Languages and Compilation

Based on the Jean-Christophe Filliâtre's Courses
given at École Polytechnique & École Normale Supérieure

Lecture 12 - optimizing compiler (3/3)

Léon Gondelman

AALBORG UNIVERISITY | COPENHAGEN | 2025

we took as example a fragment of C language

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

phase 1 : instruction selection

- replace C arithmetic operations with MIPS operations
- explicit memory access with constant offset over signed 16 bits

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

phase 2 : RTL (Register Transfer Language)

- from code as abstract syntax tree to **control-flow graph**
- **pseudo-registers** for function parameters and intermediate computations

```
%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
```

phase 3 : ERTL (Explicit Register Transfer Language)

- explicit calling conventions and instructions for handling stack frame

```
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 (Location Transfer Language)

replacing pseudo-registers with physical registers preferably and stack locations otherwise

```
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 (Linearization): the code after LTL is still a control-flow graph and we have to produce linear assembly code

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

register allocation is complex, and decomposed into several steps

1. we perform a **liveness analysis**
   - it tells when the value contained in a pseudo-register is needed for the remaining of the computation

2. we build an **interference graph**
   - it tells what are the pseudo-registers that cannot be mapped to the same location

3. we allocate registers using a **graph coloring**
   - it maps pseudo-registers to physical registers or stack locations

in the following, a *variable* stands for a pseudo-register or a physical register

Definition (live variable)

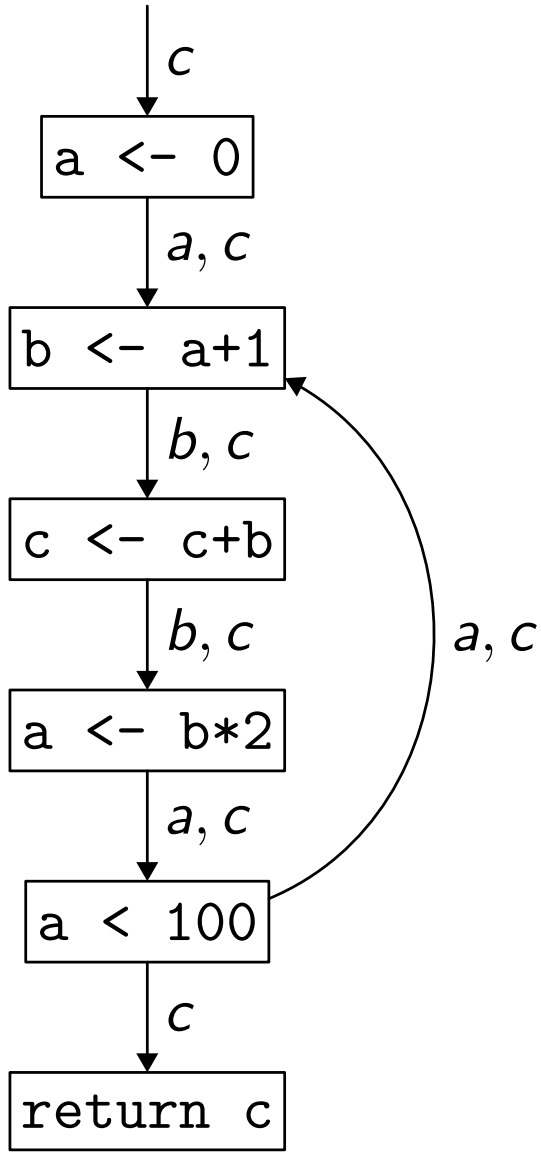
Given a program point, a variable is said to be **live** if the value it contains is likely to be used in the remaining of the computation.

we say "is likely" since "is used" is not decidable; so we seek for a sound over-approximation

Exemple

live variables are drawn on edges of the CFG

```
    a <- 0
L2: b <- a + 1
    c <- c + b
    a <- b * 2
    if a < 100 goto L2
    return c
```



live variables can be deduced from **definitions** and **uses** of variables by the various instructions

Definition

For an instruction at label l in the control-flow graph, we write

- $def(l)$ for the set of variables defined by this instruction,
- $use(l)$ for the set of variables used by this instruction.

example : for the instruction $l \equiv r_1 \leftarrow \text{add } r_2\ r_3$ we have

$$def(l) = \{r_1\} \quad \text{et} \quad use(l) = \{r_2, r_3\}$$

to compute live variables, it is handy to map them to labels in the control-flow graph (instead of edges)

but then we have to distinguish between variables **live at entry** and variables **live at exit** of a given instruction

Definition

For an instruction at label l in the control-flow graph, we write

- $in(l)$ for the set of live variables on the set of incoming edges to l,
- $out(l)$ for the set of live variables on the set of outcoming edges from l.

