École Polytechnique

INF564 – Compilation

Jean-Christophe Filliâtre

code production (3/3)
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*)

```assembly
#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*)

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Next Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>L17</td>
<td>alloc_frame</td>
<td>L16</td>
</tr>
<tr>
<td>L16</td>
<td>mov %rbx #7</td>
<td>L15</td>
</tr>
<tr>
<td>L15</td>
<td>mov %r12 #8</td>
<td>L14</td>
</tr>
<tr>
<td>L14</td>
<td>mov %rdi #1</td>
<td>L10</td>
</tr>
<tr>
<td>L10</td>
<td>mov #1 #6</td>
<td>L9</td>
</tr>
<tr>
<td>L9</td>
<td>jle $1 #6</td>
<td>L8, L7</td>
</tr>
<tr>
<td>L8</td>
<td>mov $1 #2</td>
<td>L1</td>
</tr>
<tr>
<td>L1</td>
<td>goto</td>
<td>L22</td>
</tr>
<tr>
<td>L22</td>
<td>mov #2 %rax</td>
<td>L21</td>
</tr>
<tr>
<td>L21</td>
<td>mov #7 %rbx</td>
<td>L20</td>
</tr>
<tr>
<td>L20</td>
<td>mov #8 %r12</td>
<td>L19</td>
</tr>
<tr>
<td>L19</td>
<td>delete_frame</td>
<td>L18</td>
</tr>
<tr>
<td>L18</td>
<td>return</td>
<td></td>
</tr>
<tr>
<td>L7</td>
<td>mov #1 #5</td>
<td>L6</td>
</tr>
<tr>
<td>L6</td>
<td>add $-1 #5</td>
<td>L5</td>
</tr>
<tr>
<td>L5</td>
<td>goto</td>
<td>L13</td>
</tr>
<tr>
<td>L13</td>
<td>mov #5 %rdi</td>
<td>L12</td>
</tr>
<tr>
<td>L12</td>
<td>call fact(1)</td>
<td>L11</td>
</tr>
<tr>
<td>L11</td>
<td>mov %rax #3</td>
<td>L4</td>
</tr>
<tr>
<td>L4</td>
<td>mov #1 #4</td>
<td>L3</td>
</tr>
<tr>
<td>L3</td>
<td>mov #3 #2</td>
<td>L2</td>
</tr>
<tr>
<td>L2</td>
<td>imul #4 #2</td>
<td>L1</td>
</tr>
</tbody>
</table>

Jean-Christophe Filliâtre

INF564 – Compilation

code production (3/3) 5
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,%rdi 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

$$\text{mov } w \ v$$

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 $w_1, \ldots, w_n$, we proceed as follows:

- **if the instruction is not** \texttt{mov} \texttt{w v}, \texttt{we add the n interference edges} $v \rightarrow w_i$

- **if this is an instruction** \texttt{mov} \texttt{w v}, \texttt{we add the interference edges} $v \rightarrow w_i$ for the $w_i$ other than $w$ and \texttt{we add a preference edge} $v \rightarrow w$.

(if an edge $v \rightarrow w$ is both a preference and interference, we only keep the interference edge)
example: factorial

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
let us have a look at the available colors

<table>
<thead>
<tr>
<th></th>
<th>available colors</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>%r12, %rbx</td>
</tr>
<tr>
<td>#2</td>
<td>all of them</td>
</tr>
<tr>
<td>#3</td>
<td>all of them</td>
</tr>
<tr>
<td>#4</td>
<td>all of them</td>
</tr>
<tr>
<td>#5</td>
<td>all of them</td>
</tr>
<tr>
<td>#6</td>
<td>all of them</td>
</tr>
<tr>
<td>#7</td>
<td>%rbx</td>
</tr>
<tr>
<td>#8</td>
<td>%r12</td>
</tr>
</tbody>
</table>
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**).
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 **heuristics** to color the graph, looking for

- a linear (of 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
George’s criterion

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
George-Appel algorithm

implemented with five mutually recursive functions

\begin{align*}
simplify(g) &= 
  
  \ldots 
  
  
  \text{coalesce}(g) &= 
  
  \ldots 
  
  \text{freeze}(g) &= 
  
  \ldots 
  
  \text{spill}(g) &= 
  
  \ldots 
  
  \text{select}(g, v) &= 
  
  \ldots 
\end{align*}

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

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

\[
\begin{array}{l}
\text{\%rbp} \rightarrow \text{old \%rbp} \\
\text{\%rsp} \rightarrow \text{local } m \\
\text{\vdots}
\end{array}
\]

several pseudo-registers may use the same slot, if they do not interfere ⇒ 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

\[
\begin{array}{l}
\text{fact}(1) \\
\text{entry : L17} \\
\begin{align*}
\text{L17: } \text{alloc_frame} & \quad \rightarrow \text{L16} \\
\text{L16: } \text{mov} \ %\text{rbx} & \ -8(%\text{rbp}) \rightarrow \text{L15} \\
\text{L15: } \text{mov} \ %\text{r12} & \ %\text{r12} \quad \rightarrow \text{L14} \\
\text{L14: } \text{mov} \ %\text{rdi} & \ %\text{rbx} \quad \rightarrow \text{L10} \\
\text{L10: } \text{mov} \ %\text{rbx} & \ %\text{rbx} \quad \rightarrow \text{L9} \\
\text{L9 : } \text{jle} \ %\text{rbx} & \quad \rightarrow \text{L8, L7} \\
\text{L8 : } \text{mov} \ %\text{r1} & \ %\text{rax} \quad \rightarrow \text{L1} \\
