École Polytechnique

CSC_52064 – Compilation

Jean-Christophe Filliâtre

optimizing compiler (1/2)

goal for the next two lectures: implementing an **optimizing compiler**

we intend to use x86-64 in the best possible way, notably

- its 16 registers
  - to pass parameters and to return results
  - for intermediate computations
- its instructions
  - such as the ability to add a constant to a register

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

emitting optimized code in a single pass is doomed to failure

we decompose code production into **several phases**

1. instruction selection
2. RTL (*Register Transfer Language*)
3. ERTL (*Explicit Register Transfer Language*)
4. LTL (*Location Transfer Language*)
5. linearization

(we follow the architecture of the CompCert compiler by Xavier Leroy;
see `http://compcert.inria.fr/`)

the starting point the abstract syntax tree output by the type checker

$$
\begin{tikzcd}
\texttt{Ttree} \arrow[d, "\texttt{Is}"] \\
\texttt{Istree} \arrow[d, "\texttt{Rtl}"] \\
\texttt{Rtltree} \arrow[d, "\texttt{Ertl}"] \\
\texttt{Ertltree} \arrow[d, "\texttt{Ltl}"] \\
\texttt{Ltltree} \arrow[d, "\texttt{Lin}"] \\
\texttt{X86\_64}
\end{tikzcd}
$$

this compiler architecture is independent of the programming paradigm (imperative, functional, object oriented, etc.)

it is illustrated on a small fragment of C a small fragment of **C** with

- integers (type `int`)
- heap-allocated structures, only pointers to structures, no pointer arithmetic
- functions
- library functions `putchar` and `malloc`

to keep it simple, we assume 64-bit signed integers for values of type `int` (unusual, but standard compliant) so that integers and pointers have the same size

$$
\begin{array}{rcl}
E & \rightarrow & n \\
 & | & L \\
 & | & L \;\texttt{=}\; E \\
 & | & E \; op \; E \mid \texttt{-}\; E \mid \texttt{!}\; E \\
 & | & x\texttt{(}E, \ldots, E\texttt{)} \\
 & | & \texttt{sizeof(struct}\; x\texttt{)} \\
L & \rightarrow & x \\
 & | & E\texttt{->}x \\
op & \rightarrow & \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
 & | & \texttt{\&\&} \mid \texttt{||} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \\
D & \rightarrow & T \; x(T \; x, \ldots, T \; x) \; B \\
 & | & \texttt{struct}\; x \; \{V \ldots V\}\texttt{;}
\end{array}
$$

$$
\begin{array}{rcl}
S & \rightarrow & \texttt{;} \\
 & | & E\texttt{;} \\
 & | & \texttt{if}\; (E) \; S \; \texttt{else} \; S \\
 & | & \texttt{while}\; (E) \; S \\
 & | & \texttt{return}\; E\texttt{;} \\
 & | & B \\
B & \rightarrow & \{ \; V \ldots V \; S \ldots S \; \} \\
V & \rightarrow & \texttt{int}\; x, \ldots, x\texttt{;} \\
 & | & \texttt{struct}\; x \; \texttt{*}x, \ldots, \texttt{*}x\texttt{;} \\
T & \rightarrow & \texttt{int} \mid \texttt{struct}\; x \; \texttt{*} \\
P & \rightarrow & D \ldots D
\end{array}
$$

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

```c
struct list { int val; struct list *next; };

int print(struct list *l) {
  while (l) {
    putchar(l->val);
    l = l->next;
  }
  return 0;
}
```

we assume that type checking is done

in particular, we know the type of any sub-expression

note: for mini-C, types are not useful for code generation; yet,

- type checking ensures some form of safety e.g. we do not confuse an integer and a pointer
- for a larger fragment of C, types would be needed e.g. to select signed vs unsigned operations, to perform pointer arithmetic, etc.

the first phase is **instruction selection**

goal:

- replace C arithmetic operations with x86-64 operations
- replace structure field access with explicit memory access

naively, we could simply translate each C arithmetic operation with the corresponding x86-64 operation

however, x86-64 provides us with better instructions in some cases, notably

- addition of a register and a constant
- bit shifting to the left or to the right, corresponding to a multiplication or a division by a power of 2
- comparison of a register and a constant

beside, it is advisable to perform as much evaluation as possible during compilation (partial evaluation)

examples: in some cases, we can simplify

- $(1 + e_1) + (2 + e_2)$ into $e_1 + e_2 + 3$
- $e + 1 < 10$ into $e < 9$
- $!(e_1 < e_2)$ into $e_1 \geq e_2$
- $0 \times e$ into 0

important: the semantics must be preserved if some left/right evaluation order would be specified, we could simplify $(0 - e_1) + e_2$ into $e_2 - e_1$ only when e_1 and e_2 do not interfere

for instance if e_1 and e_2 are pure i.e. without side effect

with C, the evaluation order is not specified, so we can make the simplification with **unsigned** C arithmetic, we could not replace $e + 1 < 10$ with $e < 9$ since $e + 1$ may be 0 by arithmetic overflow (the standard says that unsigned arithmetic wraps around)

if e is the greatest integer, $e + 1 < 10$ holds but $e < 9$ does not

with **signed** arithmetic, however, arithmetic overflow is an undefined behavior (meaning that the compiler may choose **any** behavior)

consequently, we can turn $e + 1 < 10$ into $e < 9$ with type `int` we can replace $0 \times e$ with 0 only if expression e has no side effect

since our expressions may involve function calls,
checking whether e has no effect is not decidable

but we can over-approximate the absence of effect

$$
\begin{array}{rcl}
pure(n) & = & true \\
pure(x) & = & true \\
pure(e_1 \texttt{ + } e_2) & = & pure(e_1) \wedge pure(e_2) \\
 & \vdots & \\
pure(e_1 \texttt{ = } e_2) & = & false \\
pure(f(e_1, \dots, e_n)) & = & false \quad \text{(we don't know)}
\end{array}
$$

to implement partial evaluation, we can use **smart constructors**

a smart constructor behaves like a syntax tree constructor but it performs some simplifications on the fly

example: for addition, we introduce a smart constructor such as

```
val mk_add: expr -> expr -> expr      (* OCaml *)
```

```
Expr mkAdd(Expr e1, Expr e2)          // Java
```

here are some simplifications for addition:

$$
\begin{array}{rcl}
\texttt{mkAdd}(n_1, n_2) & = & n_1 + n_2 \\
\texttt{mkAdd}(0, e) & = & e \\
\texttt{mkAdd}(e, 0) & = & e \\
\texttt{mkAdd}(n, e) & = & \texttt{addi}\ n\ e \\
\texttt{mkAdd}(e, n) & = & \texttt{addi}\ n\ e \\
\texttt{mkAdd}(\texttt{addi}\ n_1\ e, n_2) & = & \texttt{mkAdd}(n_1 + n_2, e) \\
\texttt{mkAdd}(e_1, e_2) & = & \texttt{add}\ e_1\ e_2 \quad \text{otherwise}
\end{array}
$$

of course, the smart constructor must terminate

one has to figure out a positive measure over expressions that strictly decreases at each recursive call of the smart constructor instruction selection is where we can make a good use of `lea` (see lab 1)

example: $9x$ can be computed using `lea(`x`,`x`,8)` instruction selection is then a recursive process over the expressions

$$
\begin{array}{rcl}
IS(e_1\texttt{+}e_2) & = & \texttt{mkAdd}(IS(e_1), IS(e_2)) \\
IS(e_1\texttt{-}e_2) & = & \texttt{mkSub}(IS(e_1), IS(e_2)) \\
IS(\texttt{!}\,e_1) & = & \texttt{mkNot}(IS(e_1)) \\
IS(\texttt{-}e_1) & = & \texttt{mkSub}(0, IS(e_1)) \\
 & \vdots &
\end{array}
$$

