

École Polytechnique

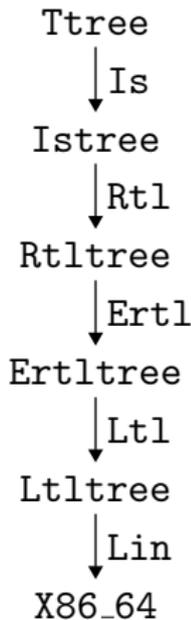
CSC_52064 – **Compilation**

Jean-Christophe Filliâtre

optimizing compiler (2/2)

code production is split into several phases:

1. instruction selection
2. RTL (*Register Transfer Language*)
3. ERTL (*Explicit Register Transfer Language*)
4. LTL (*Location Transfer Language*)
 - 4.1 liveness analysis
 - 4.2 interference graph
 - 4.3 register allocation
5. linearization (assembly)



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

phase 1: instruction selection

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

phase 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
```

phase 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

```

phase 4: LTL (*Location Transfer Language*)

we have already done the **liveness analysis** *i.e.* we have determined for each variable (pseudo-register or physical register) at which moments its value is likely to be used in the remaining of the computation

```

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

```

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 us consider an instruction that **defines** a variable v : then any other variable w live **out** of this instruction may interfere with v

however, in the particular case of

```
mov w v
```

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 that defines a variable v and whose out live variables, other than v , are w_1, \dots, w_n , we proceed as follows:

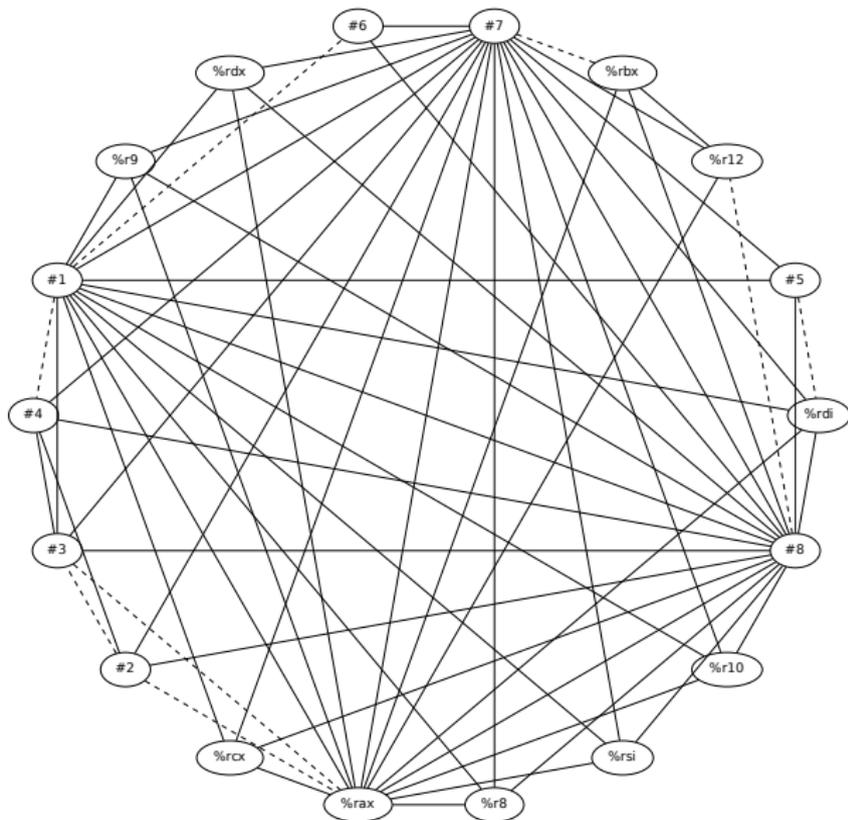
- if the instruction is not `mov w v`, we add the n interference edges $v - w_i$
- if this is an instruction `mov w v`, we add the interference edges $v - w_i$ for the w_i other than w and we add a preference edge $v - w$.