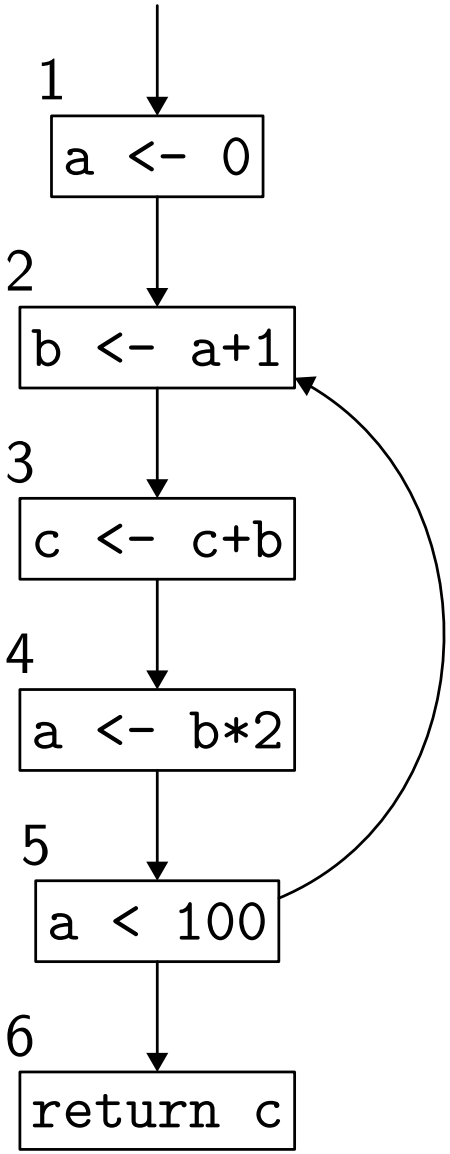
the equations defining $in(l)$ and $out(l)$ are the following

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

these are mutually recursive functions and we seek for the smallest solution

we are in the case of a monotonous function over a finite domain and thus we can use Tarski's theorem (see lecture 4)

fixpoint computation



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

	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	b	...	b,c	b,c
4	b	a	b		b	a	...	b,c	a,c
5	a		a	a	a	a,c	...	a,c	a,c
6	c		c		c		...	c	

we get the fixpoint with 7 iterations

assuming the control-flow graph has N nodes and N variables, a brute force computation has complexity $O(N^3)$ in the worst case

we can improve efficiency in several ways

- traversing the graph in "**reverse order**" and computing *out* before *in* (on the previous example, we converge in 3 iterations instead of 7)
- **merging nodes** with a unique predecessor and a unique successor (basic blocks)
- using a more subtle algorithm that only recomputes the *in* and *out* that may have changed; this is **Kildall's algorithm**

Kildall's algorithm

idea: if $in(l)$ changes, then we only need to redo the computation for the predecessors of l

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

here is the algorithm:

```
let WS be a set containing all nodes
while WS is not empty
    remove a node l from WS
    old_in <- in(l)
    out(l) <- ...
    in(l)  <- ...
    if in(l) is different from old_in(l) then
        add all predecessors of l in WS
```

computing *def* and *use*

computing the sets $def(l)$ (definitions) and $use(l)$ (uses) is straightforward for most instructions

examples

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

this is more subtle for **function calls**

calls: all *caller-saved* registers can be erased by the call and the $min(4, n)$ first registers of `parameters` may be used

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

system calls: only `$a0` and `$v0` are relevant

```
  | Esyscall l ->
      [v0], [a0; v0]
```

