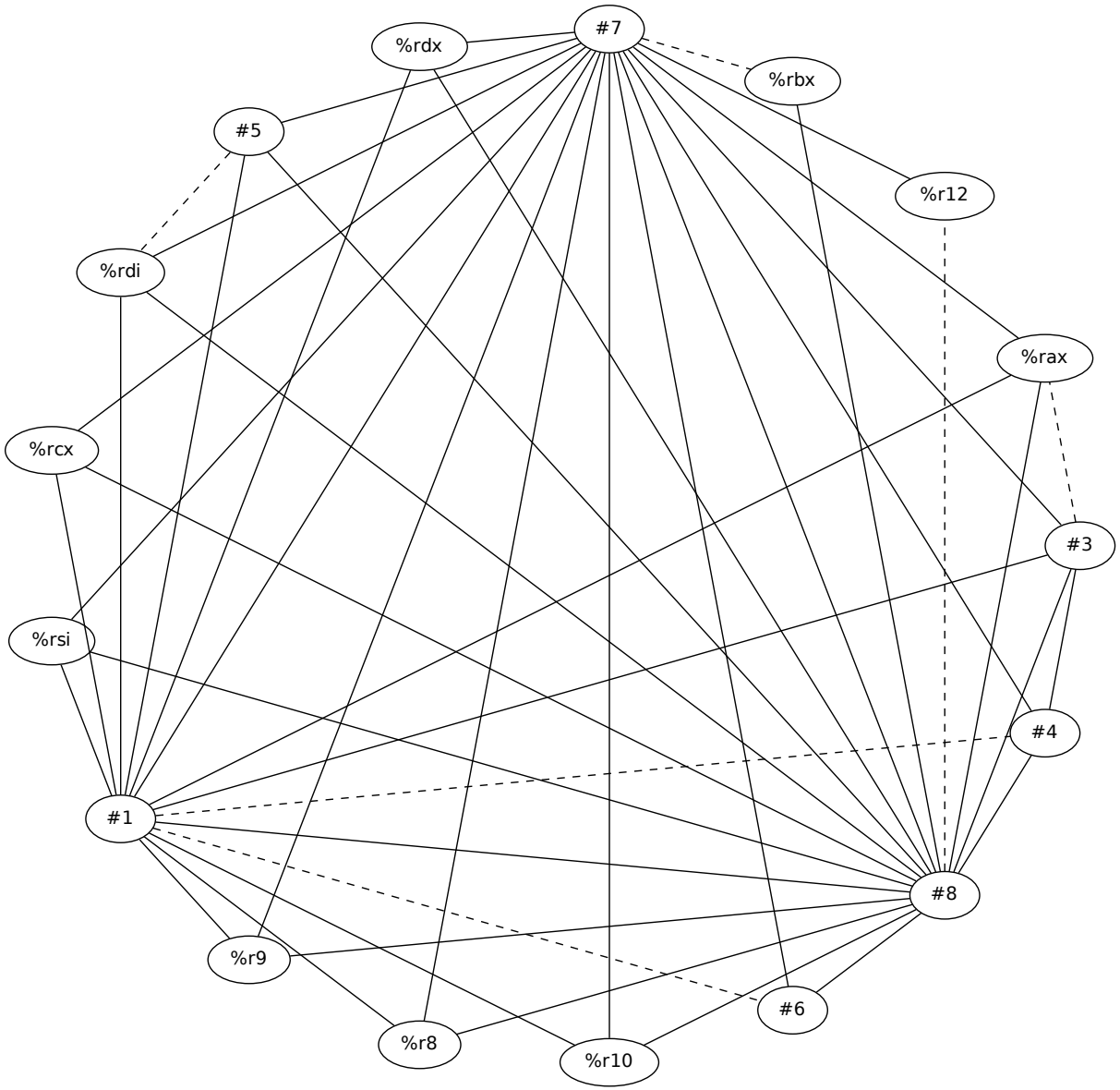
exemplo

2.