and a morphism for the other constructs

instruction selection also introduces explicit memory access, written `load` and `store`

a memory address is given by an expression together with a constant offset (so that we make good use of indirect addressing) in our case, structure fields reads and assignments are turned into memory accesses

we have a simple schema where each field is exactly one word long (since type `int` is assumed to be 64 bits)

so

$$\begin{array}{rcl} IS(e_1\texttt{->}x) & = & \texttt{load}\ IS(e_1)\ (n \times \texttt{wordsize}) \\ IS(e_1\texttt{->}x\ \texttt{=}\ e_2) & = & \texttt{store}\ IS(e_1)\ (n \times \texttt{wordsize})\ IS(e_2) \end{array}$$

where n is the index for field x in the structure and $\texttt{wordsize} = 8$ (64 bits) with the following structure

```
struct S { int a; int b; };
```

the instruction selection for expression

```
p->a = p->b + 2
```

is

```
store p 0 (addi 2 (load p 8))
```

with more types, we would have variable-length fields

the compiler then inserts **padding** bytes to align fields according to their types

example: with the structure

```
struct S { int a; char b; int *c; char d; };
```

we get a total of 24 bytes and the following layout

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
a				b	...			c								d	...						

instruction selection is a morphism over statements (`if`, `while`, etc.)

for functions, we erase types (not needed anymore) and we gather all variables at the function level (type checking made all variables distinct)

```c
struct list {
  int val;
  struct list *next; };

int print(struct list *l) {
  struct list *p;
  p = l;
  while (p) {
    int c;
    c = p->val;
    putchar(c);
    p = p->next;
  }
  return 0;
}
```

```
// no need for type list
// anymore


print(l) {
  locals p, c;
  p = l;
  while (p) {

    c = load p 0;
    putchar(c);
    p = load p 8;
  }
  return 0;
}
```

the classic factorial

```
int fact(int x) {

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