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

here is what we get
with function fact

10 physical registers
+
8 pseudo-registers

dashed = preference
edges



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

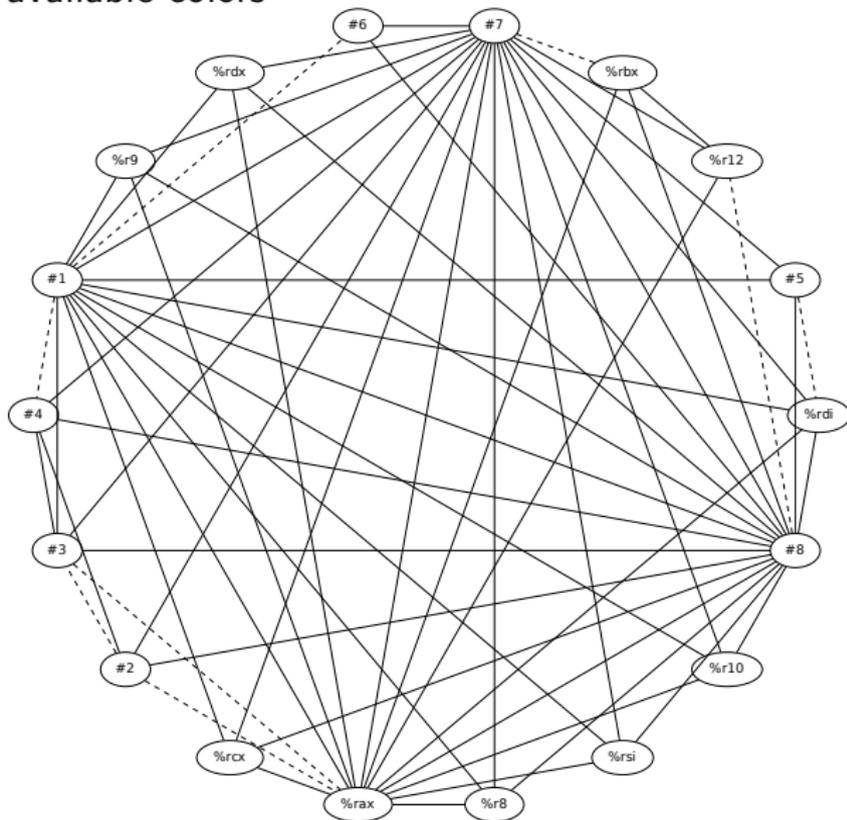
- the colors are the physical registers
- two vertices linked by some interference edge cannot receive the same color
- two vertices linked by some preference edge should receive the same color as much as possible

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

let us have a look at the available colors

	available colors
#1	%r12, %rbx
#2	all of them
#3	all of them
#4	all of them
#5	all of them
#6	all of them
#7	%rbx
#8	%r12



on this example, we immediately see that the graph coloring has no solution

- only two colors for #1, #7, and #8
- the three of them interfere

if a vertex cannot be colored, it will be allocated on the stack; it is called a **spilled register** (en français, un registre **vidé en mémoire**)

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 **heuristics** to color the graph, looking for

- a linear (or quasi-linear) complexity
- a 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; 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 stack

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 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

for this, we use 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 not to damage K -colorability

Definition (George's criterion)

A pseudo-register vertex v_2 can be merged with a vertex v_1 , if any neighbor of v_1 that is a physical register or has degree $\geq K$ is also a neighbor of v_2 .

Similarly, a physical vertex v_2 can be merged with a vertex v_1 , if any neighbor of v_1 that is a pseudo-register or has degree $\geq K$ is also a neighbor of v_2 .

the vertex v_1 is removed and the graph is updated

implemented with five mutually recursive functions

```
simplify(g) =  
    ...  
coalesce(g) =  
    ...  
freeze(g) =  
    ...  
spill(g) =  
    ...  
select(g, v) =  
    ...
```

note: the stack of vertices is thus implicit

```
simplify(g) =  
  if there exists a vertex v without any preference edge  
    with minimal degree < K  
  then  
    return select(g, v)  
  else  
    return coalesce(g)
```

```
coalesce(g) =  
  if there exists a preference edge v1-v2  
    satisfying George's criterion  
  then  
    g <- merge(g, v1, v2)  
    c <- simplify(g)  
    c[v1] <- c[v2]  
    return c  
  else  
    return freeze(g)
```

```
freeze(g) =  
  if there exists a vertex v with minimal degree < K  
  then  
    g <- remove preference edges from v  
    return simplify(g)  
  else  
    return spill(g)
```