`simplify g` $\rightarrow$

`coalesce g` $\rightarrow$

seleciona #4- - -#1

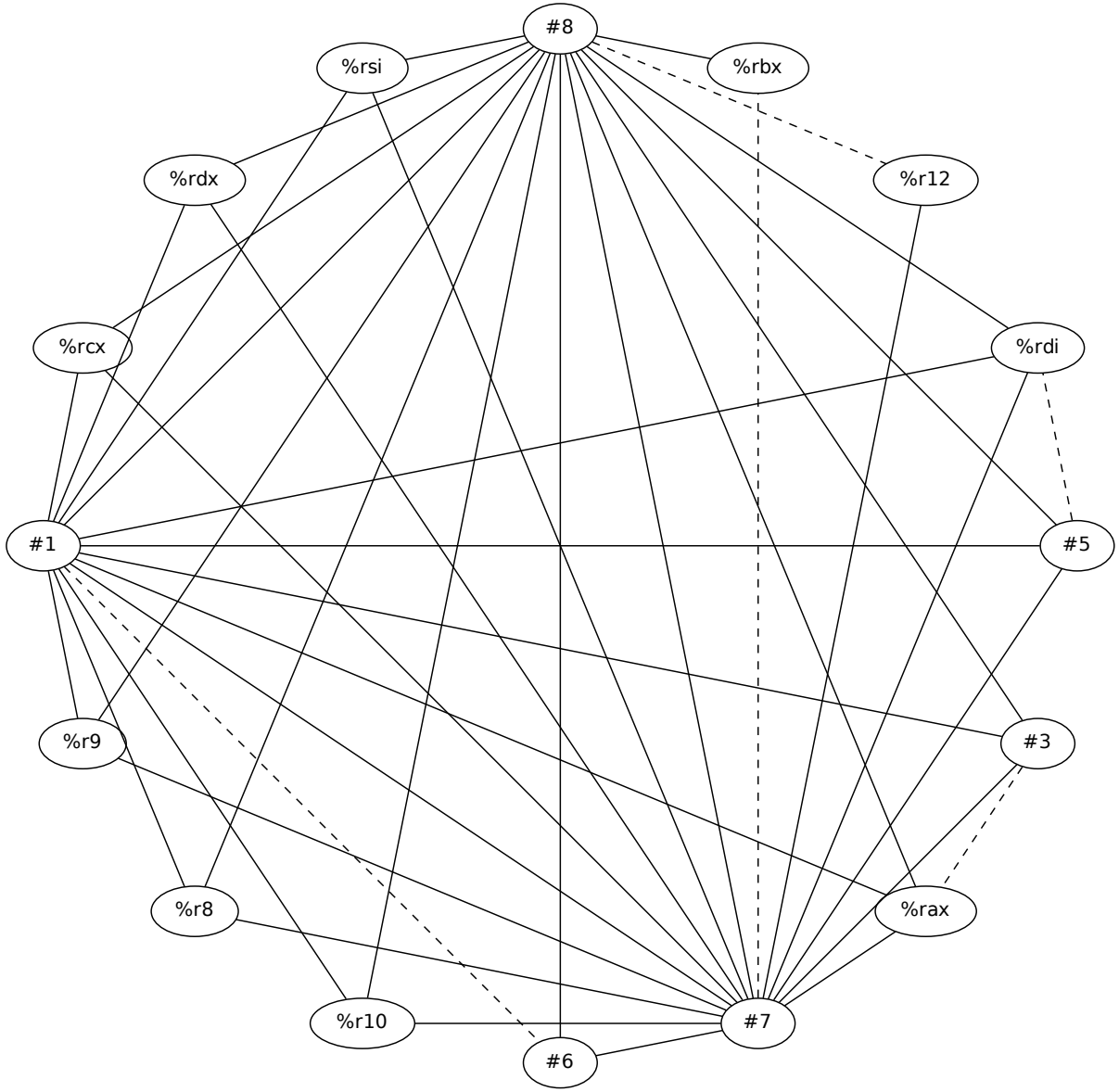


3.

`simplify g` $\rightarrow$

`coalesce g` $\rightarrow$

seleciona #6- - -#1

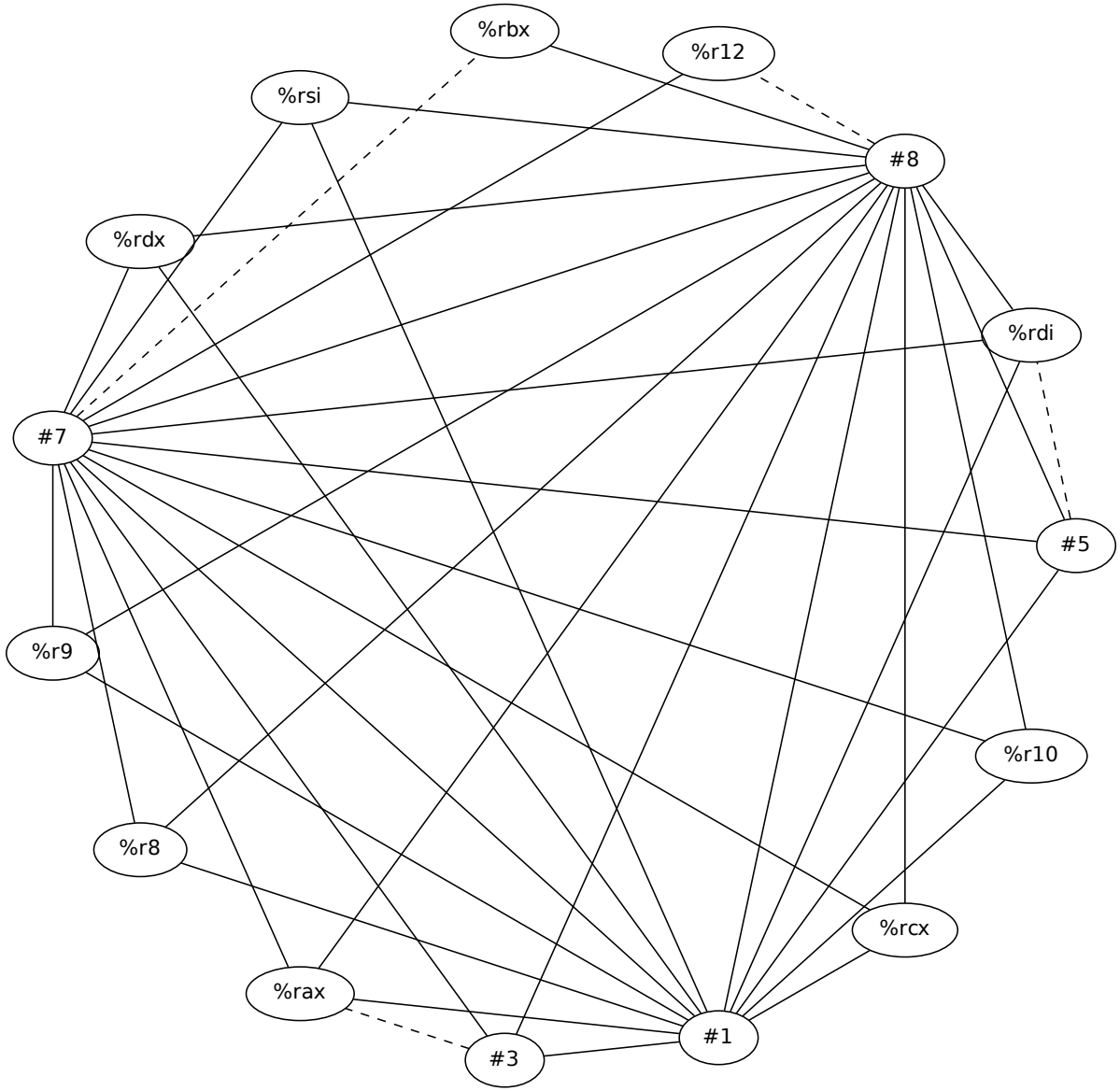


4.