```
fact(x) {
  locals:
  if (setle x $1) return 1;
  return imul x fact(addi $-1 x);
}
```

the next phase transforms the code into **RTL** (*Register Transfer Language*)

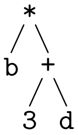
goal:

- get rid of the tree structure of expressions and statements, in favor of a **control-flow graph** (CFG), to ease further phases; in particular, we make no distinction between expressions and statements anymore

- introduce **pseudo-registers** to hold function parameters and intermediate computations; there are infinitely many pseudo-registers, that will later be either x86-64 registers or stack locations let us consider the C expression

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

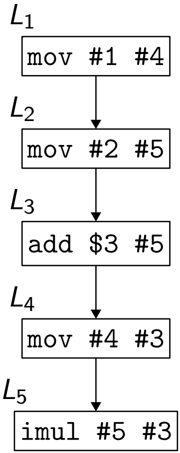
that is the syntax tree



let us assume that `b` and `d` are in pseudo-registers #1 and #2

and the final value in #3

then we build a CFG such as



the CFG is a map from labels (program points) to RTL instructions

conversely, each RTL instruction lists the labels of the next instructions

for instance, the RTL instruction

$$\texttt{mov}\ n\ r\ \ \rightarrow L$$

means "load constant n into pseudo-register r and transfer control to label L"

`mov` $n\ r \to L$	
`load` $n(r_1)\ r_2 \to L$	
`store` $r_1\ n(r_2) \to L$	
`unop` $op\ r \to L$	unary operation (`neg`, etc.)
`binop` $op\ r_1\ r_2 \to L$	binary operation (`add`, `mov`, etc.)
`ubranch` $br\ r \to L_1, L_2$	unary branching (`jz`, etc.)
`bbranch` $br\ r_1\ r_2 \to L_1, L_2$	binary branching (`jle`, etc.)
`call` $r \leftarrow f(r_1, \dots, r_n) \to L$	
`goto` $\to L$	

we build a separate CFG for each function, with its own pseudo-registers (**intra**procedural analysis)

we build the CFG from bottom to top, which means we always know the label of the continuation (the next instructions) to translate an expression, we provide

- a pseudo-register r_d to receive its value
- a label L_d corresponding to the continuation

we return the label of the entry point for the evaluation of the expression

$$RTL(e, r_d, L_d)$$

the translation is pretty straightforward

$$
\begin{array}{rcll}
RTL(n, r_d, L_d) & = & \text{add } L_1 : \texttt{mov } n\ r_d\ \ \to L_d & \text{with } L_1 \text{ fresh} \\
 & & \text{return } L_1 & \\[1em]
RTL(\texttt{add } e_1\ e_2, r_d, L_d) & = & \text{add } L_3 : \texttt{add } r_2\ r_d\ \ \to L_d & \text{with } r_2, L_3 \text{ fresh} \\
 & & L_2 \leftarrow RTL(e_2, r_2, L_3) & \\
 & & L_1 \leftarrow RTL(e_1, r_d, L_2) & \\
 & & \text{return } L_1 &
\end{array}
$$

etc.

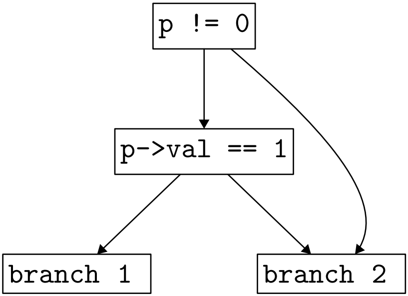
(read the code from bottom to top)

for local variables, we set up a table where each variable is mapped to a fresh pseudo-register

then reading or writing a local variable is a `mov` instruction (one of the RTL binary operations) to translate C operations `&&` and `||`, as well as `if` and `while` statements, we use RTL **branching** instructions

example:

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



(the four blocks are sub-graphs)

to translate a condition, we provide two labels

- a label L_t corresponding to the continuation if the condition holds
- a label L_f when it does not hold

we return the label of the entry point for the evaluation of the condition

$$RTL_c(e,\, L_t,\, L_f)$$

$$
\begin{array}{rcl}
RTL_c(e_1 \mathbin{\&\&} e_2, L_t, L_f) & = & RTL_c(e_1,\ RTL_c(e_2, L_t, L_f),\ L_f) \\[1em]
RTL_c(e_1 \mathbin{||}, e_2, L_t, L_f) & = & RTL_c(e_1,\ L_t,\ RTL_c(e_2, L_t, L_f)) \\[1em]
RTL_c(e_1 \mathbin{<=} e_2, L_t, L_f) & = & \text{add } L_3 : \texttt{bbranch jle } r_2\ r_1\ \ \to L_t, L_f \\
 & & L_2 \leftarrow RTL(e_2, r_2, L_3) \\
 & & L_1 \leftarrow RTL(e_1, r_1, L_2) \\
 & & \text{return } L_1 \\[1em]
RTL_c(e, L_t, L_f) & = & \text{add } L_2 : \texttt{ubranch jz } r\ \ \to L_f, L_t \\
 & & L_1 \leftarrow RTL(e, r, L_2) \\
 & & \text{return } L_1
\end{array}
$$

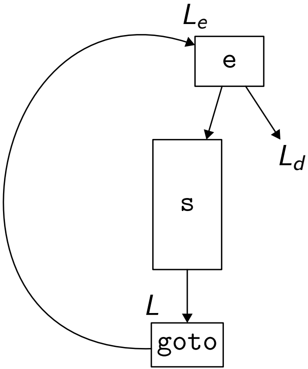
(of course, we can handle more particular cases) to translate `return`, we provide a pseudo-register r_{ret} to receive the function result and a label L_{ret} corresponding to the function exit

$$
\begin{array}{rcl}
RTL(\texttt{;}, L_d) & = & \text{return } L_d \\
RTL(\texttt{return } e\texttt{;}, L_d) & = & RTL(e, r_{ret}, L_{ret}) \\
RTL(\texttt{if}\,\texttt{(}e\texttt{)}\,s_1 \texttt{ else } s_2, L_d) & = & RTL_c(e,\ RTL(s_1, L_d),\ RTL(s_2, L_d))
\end{array}
$$

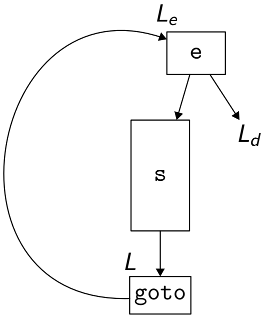
etc.

for a `while` loop, we have to build a cycle in the CFG

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



$$
\begin{array}{rcl}
RTL(\texttt{while}(e)s, L_d) & = & L_e \leftarrow RTL_c(e, \ , RTL(s, L), \ L_d) \\
& & \text{add } L : \text{goto } L_e \\
& & \text{return } L_e
\end{array}
$$



the formal parameters of a function, and its result, now are pseudo-registers