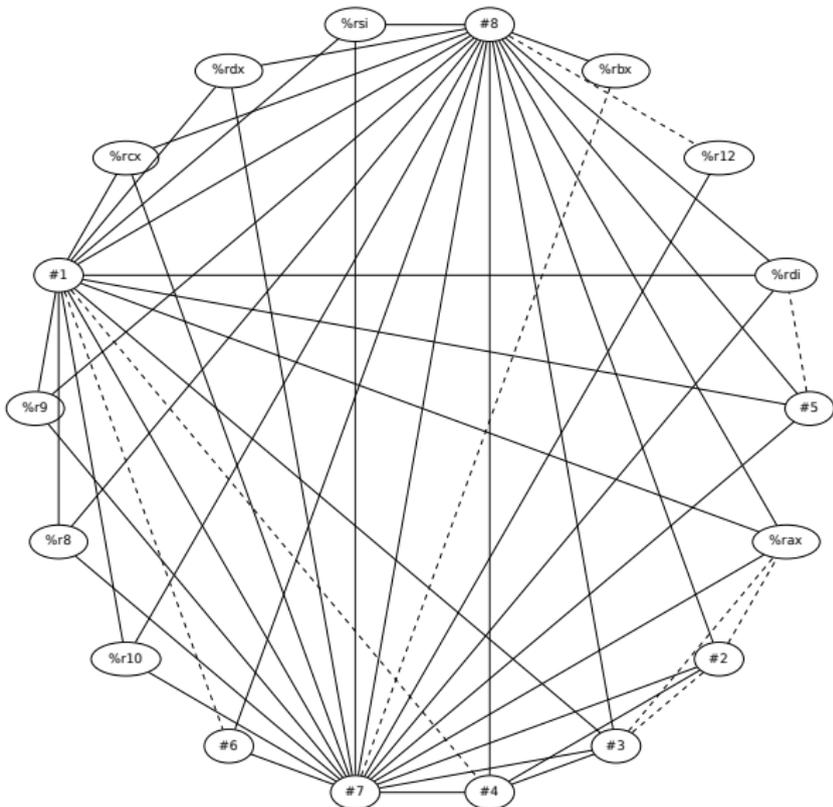
```
spill(g) =  
  if g is empty  
  then  
    return the empty coloring  
  else  
    choose a vertex v with minimal spill cost  
    return select(g, v)
```

the spill cost function can be for instance

$$\text{cost}(v) = \frac{\text{number of uses of } v}{\text{degree of } v}$$

```
select(g, v) =  
  remove vertex v from g  
  c ← simplify(g)  
  if there exists a color r for v  
  then  
    c[v] ← r  
  else  
    c[v] ← spill  
  return c
```

1.
`simplify(g) →`
`coalesce(g) →`
`selects #2- -#3`



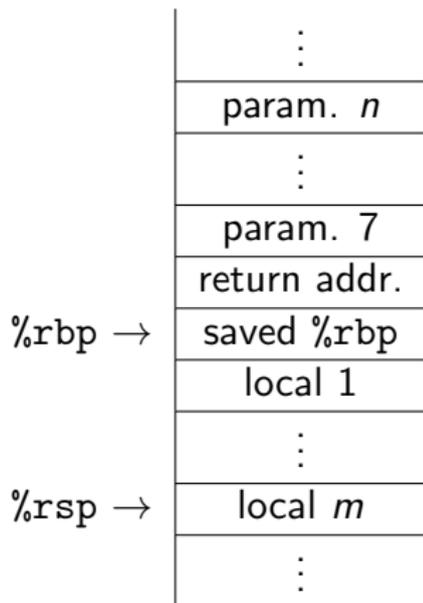
then we pop

8. coalesce #8- - %r12 → c[#8] = %r12
7. select #1 → c[#1] = %rbx
6. select #7 → c[#7] = spill
5. coalesce #5- - %rdi → c[#5] = %rdi
4. coalesce #3- - %rax → c[#3] = %rax
3. coalesce #6- - #1 → c[#6] = c[#1] = %rbx
2. coalesce #4- - #1 → c[#4] = c[#1] = %rbx
1. coalesce #2- - #3 → c[#2] = c[#3] = %rax

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



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

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

algorithm:

1. merge all preference edges, since `mov` between two spilled registers is really costly
2. then use the simplification algorithm

we get the following register allocation

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

which we *would* give the following code

```

fact(1)
  entry : L17

L17: alloc_frame      --> L16
L16: mov %rbx -8(%rbp)-->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 -8(%rbp) %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