\text{L1 : } \text{goto} & \quad \rightarrow \text{L22} \\
\text{L22: } \text{mov} \ %\text{rax} & \ %\text{rax} \quad \rightarrow \text{L21} \\
\text{L21: } \text{mov} \ -8(%\text{rbp}) & \ %\text{rbx} \rightarrow \text{L20} \\
\end{align*}
\end{array}
\]

\[
\begin{array}{l}
\begin{align*}
\text{L20: } \text{mov} \ %\text{r12} & \ %\text{r12} \quad \rightarrow \text{L19} \\
\text{L19: } \text{delete_frame} & \quad \rightarrow \text{L18} \\
\text{L18: } \text{return} & \\
\text{L7 : } \text{mov} \ %\text{rbx} & \ %\text{rdi} \quad \rightarrow \text{L6} \\
\text{L6 : } \text{add} \ -1 \ %\text{rdi} & \quad \rightarrow \text{L5} \\
\text{L5 : } \text{goto} & \quad \rightarrow \text{L13} \\
\text{L13: } \text{mov} \ %\text{rdi} & \ %\text{rdi} \quad \rightarrow \text{L12} \\
\text{L12: } \text{call} \ \text{fact}(1) & \quad \rightarrow \text{L11} \\
\text{L11: } \text{mov} \ %\text{rax} & \ %\text{rax} \quad \rightarrow \text{L4} \\
\text{L4 : } \text{mov} \ %\text{rbx} & \ %\text{rbx} \quad \rightarrow \text{L3} \\
\text{L3 : } \text{mov} \ %\text{rax} & \ %\text{rax} \quad \rightarrow \text{L2} \\
\text{L2 : } \text{imul} \ %\text{rbx} & \ %\text{rax} \quad \rightarrow \text{L1} \\
\end{align*}
\end{array}
\]
as we notice, many instructions

\[ \text{mov } v \rightarrow v \]

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

dthis will be done during the translation to LTL
the LTL language

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
LTL instructions

\begin{itemize}
  \item \texttt{call} \( f \rightarrow L \) identical to ERTL
  \item \texttt{goto} \( \rightarrow L \)
  \item \texttt{return}
  \item \texttt{load} \( n(r_1) \ r_2 \rightarrow L \) identical to ERTL
  \item \texttt{store} \( r_1 \ n(r_2) \rightarrow L \) but with physical registers
  \item \texttt{mov} \( n \ d \rightarrow L \) identical to ERTL
  \item \texttt{unop} \( op \ d \rightarrow L \) but with operands
  \item \texttt{binop} \( op \ d_1 \ d_2 \rightarrow L \) \((d = \text{register or stack slot})\)
  \item \texttt{ubranch} \( br \ d \rightarrow L_1, L_2 \)
  \item \texttt{bbranch} \( br \ d_1 \ d_2 \rightarrow L_1, L_2 \)
  \item \texttt{push} \( d \rightarrow L \)
  \item \texttt{pop} \( r \) new instruction
\end{itemize}
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

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

\(color(r)\) being a physical register (e.g. \(\text{movq} \ \$42, %rax\)) or a stack slot (e.g. \(\text{movq} \ \$42, -8(%rbp)\))
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:

\[
\text{movq } n(r1), m(%rbp)
\]

(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?
temporary registers

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

\[ L_1: \text{load } n(r_1) \quad r_2 \rightarrow L \]

<table>
<thead>
<tr>
<th></th>
<th>( r_2 \text{ physical register} )</th>
<th>( r_2 \text{ on stack} )</th>
</tr>
</thead>
</table>
| \( r_1 \text{ physical register} \) | \( L_1: \text{load } n(r_1) \quad r_2 \rightarrow L \) | \( L_1: \text{load } n(r_1) \quad \%r10 \rightarrow L_2 \)
|               | \( L_2: \text{load } n(\%r10) \quad r_2 \rightarrow L \) | \( L_2: \text{mov } \%r10 \quad n_2(\%rbp) \rightarrow L \) |
| \( r_1 \text{ on stack} \)        | \( L_1: \text{mov } n_1(\%rbp) \quad \%r10 \rightarrow L_2 \) | \( L_1: \text{mov } n_1(\%rbp) \quad \%r10 \rightarrow L_2 \)
|               | \( L_2: \text{load } n(\%r10) \quad \%r11 \rightarrow L_3 \) | \( L_2: \text{load } n(\%r10) \quad \%r11 \rightarrow L_3 \)
|               | \( L_3: \text{mov } \%r11 \quad n_2(\%rbp) \rightarrow L_2 \) | \( L_3: \text{mov } \%r11 \quad n_2(\%rbp) \rightarrow L_2 \) |

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

We make some special treatment during the translation:

- an instruction \texttt{mov} \( r_1 \ r_2 \rightarrow L \) is translated to \texttt{goto} \( \rightarrow L \) 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 \texttt{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

<table>
<thead>
<tr>
<th>ERTL</th>
<th>LTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>alloc_frame → L</td>
<td>push %rbp</td>
</tr>
<tr>
<td></td>
<td>mov %rsp %rbp</td>
</tr>
<tr>
<td></td>
<td>add −8m %rsp</td>
</tr>
<tr>
<td>delete_frame → L</td>
<td>mov %rbp %rsp</td>
</tr>
<tr>
<td></td>
<td>pop %rbp</td>
</tr>
</tbody>
</table>

(which simplifies when \( m = 0 \))
translating ERTL to LTL

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