```
#3 f(#1, #2) { ... }
```

as well as actual parameters and result in a call

```
call #4 <- f(#5, #6)
```

translating a function involves the following steps:

1. we allocate fresh pseudo-registers for its parameters, its result, and its local variables
2. we start with an empty graph
3. we pick a fresh label for the function exit
4. we translate the function body to RTL code, and the output is the entry label in the CFG

with the factorial function

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

we get

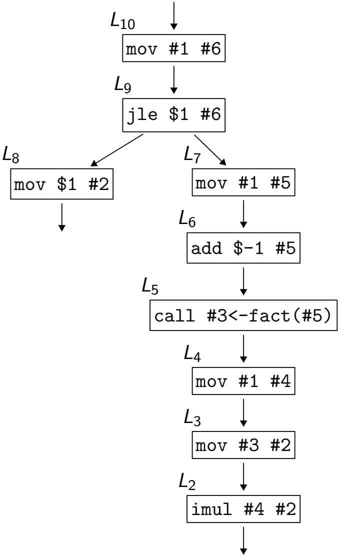
```
#2 fact(#1)
  entry : L10
  exit  : L1
  locals:
  L10: mov #1 #6  --> L9
  L9 : jle $1 #6  --> L8, L7
  L8 : mov $1 #2  --> L1
  L7 : mov #1 #5        --> L6
  L6 : add $-1 #5       --> L5
  L5 : call #3<-fact(#5)--> L4
  L4 : mov #1 #4        --> L3
  L3 : mov #3 #2        --> L2
  L2 : imul #4 #2       --> L1
```

(the graph is printed arbitrarily)

with the factorial function

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

we get



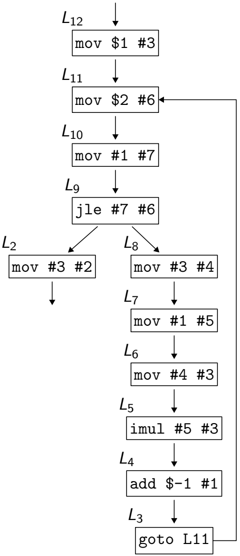
with a loop

```
int loop(int x) {
  int r;
  r = 1;
  while (2 <= x) {
    r = r * x;
    x = x - 1;
  }
  return r;
}
```

```
#2 loop(#1)
  entry : L12
  exit  : L1
  locals: #3
  L12: mov $1 #3  --> L11
  L11: mov $2 #6  --> L10
  L10: mov #1 #7  --> L9
  L9 : jle #7 #6  --> L8, L2
  L8 : mov #3 #4  --> L7
  L7 : mov #1 #5  --> L6
  L6 : mov #4 #3  --> L5
  L5 : imul #5 #3 --> L4
  L4 : add $-1 #1 --> L3
  L3 : goto L11
  L2 : mov #3 #2  --> L1
```

with a loop