`return`: `$v0`, `$ra` and all *callee-saved* registers may be used

```
  | Ereturn ->
      [], result :: ra :: callee_saved
```

this was the ERTL code for `fact`

```
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
```

liveness for fact

liveness analysis gives us:

```
L19: alloc_frame  --> L18     in=$a0,$ra,$s0,$s1        out=$a0,$ra,$s0,$s1
L18: %10 <- $ra   --> L17     in=$a0,$ra,$s0,$s1        out=$a0,$s0,$s1
L17: %11 <- $s0   --> L16     in=$a0,$s0,$s1,%10        out=$a0,$s1,%10,%11
L16: %12 <- $s1   --> L15     in=$a0,$s1,%10,%11        out=$a0,%10,%11,%12
L15: %1 <- $a0    --> L11     in=$a0,%10,%11,%12        out=%1,%10,%11,%12
L11: %8 <- %1     --> L10     in=%1,%10,%11,%12         out=%1,%10,%11,%12,%8
L10: %9 <- 1      --> L9      in=%1,%10,%11,%12,%8      out=%1,%10,%11,%12,%8,%9
L9: ble %8 %9     --> L8, L7  in=%1,%10,%11,%12,%8,%9   out=%1,%10,%11,%12
L8: %2 <- 1       --> L1      in=%10,%11,%12            out=%10,%11,%12,%2
L1: goto L25                  in=%10,%11,%12,%2         out=%10,%11,%12,%2
L25: $v0 <- %2    --> L24     in=%10,%11,%12,%2         out=$v0,%10,%11,%12
L24: $ra <- %10   --> L23     in=$v0,%10,%11,%12        out=$ra,$v0,%11,%12
L23: $s0 <- %11   --> L22     in=$ra,$v0,%11,%12        out=$ra,$s0,$v0,%12
L22: $s1 <- %12   --> L21     in=$ra,$s0,$v0,%12        out=$ra,$s0,$s1,$v0
L21: delete_frame --> L20     in=$ra,$s0,$s1,$v0        out=$ra,$s0,$s1,$v0
L20: return                   in=$ra,$s0,$s1,$v0        out=
L7: %3 <- %1      --> L6      in=%1,%10,%11,%12         out=%1,%10,%11,%12,%3
L6: %6 <- %1      --> L5      in=%1,%10,%11,%12,%3      out=%10,%11,%12,%3,%6
L5: %7 <- 1       --> L4      in=%10,%11,%12,%3,%6      out=%10,%11,%12,%3,%6,%7
L4: %5 <- sub %6 %7-->L3      in=%10,%11,%12,%3,%6,%7   out=%10,%11,%12,%3,%5
L3: goto L14                  in=%10,%11,%12,%3,%5      out=%10,%11,%12,%3,%5
L14: $a0 <- %5   --> L13      in=%10,%11,%12,%3,%5      out=$a0,%10,%11,%12,%3
L13: call fact   --> L12      in=$a0,%10,%11,%12,%3     out=$v0,%10,%11,%12,%3
L12: %4 <- $v0    --> L2      in=$v0,%10,%11,%12,%3     out=%10,%11,%12,%3,%4
L2: %2 <- mul %3 %4-->L1      in=%10,%11,%12,%3,%4      out=%10,%11,%12,%2
```

we now build an **interference graph** that represents the constraints over pseudo-registers

Definition (interference)

We say that two variables v_1 and v_2 **interfere** if they cannot be implemented by the same location (physical register or memory slot).

since interference is not decidable, we look for sufficient conditions

let's consider an instruction

$$v \leftarrow e$$

that **defines** a variable v; then any other variable w live **out** of this instruction may interfere with v (mapping v to the register of w would make w not "live")

however, in the particular case of `move`

$$v \leftarrow w$$

we wish instead not to declare that v and w interfere, since mapping v and w to the same location will eliminate this instruction

so we adopt the following definition

Definition (interference graph)

The **interference graph** of a function is an undirected graph whose vertices are the variables and whose edges are of two kinds: interference or preference.
For each instruction I that defines a variable v and whose out live variables, other than v, are $w_1, \ldots, w_n$, we proceed as follows:

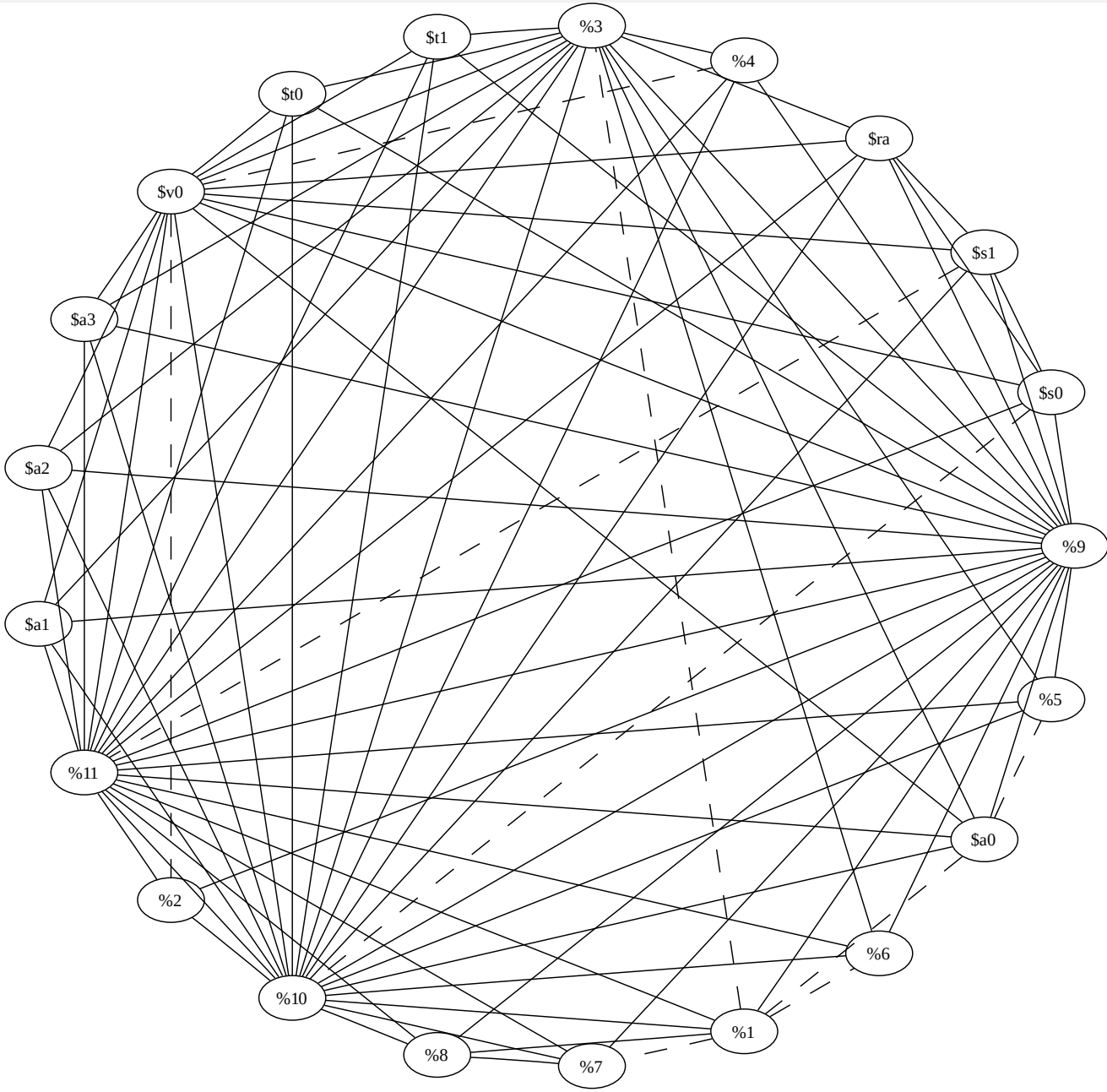
- if I is **not** a $v \leftarrow w$ move, we add the n **interference edges** $v - w_i$
- if I **is** a $v \leftarrow w$ move, we add the **interference edges** $v - w_i$ for the w_i **other than** w and we add a **preference edge** $v - w$.

(if an edge $v - w$ is both a preference and interference, we only keep the interference edge)

example : factorial

interference graph for `fact`

dashed=preference edges



4.3: graph coloring

we can see register allocation as a **graph coloring** problem:

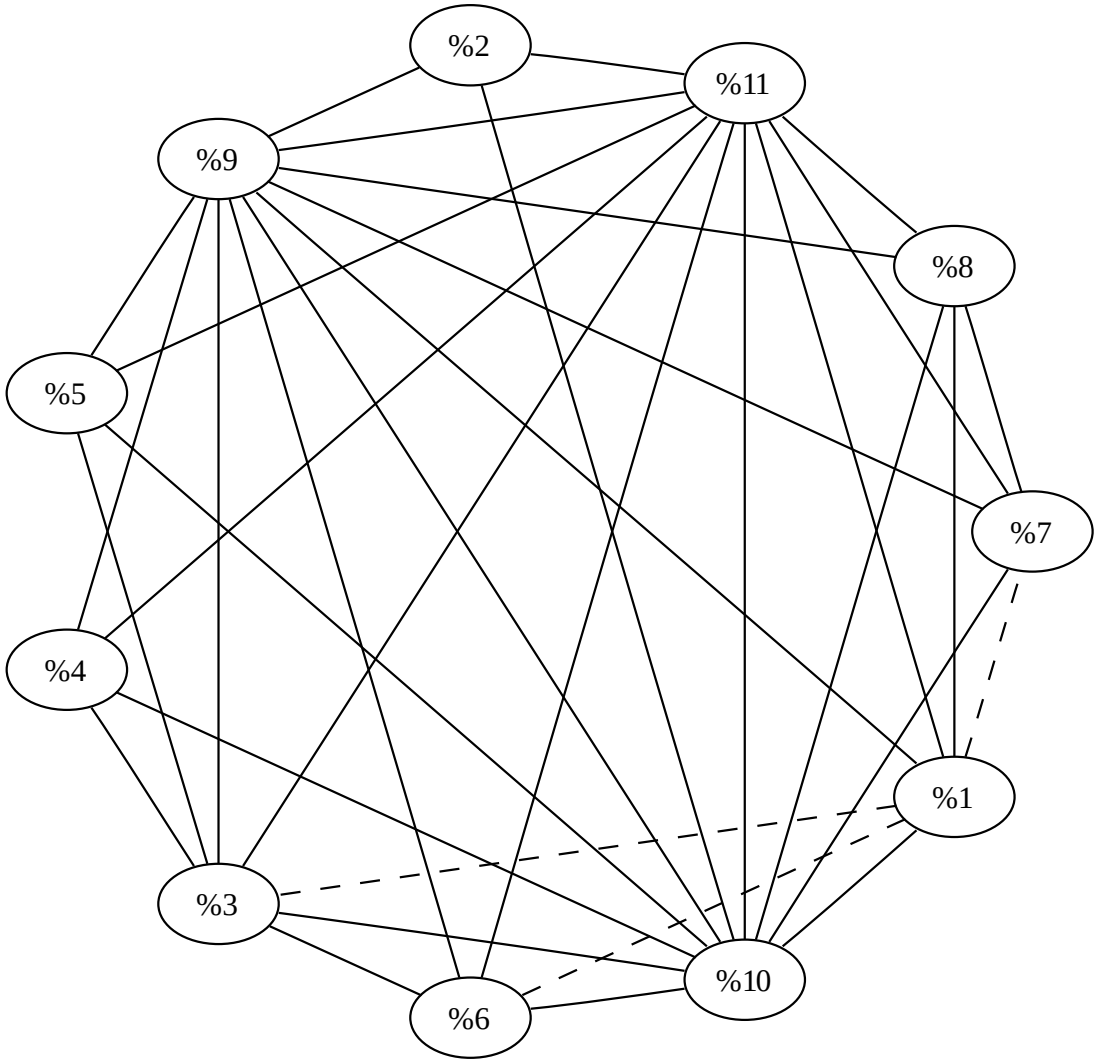
- the colors are the physical registers
- two vertices linked by some interference edge **cannot** receive the same color
- two vertices linked by some preference edge **should** receive the same color as much as possible

note: the graph contains vertices that are physical registers, *i.e.*, that are already colored

Gregory Chaitin, *Register allocation and spilling via graph coloring*, 1982

if we remove vertices that are already colored, we get the following coloring problem to solve

	possible colors
%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	



on this example, we can see immediately that the coloring is impossible

- no color available for %9 (the pseudo-register that will store `$ra`)
- `$s0` and `$s1` are the only possible colors for %10 and %11, but then there are no colors left for %3

if a vertex can't be colored, it will be **allocated on the stack**; it is called a *spilled* pseudo-register

another difficulty

even if the graph can be colored, figuring it out would be too costly (the problem is NP-complete)

so we are going to use **heuristiques**, to color the graph, looking for

- a linear (of quasi-linear) complexity
- good use of preference edges

one of the best algorithms is due to George and Appel
(*Iterated Register Coalescing*, 1996)

it uses the following ideas

let K be the number of colors (*i.e.* the number of physical registers)

a first idea, due to Kempe (1879!), is the following:

if a vertex has a degree $< K$, then we can remove it from the graph, color the remaining graph, and then assign it a color (because it has less interference edges than available colors); this is called **simplification**

removing a vertex decreases the degree of other vertices and thus can trigger other simplifications

removed vertices are put on a temporary stack of visited nodes

when there are only vertices with degree $\geq K$, we pick up one vertex as **potential spill**; it is removed from the graph and put on the temporary stack, and the simplification process restarts

we preferably choose a vertex

- that is seldom used (memory access is costly)
- has a strong degree (to favor new simplifications)

when the graph is empty, we start the coloring process, called **selection**

we pop vertices from the stack, and for each

- if it has a small degree, we are guaranteed to find a color
- if it has a high degree (a potential spill), then
  - either it can be colored because its neighbors use less than K colors (**optimistic coloring**)
  - or it cannot be colored and it is spilled to memory (**actual spill**)

last, we must make good use of preference edges

using a technique called **coalescing** that merges two vertices of the graph; since it may increase the degree (of the resulting vertex), we add a conservative criterion (George's criterion) not to damage K-colorability: coalescing x and y is safe if for every neighbor t of y, either

- t already interferes with x, or
- t has less than k neighbors, where k is the number of available registers.

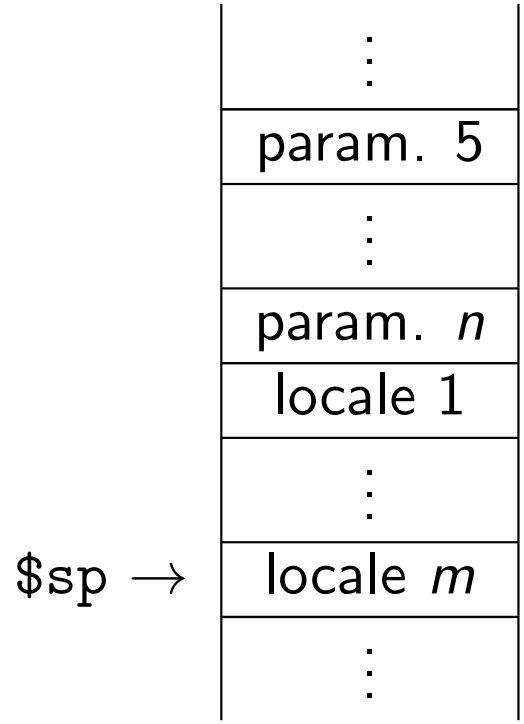
In simpler terms, if y's neighbors can be colored even after x and y are merged, then the coalescing is safe.

note: this phase is used after simplification but triggers a new simplification if we have a coalescing

what about *spilled* pseudo-registers?

what do we do with spilled pseudo-registers?

they are mapped to stack slots, in the lower part of the stack frame under the parameters



several pseudo-registers may use the same slot, if they do not interfere ⇒ how to minimize m ?

coloring, again this is yet another graph coloring problem, but this time with an infinite number of colors (stack slots)

algorithm :

1. merge all preference edges (coalescence),
   since `move` between two spilled registers is really costly
2. then use the simplification algorithm again

we get the following register allocation

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

which we *would* give the following code

```
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
```

as we notice, many instructions

$$v \leftarrow v$$

can now be eliminated; this was the purpose of preference edges

conversely, we have now instructions such as

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

which don't have direct correspondence in MIPS; what to do?

both cases will be taken care of during the translation to LTL
(see appendix of these slides)

so we get the following LTL code for `fact`

```
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
```

one last step is needed: the code is still a **control-flow graph** and we have to produce **linear assembly code**

to be precise, LTL branching instructions contain:

- a label for a positive test
- another label for a negative test

while assembler branching instructions:

- contain a single label for a positive test
- move to the next instruction for a negative test

the linearization consists in traversing the control-flow graph and outputting assembly code, while keeping track of visited labels

for a branching instruction, we try to produce idiomatic assembly code when the negative part of the code is not yet visited

in the worst case, we use some unconditional jump (`j`)

MIPS code is produced sequentially using a function

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

we use two tables

one to store visited labels

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

and one to store labels that are targets of jumps (we don't know yet when the instruction is visited and emitted)

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

the linearization is implemented by two mutually recursive functions

- a function `lin` outputs code from a given label, if not yet visited, and emits a jump to that label otherwise