`simplify g` $\rightarrow$

`coalesce g` $\rightarrow$

seleciona #3- - -`%rax`

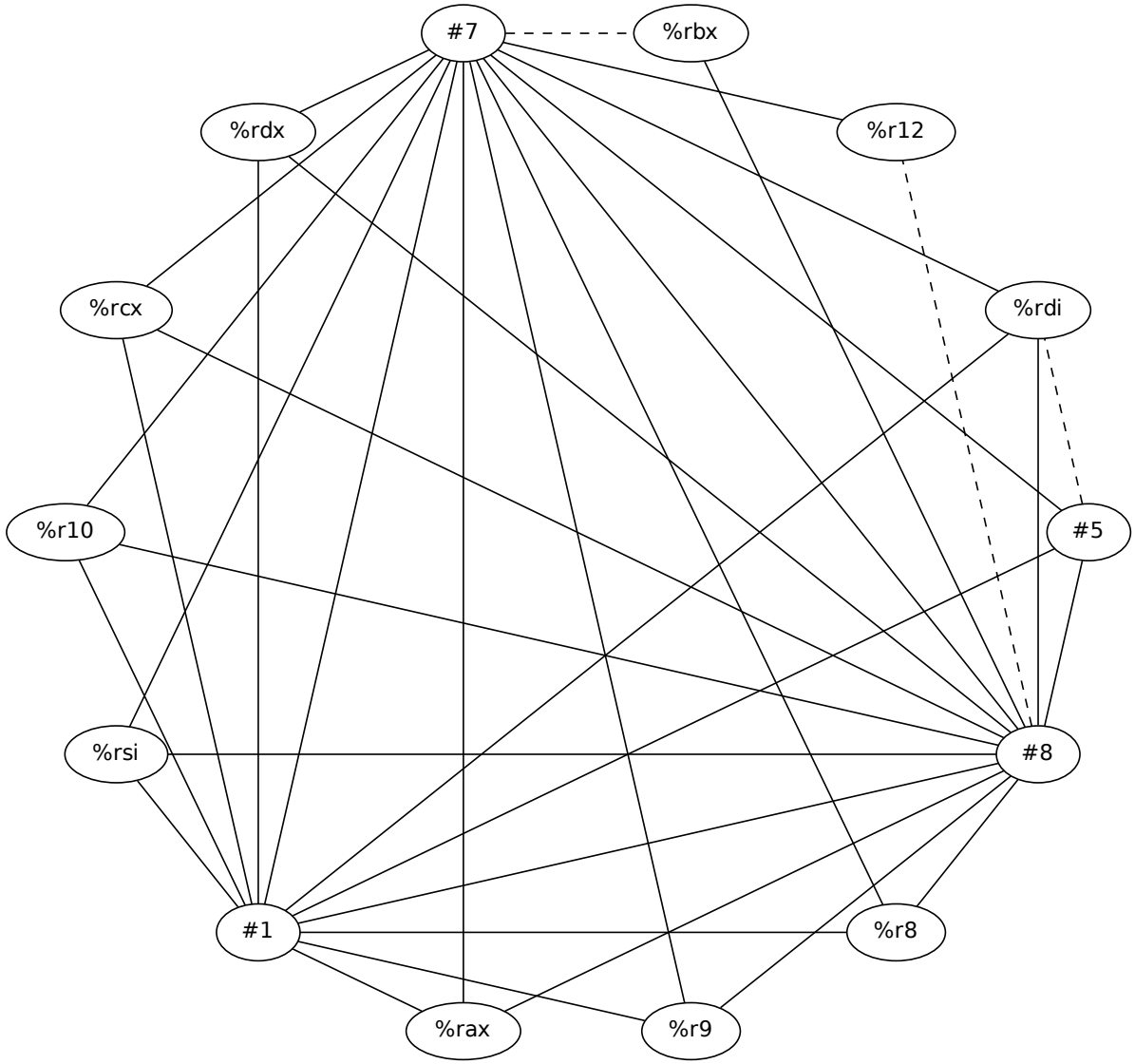


5.