```

as we notice, many instructions

```
mov v v
```

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

this will be done during the translation to LTL

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

`call $f \rightarrow L$`

identical to ERTL

`goto $\rightarrow L$` `return``load $n(r_1) r_2 \rightarrow L$`

identical to ERTL

`store $r_1 n(r_2) \rightarrow L$`

but with physical registers

`mov $n d \rightarrow L$`

identical to ERTL

`unop $op d \rightarrow L$`

but with operands

`binop $op d_1 d_2 \rightarrow L$` $(d = \text{register or stack slot})$ `ubbranch $br d \rightarrow L_1, L_2$` `bbranch $br d_1 d_2 \rightarrow L_1, L_2$` `push $d \rightarrow L$` `pop r`

new instruction

additionally, `alloc_frame`, `delete_frame`, and `get_param` disappear, being now replaced by explicit use of `%rsp` / `%rbp`

we translate each ERTL instruction into one or several LTL instructions, using

- the graph coloring
- the stack frame structure (which is now known for each function)

a variable x can be

- already a physical register
- a pseudo-register mapped to a physical register
- a pseudo-register mapped to a stack slot

in some cases, the translation is easy because the assembly language allows all combinations

example: the ERTL instruction

$$L_1 : \text{mov } n \ r \rightarrow L$$

is mapped to a single LTL instruction

$$L_1 : \text{mov } n \ \text{color}(r) \rightarrow L$$

color(r) being a physical register (e.g. `movq $42, %rax`) or a stack slot (e.g. `movq $42, -8(%rbp)`)

in other cases, however, this is more difficult as not all operand combinations are allowed

memory access is one such example:

$$L_1 : \text{load } n(r_1) \ r_2 \rightarrow L$$

raises an issue when r_2 is on the stack, as we can't write

```
movq n(r1), m(%rbp)
```

(too many memory references for 'movq')

similarly when r_1 is on the stack

we have to use some intermediate register

problem: which register to 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 `%r10` and `%r11`)

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)

with two temporary registers, it is now easy to translate ERTL to LTL

example: with ERTL instruction

$$L_1: \text{load } n(r_1) \ r_2 \rightarrow L$$

	r_2 physical register	r_2 on stack
r_1 physical register	$L_1: \text{load } n(r_1) \ r_2 \rightarrow L$	$L_1: \text{load } n(r_1) \ \%r10 \rightarrow L_2$ $L_2: \text{mov } \%r10 \ n_2(\%rbp) \rightarrow L$
r_1 on stack	$L_1: \text{mov } n_1(\%rbp) \ \%r10 \rightarrow L_2$ $L_2: \text{load } n(\%r10) \ r_2 \rightarrow L$	$L_1: \text{mov } n_1(\%rbp) \ \%r10 \rightarrow L_2$ $L_2: \text{load } n(\%r10) \ \%r11 \rightarrow L_3$ $L_3: \text{mov } \%r11 \ n_2(\%rbp) \rightarrow L_2$

(here one temporary register is enough but two are needed for store)

we make some special treatment during the translation

- an instruction `mov r1 r2 → L` is translated to `goto → L` when r_1 and r_2 have the same color

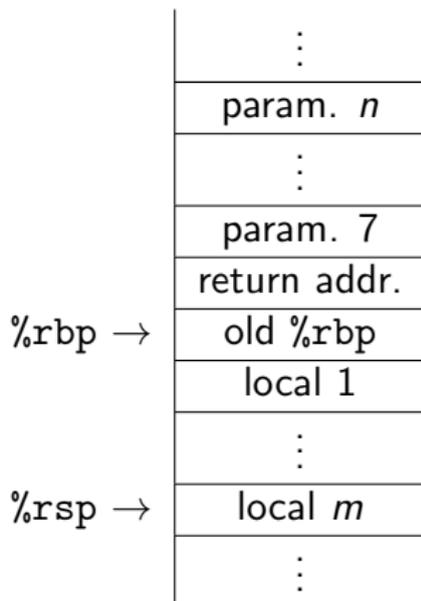
this is where we get the benefits of a good register allocation

- the x86-64 instruction `imul` requires its second operand to be a register \Rightarrow one has to use a temporary if this is not the case
- a binary operation cannot have two memory operands \Rightarrow use a temporary if needed

we can now translate `alloc_frame` and `delete_frame` into explicit use of `%rsp` / `%rbp`