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

- a function `instr` outputs code for a given label and a given instruction, unconditionally

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

the function `lin` is a mere **graph traversal**

if the instruction is not yet visited, we mark it as visited and we call function `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)
```

otherwise we mark the label as a target in the assembly code and we output some unconditional jump to that label

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

the function `instr` outputs MIPS code and calls `lin` recursively on the next label

```
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
  | ...
```

the interesting case is that of a branching instruction (let's consider `Emubranch` ; it's the same for `Embbranch`)

we first consider the case where the code corresponding to a negative test has not yet been produced

```
  | 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
```

(where `ubranch` is the function that produces MIPS instruction for the branching)

otherwise, it may be the case that the code corresponding to a positive test has not yet been produced and then we can **switch the condition**:

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

where

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

finally, in the case where both branches has already been visited, we have no other choice than emitting some unconditional jump

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

note: we can try to estimate which case will be true more often

the code contains many `goto` (`while` loops of the RTL phase, calling conventions in the ERTL phase, removal of `move` instructions in the LTL phase)

we now eliminate unnecessary `goto` when possible

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

the main program with all the compilation phases

```
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
```

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

```
fact__2:
        li   $v0, 1
        ble  $a0, $v0, L1
        addi $sp, $sp, -8
        sw   $ra, 4($sp)
        sw   $a0, 0($sp)
        addi $a0, $a0, -1
        jal  fact__2
        lw   $v1, 0($sp)
        mul  $v0, $v1, $v0
        lw   $ra, 4($sp)
        addi $sp, $sp, 8
L1:
        jr   $ra
```

but it is always easier to optimize **one** program

take away

understanding programming languages is essential for

- **coding**
  - have a precise execution model in mind
  - choose the right abstractions
- doing **research** in (applied/fundamental) Computer Science
  - design new languages
  - design tools (i.e. static analysis)

in particular, we explained

- what is the stack
- various passing modes
- what is an object
- static and dynamic typing

compilation involves

- numerous techniques
- several passes, mostly orthogonal

most of these techniques can be reused in contexts other than code generation, such as

- computational linguistics
- computer-assisted proofs
- databases

compilation also means...

many other things we didn't have time to explore

module systems
common sub-expression
program transformations
abstract interpretation
alias analysis
loop unrolling
interprocedural analysis
memory caches
logic programming
just-in-time compilation
instruction scheduling
etc.

...but in the end

there are no good programming languages,

only good programmers

appendix

we still have a control-flow graph
most LTL instructions LTL are the same as in ERTL, but operands are now physical registers or stack slots

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

additionally `Eget_stack_param` and `Eset_stack_param` disappear, being now replaced by general instructions manipulating the stack using `$sp`

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

we translate each ERTL instruction using a function that takes as arguments the graph coloring and the size of the stack frame (which is now known for each function)

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

let's consider the case of an instruction that loads the constant `n` in the variable `r` :

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

we have two possible cases:

- either `r` is physical register `h` or a pseudo-register mapped to a physical register `h`; then translation is straightforward

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

- or `r` is a spilled pseudo-register, and translation is not trivial: we need to load `n` in a physical register, then use the latter to write to memory

  problem : which physical register shall we use?

we go for a simple solution: two registers are used as temporary registers from transfers to/from memory, and are not used anywhere else (we choose (we choose `$v1` and `$fp` here)

in practice, we can't always waste two registers like this; we have to patch the interference graph and rerun the register allocation to free a register for the transfer (fortunately, it quickly converges (2 or 3 steps) in practice)

with two temporary registers

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

to write into the variable `r` we define a function `write`, that takes as arguments graph coloring `c` and the label for where to go after the write; the function then computes the physical register and the continuation label

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

we can now translate from ERTL to LTL:

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

conversely, we define a function `read1` to read the content of a variable (using a handler `f` in case of a physical register):

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

i.e. we use it as follows:

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

we proceed in a similar way when we need to read the content of two variables (binary operations) and use it as follows:

```
| 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))
```

the Mmove instruction requires a special treatment:

```
  | 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
```

and now that we know the size of the stack frame, we can translate `Eget_stack_param` in terms of access of it w.r.t. `$sp`

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

(but the `Eset_stack_param` does not change)

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

and we can translate `Ealloc_frame` and `Edelete_frame` in terms of `$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)
```

ERTL to LTL

```
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; }
```