`simplify g` $\rightarrow$

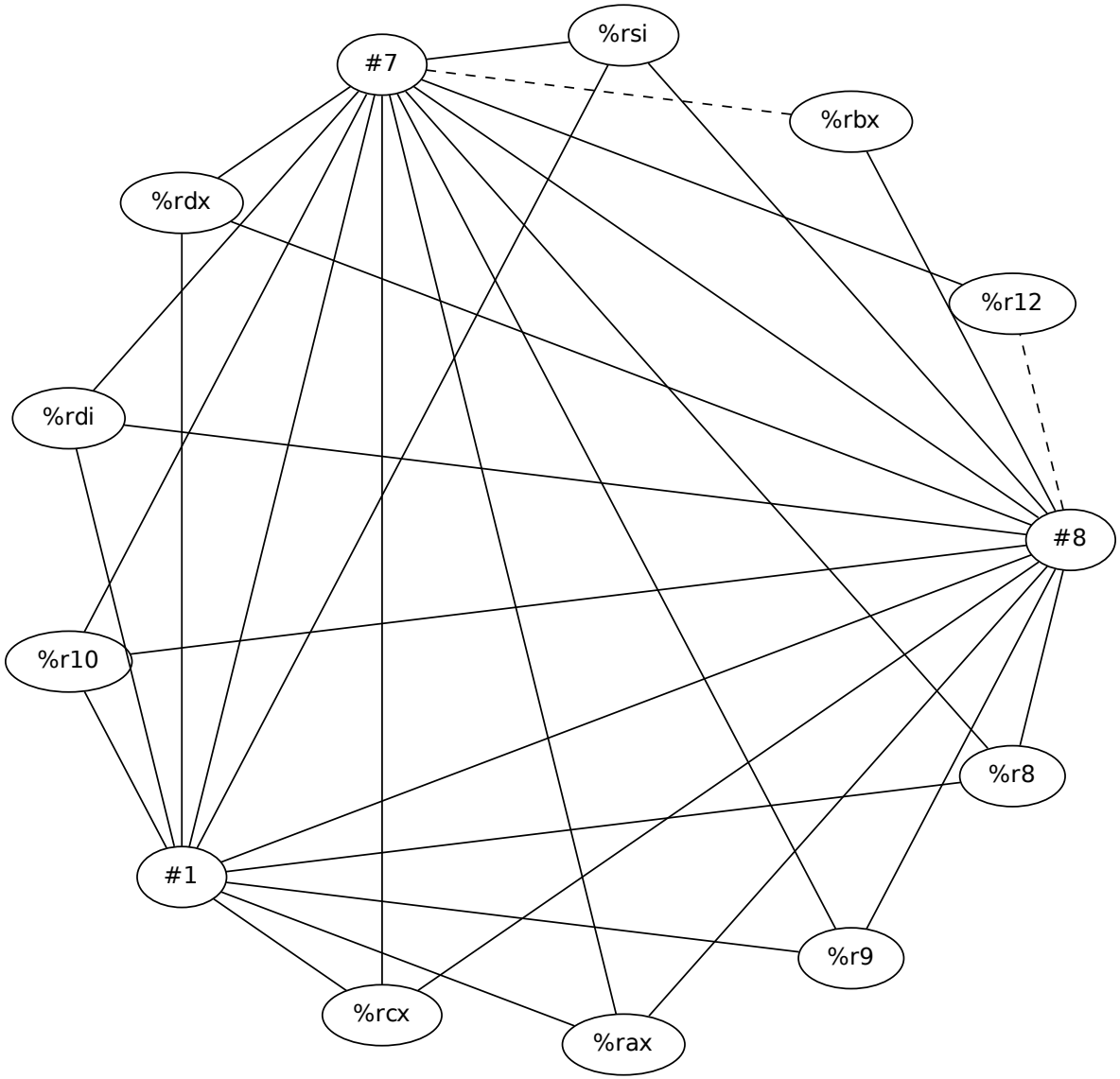
`coalesce g` $\rightarrow$

seleciona #5- - -`%rdi`



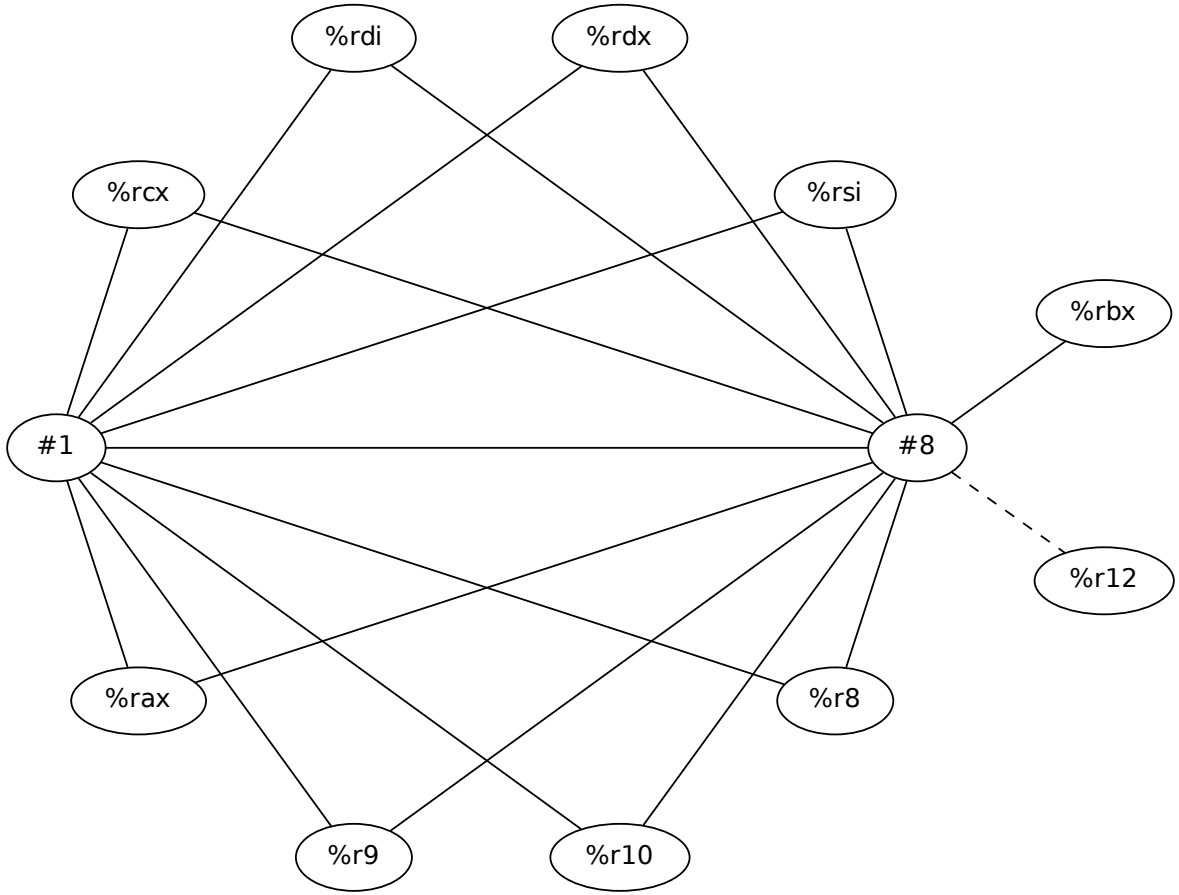
6.

`simplify g` $\rightarrow$
`coalesce g` $\rightarrow$
`freeze g` $\rightarrow$
`spill g` $\rightarrow$
`select g #7`



7.