ERTL	LTL
<code>alloc_frame → L</code>	<code>push %rbp</code> <code>mov %rsp %rbp</code> <code>add -8m %rsp</code>
<code>delete_frame → L</code>	<code>mov %rbp %rsp</code> <code>pop %rbp</code>

(which simplifies when $m = 0$)



to translate a function f

1. make the liveness analysis
2. build the interference graph
3. color it
4. deduce the value of m
5. translate ERTL instructions to LTL

for the factorial, we get the following LTL code

```

fact()
  entry : L17
  L17: add $-8 %rsp      --> L16
  L16: mov %rbx -8(%rbp) --> 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 -8(%rbp) %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

```

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 assemble 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 (`jmp`)

we use two tables

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

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

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`
- otherwise we mark the label as a target and we output some unconditional jump to that label

the function `instr` outputs x86-64 code and calls `lin` recursively on the next label

```
instr( $L_1 : \text{mov } n \ d \rightarrow L$ ) =      output  $L_1 : \text{movq } n, \ d$   
                                         call lin(L)  
instr( $L_1 : \text{load } n(r_1) \ r_2 \rightarrow L$ ) = output  $L_1 : \text{movq } n(r_1), \ r_2$   
                                         call lin(L)  
etc.
```

the interesting case is that of a branching instruction

we first consider the case where the negative label (L_3) is not yet visited

$$\text{instr}(L_1 : \text{branch } cc \rightarrow L_2, L_3) = \begin{array}{l} \text{output } L_1 : \text{jcc } L_2 \\ \text{call lin}(L_3) \\ \text{call lin}(L_2) \end{array}$$

otherwise, it may be the case that the positive label (L_2) is not yet visited and we can **switch the condition**

$$\text{instr}(L_1 : \text{branch } cc \rightarrow L_2, L_3) = \begin{array}{l} \text{output } L_1 : j\overline{cc} L_3 \\ \text{call lin}(L_2) \\ \text{call lin}(L_3) \end{array}$$

where condition \overline{cc} is the opposite of condition cc

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

$$\text{instr}(L_1 : \text{branch } cc \rightarrow L_2, L_3) = \begin{array}{l} \text{output } L_1 : \text{jcc } L_2 \\ \text{output jmp } L_3 \end{array}$$

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

the code contains many goto (C while loops in the RTL phase, calling conventions in the ERTL phase, removal of mov in the LTL phase)

we now eliminate unnecessary gotos when possible

$$\begin{aligned} \text{instr}(L_1 : \text{goto} \rightarrow L_2) &= \text{output jmp } L_2 \text{ if } L_2 \text{ is already visited} \\ &= \text{output label } L_1 \\ &\quad \text{call lin}(L_2) \qquad \text{otherwise} \end{aligned}$$

et voilà !

```

fact: pushq %rbp
      movq %rsp, %rbp
      addq $-8, %rsp
      movq %rbx, -8(%rbp)
      movq %rdi, %rbx
      cmpq $1, %rbx
      jle L8
      movq %rbx, %rdi      ## useless, too bad!
      addq $-1, %rdi
      call fact
      imulq %rbx, %rax

```

```

L1:
      movq -8(%rbp), %rbx
      movq %rbp, %rsp
      popq %rbp
      ret

```

```

L8:
      movq $1, %rax
      jmp L1

```

we could do better **manually**

```
fact: cmpq    $1, %rdi    # x <= 1 ?
      jle    L3
      pushq  %rdi       # saves x on the stack
      decq   %rdi
      call   fact       # fact(x-1)
      popq   %rcx
      imulq %rcx, %rax  # x * fact(x-1)
      ret

L3:   movq    $1, %rax
      ret
```