```c
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: 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 \texttt{instr} outputs x86-64 code and calls \texttt{lin} recursively on the next label

\[
\texttt{instr}(L_1: \texttt{mov} n \; d \rightarrow L) = \quad \text{output } L_1: \texttt{movq} \; n, \; d \\
\text{call lin}(L)
\]

\[
\texttt{instr}(L_1: \texttt{load} \; n(r_1) \; r_2 \rightarrow L) = \quad \text{output } L_1: \texttt{movq} \; n(r_1), \; r_2 \\
\text{call lin}(L)
\]

\textit{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

\[
\text{instr}(L_1 : \text{branch } cc \rightarrow L_2, L_3) = \begin{cases} 
\text{output } L_1 : \text{jcc } L_2 \\
\text{call lin}(L_3) \\
\text{call lin}(L_2)
\end{cases}
\]
otherwise, it may be the case that the positive label ($L_2$) is not yet visited and we can **switch the condition**

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

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

\[
\text{instr}(L_1 : \text{branch } cc \rightarrow L_2, L_3) = \begin{cases} 
\text{output } L_1 : \text{jcc } L_2 \\
\text{output jmp } L_3
\end{cases}
\]

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

\[
\text{instr}(L_1 : \text{goto} \rightarrow L_2) = \begin{cases} 
\text{output jmp } L_2 & \text{if } L_2 \text{ is already visited} \\
\text{output label } L_1 \text{ call lin}(L_2) & \text{otherwise}
\end{cases}
\]
et voilà !
factorial

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

```assembly
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
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)
  
  \[
  \text{frontend} \rightarrow \text{SSA} \rightarrow \text{RTL} \rightarrow \cdots
  \]

  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

\[
\text{source} \rightarrow \text{IR} \rightarrow \text{assembly} \rightarrow \text{optimizations}
\]
the C compiler clang is built on LLVM

one can get the IR code with

```bash
> clang -01 -c -emit-llvm fact.c -o fact.bc
```

and make it readable with

```bash
> 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 \texttt{\phi} \textbf{operator} to reconcile several branches of the control-flow graph

for instance,

\[
\%x5 = \texttt{\phi i32} \ [ \%x7, \%L4 \ ], \ [ \%x0, \%L3 \ ]
\]

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

during this phase includes

- making calling conventions explicit (≈ ERTL)
- register allocation (≈ 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**

<table>
<thead>
<tr>
<th>%x11</th>
<th>%x6</th>
<th>%x8</th>
<th>%eax</th>
</tr>
</thead>
<tbody>
<tr>
<td>%x0</td>
<td>%x5</td>
<td></td>
<td>%edi</td>
</tr>
<tr>
<td></td>
<td>%x7</td>
<td>%ecx</td>
<td></td>
</tr>
</tbody>
</table>
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

![Diagram showing source code translation and optimizations](https://via.placeholder.com/150)
lab 8

LTL code production
assembly code
some code is provided (for OCaml and Java)

- LTL abstract syntax
- LTL interpreter to test
- LTL pretty-printer to debug
- assembly code pretty-printer
what to remember from this course?
understanding programming languages is fundamental

• to good programming
  • have a precise execution model in mind
  • pick up the good abstractions

• to research in Computer Science
  • design new languages
  • design tools
among other things, we have explained

- the stack
- parameter passing modes
- objects
- closures
compilers involve
- many different techniques
- several phases, often independent

most of these techniques can be reused in contexts other than assembly code production, such as
- computational linguistics
- computer-assisted proofs
- database queries
many other things we haven’t explored by lack of time

module systems
common sub-expressions
program transformation
abstract interpretation
alias analysis
loop unrolling
interprocedural optimization
peephole optimization
pipeline
memory cache
logic programming
just-in-time compilation
instruction scheduling
etc.
• project due **Sunday March 19**, 6pm
  • graded on the basis of report reading + code review

• exam Monday March 13 2pm–5pm
  • lecture notes are allowed
  • archives (2018, 2019, 2021, 2022) available on the course web site