`simplify g` $\rightarrow$
`select g #1`

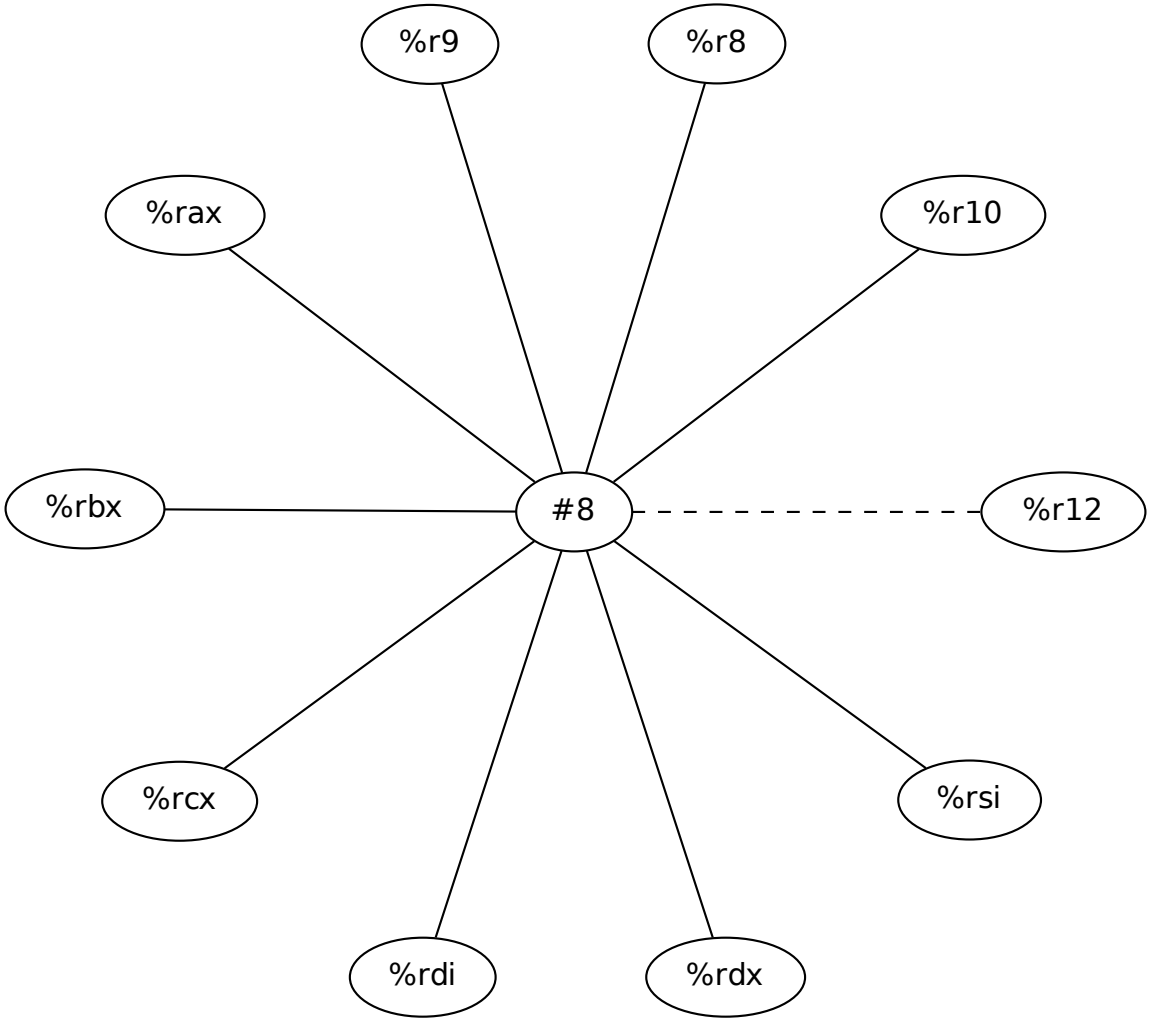


8.

`simplify g` $\rightarrow$

`coalesce g` $\rightarrow$

seleciona #8- - -`%r12`



9.

`simplify g` $\rightarrow$

`coalesce g` $\rightarrow$

`freeze g` $\rightarrow$

`spill g` $\rightarrow$

o grafo está vazio

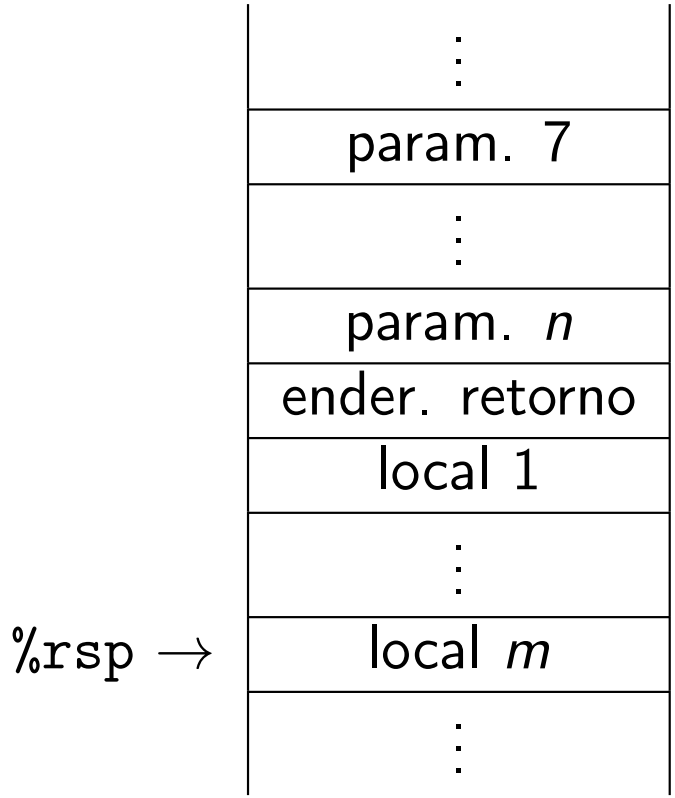
em seguida, tiramos da pilha

8.	`coalesce #8- - -%r12`	$\rightarrow$	$c[\#8] = \texttt{\%r12}$
7.	`select #1`	$\rightarrow$	$c[\#1] = \texttt{\%rbx}$
6.	`select #7`	$\rightarrow$	$c[\#7] = \texttt{spill}$
5.	`coalesce #5- - -%rdi`	$\rightarrow$	$c[\#5] = \texttt{\%rdi}$
4.	`coalesce #3- - -%rax`	$\rightarrow$	$c[\#3] = \texttt{\%rax}$
3.	`coalesce #6- - -#1`	$\rightarrow$	$c[\#6] = c[\#1] = \texttt{\%rbx}$
2.	`coalesce #4- - -#1`	$\rightarrow$	$c[\#4] = c[\#1] = \texttt{\%rbx}$
1.	`coalesce #2- - -#3`	$\rightarrow$	$c[\#2] = c[\#3] = \texttt{\%rax}$

e os pseudo-registos derramados (*spilled*) ?

que fazer com os pseudo-registos derramados ?

associamos as suas localizações na pilha, na zona baixa da tabela de activação, por baixo dos parâmetros



vário pseudo-registos podem ocupar o mesmo local na pilha se estes não se interferem entre si ⇒ como minimizar m ?