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

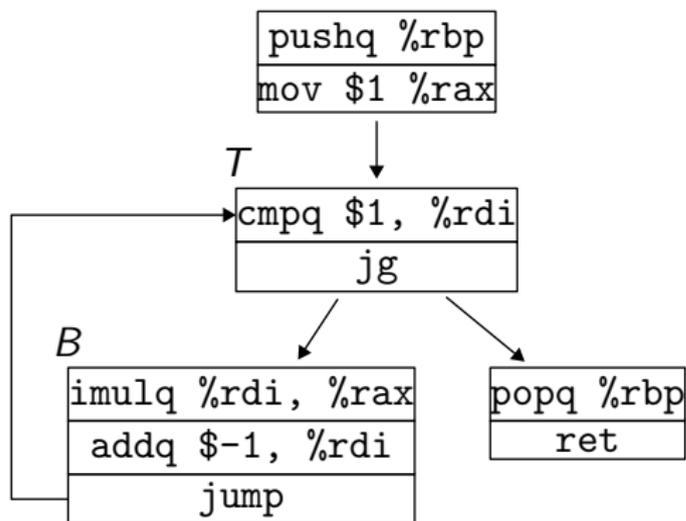
```
int fib(int n) {
    if (n <= 1) return n;
    return fib(n-2) + fib(n-1);
}
```

```
fib: pushq %rbp
     movq %rsp, %rbp
     addq $-16, %rsp
     movq %rbx, -16(%rbp)
     movq %r12, -8(%rbp)
     movq %rdi, %rbx
     cmpq $1, %rbx
     jle  L14
     movq %rbx, %rdi
     addq $-2, %rdi
     call fib
     movq %rax, %r12
```

```
     addq $-1, %rbx
     movq %rbx, %rdi
     call fib
     addq %rax, %r12
L6:  movq %r12, %rax
     movq -16(%rbp), %rbx
     movq -8(%rbp), %r12
     movq %rbp, %rsp
     popq %rbp
     ret
L14: movq %rbx, %r12
     jmp L6
```

```
int loop(int x) {  
    int r;  
    r = 1;  
    while (x > 1) {  
        r = r * x;  
        x = x - 1;  
    }  
    return r;  
}
```

```
loop: pushq %rbp  
      movq $1, %rax  
T:    cmpq $1, %rdi  
      jg B  
      popq %rbp  
      ret  
B:    imulq %rdi, %rax  
      addq $-1, %rdi  
      jmp T
```

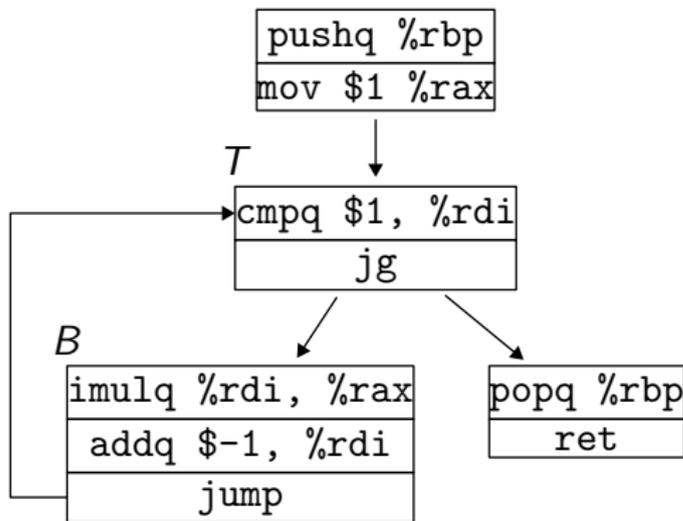


```

loop: pushq %rbp
      movq  $1, %rax
T:    cmpq  $1, %rdi
      jg   B
      popq %rbp
      ret
B:    imulq %rdi, %rax
      addq  $-1, %rdi
      jmp  T
  
```

a third example, improved

a better linearization, with only one branching per iteration



```
loop: pushq %rbp
      movq $1, %rax
      jmp T
B:    imulq %rdi, %rax
      addq $-1, %rdi
T:    cmpq $1, %rdi
      jg B
      popq %rbp
      ret
```

gcc and clang typically do that (though via loop unrolling instead)

our compiler roughly matches gcc -O1

	gcc -O1	mini-C
fib(42)	3,43	3,67
factorial 10^9 with a loop	0,86	0,86
10 000 times tak(18,12,6)	1,40	1,75

```
int tak(int x, int y, int z) { // Ikuo Takeuchi
    if (y < x)
        return tak(tak(x-1, y, z), tak(y-1, z, x), tak(z-1, x, y));
    return z; }
```

other compiler architectures

- the architecture we used here is that of **CompCert**
 - optimizations are implemented at the RTL level
- the **gcc** compiler inserts some SSA language (explained later)

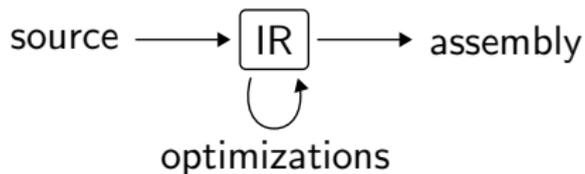
frontend \rightarrow SSA \rightarrow RTL $\rightarrow \dots$

and optimizations are implemented at both SSA and RTL levels

- the **clang** compiler is built on LLVM

this is a platform to help building optimizing compilers

LLVM offers an intermediate language, IR, and tools to optimize and compile this language



the C compiler clang is built on LLVM

one can get the IR code with

```
> clang -O1 -c -emit-llvm fact.c -o fact.bc
```

and make it readable with

```
> llvm-dis fact.bc -o fact.ll
```

```

define i32 @fact(i32) {
  %2 = icmp slt i32 %0, 2
  br i1 %2, label %10, label %3
; <label>:3:                                ; preds = %1
  br label %4
; <label>:4:                                ; preds = %3, %4
  %5 = phi i32 [ %7, %4 ], [ %0, %3 ]
  %6 = phi i32 [ %8, %4 ], [ 1, %3 ]
  %7 = add nsw i32 %5, -1
  %8 = mul nsw i32 %5, %6
  %9 = icmp slt i32 %5, 3
  br i1 %9, label %10, label %4
; <label>:10:                               ; preds = %4, %1
  %11 = phi i32 [ 1, %1 ], [ %8, %4 ]
  ret i32 %11
}

```

```
define i32 @fact(i32 %x0) {  
L1:  
    %x2 = icmp slt i32 %x0, 2  
    br i1 %x2, label %L10, label %L3  
L3:  
    br label %L4  
L4:  
    %x5 = phi i32 [ %x7, %L4 ], [ %x0, %L3 ]  
    %x6 = phi i32 [ %x8, %L4 ], [ 1, %L3 ]  
    %x7 = add nsw i32 %x5, -1  
    %x8 = mul nsw i32 %x5, %x6  
    %x9 = icmp slt i32 %x5, 3  
    br i1 %x9, label %L10, label %L4  
L10:  
    %x11 = phi i32 [ 1, %L1 ], [ %x8, %L4 ]  
    ret i32 %x11  
}
```

the IR language is much like our RTL language

- pseudo-registers (%2, %5, %6, etc.)
- a control-flow graph
- high-level calls

but there are also differences

- it is a typed language
- the code is **in SSA form** (*Single Static Assignment*): each variable is only assigned once

of course, the code we compile is likely to assign a variable multiple times

we make use of a Φ **operator** to reconcile several branches of the control-flow graph

for instance,

```
%x5 = phi i32 [ %x7, %L4 ], [ %x0, %L3 ]
```

means that %x5 receives the value of %x7 if we come from block %L4 and the value of %x0 if we come from block %L3

the benefits of SSA form are that we can now

- **attach** a property to each variable
(e.g. to be equal to 42, to be positive, to be in [34,55], etc.)
- exploit it **everywhere** this variable is used

the SSA form eases many optimizations

we get assembly code with the LLVM compiler

```
> llc fact.bc -o fact.s
```

this phase includes

- making calling conventions explicit (\approx ERTL)
- register allocation (\approx LTL)
- linearization

this is register allocation that gets rid of Φ operators (a few `mov` may be necessary)

```
> llc fact.bc -o fact.s
```

```
fact: movl    $1, %eax
      cmpl   $2, %edi
      jl     L3
L2:   imull  %edi, %eax
      leal  -1(%rdi), %ecx
      cmpl  $2, %edi
      movl  %ecx, %edi # <- was phi
      jg     2
L3:   ret
```

register allocation

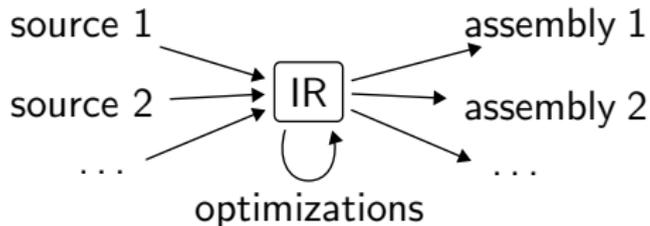
%x11	%x6	%x8	%eax
	%x0	%x5	%edi
		%x7	%ecx

one can make use of LLVM to

- implement a new compiler for a language S with only a frontend and a translation to IR

and/or

- design and implement new optimizations, over IR



- lab 8
 - mini Java continued
- lecture 9
 - garbage collection