é mais uma vez um problema de coloração de grafo, mas desta vez com um número infinito de cor possível, cada cor corresponde a um local na pilha distinto

algoritmo :

1. funde-se todas as arestas de preferência (coalescência), porque `mov` entre dois registos *spilled* custa caro
2. aplicamos em seguida o algoritmo de simplificação, escolhendo de cada vez o vértice de grau mais fraco (heurística)

o resultado da alocação de registo tem a forma seguinte

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

type coloring = color Register.map
```

a função de alocação apresenta-se então da seguinte forma

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

obtém-se a alocação de registo seguinte

```
#1 -> %rbx
#2 -> %rax
#3 -> %rax
#4 -> %rbx
#5 -> %rdi
#6 -> %rbx
#7 -> stack 0
#8 -> %r12
```

o que *daria* o código seguinte

```
fact(1)
  entry : L17

  L17: alloc_frame       -> L16
  L16: mov %rbx 0(%rsp)-> L15
  L15: mov %r12 %r12    -> L14
  L14: mov %rdi %rbx    -> L10
  L10: mov %rbx %rbx    -> L9
  L9 : jle $1 %rbx -> L8, L7
  L8 : mov $1 %rax       -> L1
  L1 : goto              -> L22
  L22: mov %rax %rax    -> L21
  L21: mov 0(%rsp) %rbx-> L20
```

```
L20: mov %r12 %r12   -> L19
L19: delete_frame    -> L18
L18: return
L7 : mov %rbx %rdi   -> L6
L6 : add $-1 %rdi    -> L5
L5 : goto            -> L13
L13: mov %rdi %rdi   -> L12
L12: call fact(1)    -> L11
L11: mov %rax %rax   -> L4
L4 : mov %rbx %rbx   -> L3
L3 : mov %rax %rax   -> L2
L2 : imul %rbx %rax -> L1
```

como constatamos, muitas instruções da forma

$$\texttt{mov}\ v\ v$$

podem ser eliminada ; é precisamente o interesse das arestas de preferências

esta eliminação será realizada durante a tradução para LTL

a maioria das instruções LTL são as mesmas que as do ERTL, com a excepção dos registos que são agora todos registos físicos ou então locais na pilha

```
type instr =
  | Eaccess_global of ident * register * label
  | Eload of register * int * register * label
  | ...
  | Econst of int32 * color * label
  | Emunop of munop * color * label
  | Embinop of mbinop * color * color * label
  | ...
```

acresce que `Ealloc_frame`, `Edelete_frame` e `Eget_param` desaparecem em benefício de uma manipulação explícita de `%rsp`

traduzimos cada instrução ERTL com a ajuda de uma função que toma em argumento a coloração do grafo de uma parte, e a estrutura da tabela de activação de outra parte (que agora é conhecida para cada função)

```
type frame = {
  f_params: int; (* tamanho parâmetro + endereço retorno *)
  f_locals: int; (* tamanho variáveis locais *)
}

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

uma variável `r` pode ser

- (desde já) um registo físico
- um pseudo-registo concretizado num registo físico
- um pseudo-registo concretizado num local na pilha

em certos casos, a tradução é fácil porque a instrução *assembly* em causa permite todas as combinações
exemplo

```
let instr colors frame = function
  | Ertltree.Econst (n, r, l) ->
      let c =
        try Register.M.find r colors with Not_found -> r in
      Econst (n, c, l)
```

nos outros casos, em contrapartida, é mais complicado porque nem todas os operandos são autorizadas
o caso de um acesso a uma variável global, por exemplo

```
  | Eaccess_global (x, r, l) ->
      ?
```

é problemática quando `r` está na pilha porque não podemos escrever

```
  movq x, n(%rsp)
```

```
(too many memory references for ‘movq’)
```

é necessário então usar um registo intermédio
problema : que registo físico utilizar ?

adotamos aqui uma solução simples : dois registos particulares vão ser utilizados como registos temporários para estas transferências com a memória, e não serão usados de outra forma
(concretamente, escolhemos usar `%rbp` e `%r11`)

na prática, não temos necessariamente o contexto de poder desperdiçar assim dois registos ; devemos assim alterar o grafo de interferência e retomar o processo de alocação de registos para poder determinar um registo livre para a transferência

felizmente, este processo volta a convergir muito rapidamente em prática (em 2 ou 3 etapas somente)

damo-nos então dois registos temporários

```
val tmp1: Register.t  (* %rbp *)
val tmp2: Register.t  (* %r11 *)
```

para escrever a variável `r`, damo-nos uma função `write`, que toma igualmente em argumento a coloração e a etiqueta onde é preciso ir após a escrita ; esta retorna o registo físico no qual se deve realmente escrever e a etiqueta onde a execução deve retomar

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

podemos agora traduzir facilmente de ERTL para LTL

```
let instr colors frame = function
  | Ertltree.Eaccess_global (x, r, l) ->
      let hwr, l = write colors r l in
      Eaccess_global (x, hwr, l)
  | ...
```

de forma inversa , damo-nos uma função `read1` para ler o conteúdo de uma variável

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

e utilizamo-la desta forma

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

damo-nos igualmente uma função `read2` para ler o conteúdo de duas variáveis

e utilizamo-la desta forma

```
  | Ertltree.Estore (r1, r2, n, l) ->
      read2 colors r1 r2 (fun hw1 hw2 ->
      Estore (hw1, hw2, n, l))
```

tratamos de forma particular as instruções `Mmov` e `Mmul`

```
  | Ertltree.Embinop (op, r1, r2, l) ->
    begin match op, lookup colors r1, lookup colors r2 with
    | Mmov, o1, o2 when o1 = o2 ->
        Egoto l
    | _, (Spilled _ as o1), (Spilled _ as o2)
    | Mmul, o1, (Spilled _ as o2) ->
        read1 colors r2 (fun hw2 ->
        Embinop (op, o1, Reg hw2, generate (
        Embinop (Mmov, Reg hw2, o2, l))))
    | _, o1, o2 ->
        Embinop (op, o1, o2, l)
    end
```

agora que conhecemos o tamanho da tabela de activação podemos traduzir `Eget_param` em termos de acesso relativos a `%rsp`

```
  | Ertltree.Eget_param (n, r, l) ->
      let hwr, l = write colors r l in
      let n = frame.f_locals + frame.f_params + n in
      Embinop (Mmov, Spilled n, Reg r, l)
```

finalmente, podemos traduzir `Ealloc_frame` e `Edelete_frame` em termos de manipulação de `%rsp`

```
| Ertltree.Ealloc_frame l
| Ertltree.Edelete_frame l when frame.f_locals = 0 ->
    Egoto l
| Ertltree.Ealloc_frame l ->
    Emunop (Maddi (- frame.f_locals), Reg rsp, l)
| Ertltree.Edelete_frame l ->
    Emunop (Maddi frame.f_locals, Reg rsp, l)
```

só temos agora de juntar todas as peças

```
let deffun debug f =
  let ln = Liveness.analyze f.fun_body in
  let ig = Interference.make ln in
  let c, nlocals = Coloring.find ig in
  let n_stack_params =
    max 0 (f.fun_formals - List.length Register.parameters) in
  let frame = { f_params = word_size * (1 + n_stack_params);
                f_locals = word_size * nlocals } in
  graph := Label.M.empty;
  Label.M.iter
    (fun l i ->
       let i = instr c frame i in graph := Label.M.add l i !graph)
    f.fun_body;
  { fun_name  = f.fun_name;
    fun_entry = f.fun_entry;
    fun_body  = !graph; }
```

para a factorial, obtemos o código LTL seguinte

```
fact()
  entry : L17
  L17: add $-8 %rsp          -> L16
  L16: mov %rbx 0(%rsp)      -> L15
  L15: goto                  -> L14
  L14: mov %rdi %rbx         -> L10
  L10: goto                  -> L9
  L9 : jle $1 %rbx       -> L8, L7
  L8 : mov $1 %rax           -> L1
  L1 : goto                  -> L22
  L22: goto                  -> L21
  L21: mov 0(%rsp) %rbx      -> L20
  L20: goto                  -> L19
  L19: add $8 %rsp           -> L18
  L18: return
  L7 : mov %rbx %rdi         -> L6
  L6 : add $-1 %rdi          -> L5
  L5 : goto                  -> L13
  L13: goto                  -> L12
  L12: call fact             -> L11
  L11: goto                  -> L4
  L4 : goto                  -> L3
  L3 : goto                  -> L2
  L2 : imul %rbx %rax        -> L1
```

resta-nos uma fase : o código ainda está na forma de um **grafo de fluxo de controlo** e o objectivo desta fase é a produção de **código *assembly* linear**

mais precisamente : as instruções de salto de LTL contém

- uma etiqueta em caso de teste positivo
- uma outra etiqueta em caso de teste negativo

enquanto as instruções de salto do *assembly*

- contém uma única etiqueta para o caso positivo
- continuam a execução na instrução seguinte em caso de teste negativo

a linearização consiste no percurso do grafo de fluxo de controlo e na produção do código X86-64 enquanto se aponte numa tabela as etiquetas já visitada

aquando de um salto, esforçamo-nos quanto possível na produção de código *assembly* natural se a parte do código correspondente a um teste negativo não foi ainda visitado.
no pior dos casos, utilizamos um salto incondicional (`jmp`)

o código X86-64 é produzido sequencialmente com a ajuda de uma função

```
val emit: Label.t -> X86_64.text -> unit
```

utilizamos duas tabelas

uma primeira tabela para as etiquetas já visitadas

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

e uma segunda para as etiquetas que deverão ficar no código *assembly*
(não o sabemos no exacto momento onde uma instrução é produzida)

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

a linearização é efectuada por duas funções mutuamente recursiva

- `lin` produz o código a partir de uma dada etiqueta, se este ainda não foi produzido, e uma instrução de salto para esta etiqueta senão

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

- `instr` produz o código a partir de uma etiqueta e da instrução correspondente, sem condição

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

a função `lin` é um simples percurso de grafo

se a instrução não foi visitada, marcamo-la como visitada e chamamos `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)
```

senão marcamos a sua etiqueta como requerida no código *assembly* e produzimos um salto incondicional para esta etiqueta

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

a função `instr` produz efectivamente o código X86-64 e chama recursivamente `lin` sobre a etiqueta seguinte

```
and instr g l = function
  | Econst (n, r, l1) ->
      emit l (movq (imm32 n) (operand r)); lin g l1
  | Eaccess_global (x, r, l1) ->
      emit l (movq (lab x) (register r)); lin g l1
  | ...
```

o caso interessante é o do salto (consideramos aqui `Emubranch` ; o caso `Embbranch` é em tudo semelhante)

consideramos em primeiro o caso favorável que occorre quando o código correspondente a um teste negativo ainda não foi produzido

```
  | 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
```

(onde `ubranch` é a função que produz as instruções X86-64 de salto, ou seja `testq`/`cmpqq` e `j`*cc*)

senão, quando acontece que o código que corresponde a um teste positivo não foi ainda produzido, podemos tirar vantagem deste facto e **inverter a condição** de salto

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

onde

```
let inv_ubranch = function
  | Mjz  -> Mjnz
  | Mjnz -> Mjz
  | ...
```

finalmente, no caso em que o código correspondente aos dois ramos já foi produzido, não temos outra escolha senão produzir um salto incondicional

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

nota : podemos tentar estimar que condição é mais frequentemente verdade

o código contém numerosos `goto` (ciclos `while` na fase RTL, inserção de código na fase ERTL, remoção de instruções `mov` na fase LTL)

eliminamos aqui os `goto` quando possível

```
  | Egoto l1 ->
      if Hashtbl.mem visited l1 then begin
        need_label l1;
        emit l (jmp l1)
      end else begin
        emit l nop; (* ficará de facto removido *)
        lin g l1
      end
```

o programa principal encadeia todas as fases da compilação

```
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
X86_64.print_program fmt code;
close_out c
```

et voilà !

```
fact:   addq  $-8, %rsp
        movq  %rbx, 0(%rsp)
        movq  %rdi, %rbx
        cmpq  $1, %rbx
        jle   L8
        movq  %rbx, %rdi
        addq  $-1, %rdi
        call  fact
        imulq %rbx, %rax
L1:
        movq  0(%rsp), %rbx
        addq  $8, %rsp
        ret
L8:
        movq  $1, %rax
        jmp   L1
```

poderíamos ter feito melhor, manualmente

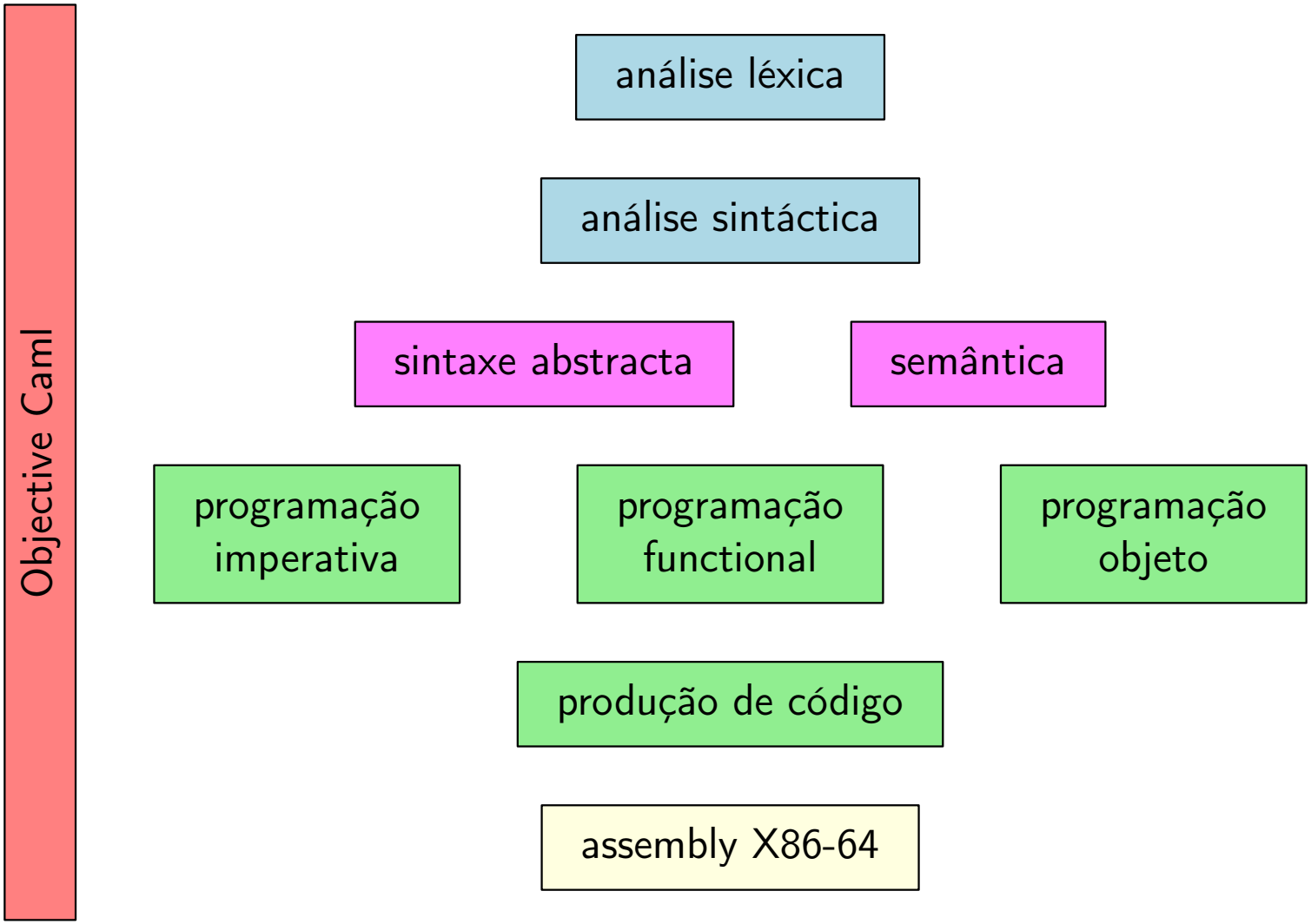
```
fact:   cmpq    $1, %rdi        # x <= 1 ?
        jle     L3
        pushq   %rdi            # salvaguarda x na pilha
        decq    %rdi
        call    fact            # fact(x-1)
        popq    %rcx
        imulq   %rcx, %rax      # x * fact(x-1)
        ret
L3:
        movq    $1, %rax
        ret
```

mas é sempre mais fácil optimizar **um** programa do que **todos**

	linhas de código
parsing	227
tipagem	170
seleção de instruções	103
RTL	203
ERTL	185
LTL	189
duração de vida	106
interferência	60
coloração	220
linearização	151
assembly	253
diversos	124
total	**1991**

vencemos o dragão!

que reter deste curso ?


Objective Caml
análise léxica
análise sintáctica
sintaxe abstracta
semântica
programação imperativa
programação functional
programação objeto
produção de código
assembly X86-64

entender, perceber as linguagens de programação é esencial para

- **programar** bem
  - ter um modelo de execução claro em mente
  - escolher as boas abstrações
- realizar **investigação** em informática
  - propor novas linguagens de programação
  - conceber ferramentas relacionadas

em particular, explicamos

- o que é a pilha
- os diferentes modos de passagem
- o que é um objecto
- o que é um fecho

mesmo se passamos algumas vezes sem grandes detalhes, cruzamo-nos com

- OCaml
- *assembly* X86-64
- Pascal
- C++
- Java
- C
- Scala
- Python
- Logo

a compilação implica

- o uso de numerosas técnicas
- combinadas em várias etapas, muitas vezes ortogonais

a maior parte destas técnicas são reutilizáveis em contextos diferentes do da produção de código máquina

- linguística
- demonstração assistida por computador
- *queries* em bases de dados

a compilação, é também...

muitas outras coisas que deixamos de lado nesta primeira abordagem à compilação

sistemas de módulos
SSA
common sub-expressions
transformação de programas
interpretação abstracta
análise de alias
inlining de ciclos
analise inter-procedimental
peephole optimization
pipeline
memória cache
programação lógica
compilação *on-the-fly*
instructions scheduling
etc.

conclusão

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)