```
int loop(int x) {
  int r;
  r = 1;
  while (2 <= x) {
    r = r * x;
    x = x - 1;
  }
  return r;
}
```



the third phase turns RTL into **ERTL** (Explicit Register Transfer Language)

goal: make **calling conventions** explicit, namely here

- the first six parameters are passed in `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` and the next ones on the stack
- the result is returned in `%rax`
- in particular, `putchar` and `malloc` are library functions with a parameter in `%rdi` and a result in `%rax`
- the division `idivq` requires dividend and quotient in `%rax`
- callee-saved registers must be preserved by the callee (`%rbx`, `%r12`, `%r13`, `%r14`, `%r15`, `%rbp`)

the stack frame is as follows:

	⋮
	param. n
	⋮
	param. 7
	return addr.
`%rbp` $\rightarrow$	saved `%rbp`
	local 1
	⋮
`%rsp` $\rightarrow$	local m
	⋮

the m local variables area will hold all the pseudo-registers that could not be allocated to physical registers; register allocation (phase 4) will determine the value of m

in ERTL, we have those same instructions as in RTL:

`mov` $n\ r \to L$	
`load` $n(r_1)\ r_2 \to L$	
`store` $r_1\ n(r_2) \to L$	
`unop` $op\ r \to L$	unary operation (`neg`, etc.)
`binop` $op\ r_1\ r_2 \to L$	binary operation (`add`, `mov`, etc.)
`ubranch` $br\ r \to L_1, L_2$	unary branching (`jz`, etc.)
`bbranch` $br\ r_1\ r_2 \to L_1, L_2$	binary branching (`jle`, etc.)
`goto` $\to L$	

in RTL, we had

$$\texttt{call}\ r \leftarrow f(r_1, \dots, r_n) \rightarrow L$$

in ERTL, we now have

$$\texttt{call}\ f(k) \rightarrow L$$

i.e. we are only left with the name of the function to call, since new instructions will be inserted to load parameters into registers and stack, and to get the result from `%rax`

we only keep the number k of parameters passed into registers (to be used in phase 4) finally, we have **new** instructions:

`alloc_frame` $\rightarrow L$	allocate the stack frame
`delete_frame` $\rightarrow L$	delete the stack frame
	(note: we do not know its size yet)
`get_param` $n\ r \rightarrow L$	access a parameter on stack (with n(`%rbp`))
`push_param` $r \rightarrow L$	push the value of r
`return`	explicit return instruction

we do not change the structure of the control-flow graph; we simply **insert new instructions**

- at the beginning of each function, to
  - allocate the stack frame
  - save the callee-saved registers
  - copy the parameters into the corresponding pseudo-registers
- at the end of each function, to
  - copy the pseudo-register holding the result into `%rax`
  - restore the callee-saved registers
  - delete the stack frame
  - execute `return`
- around each function call, to
  - copy the pseudo-registers holding the parameters into `%rdi`, ... and one the stack before the call
  - copy `%rax` into the pseudo-register holding the result after the call
  - pop the parameters, if any

we translate each RTL instruction to one/several ERTL instructions

mostly the identity operation, except for calls and division dividend and quotient are in %rax

the RTL instruction

$$L_1 : \texttt{binop div } r_1\ r_2 \to L$$

becomes three ERTL instructions

$$\begin{array}{ll} L_1: & \texttt{binop mov } r_2\ \texttt{\%rax} \to L_2 \\ L_2: & \texttt{binop div } r_1\ \texttt{\%rax} \to L_3 \\ L_3: & \texttt{binop mov \%rax } r_2 \to L \end{array}$$

where L_2 and L_3 are fresh labels

(beware of the direction: here we divide r_2 by r_1)

we translate the RTL instruction

$$L_1 : \ \texttt{call}\ r \leftarrow f(r_1, \ldots, r_n) \rightarrow L$$

into a sequence of ERTL instructions

1. copy $\min(n, 6)$ parameters $r_1, r_2, \ldots$ into `%rdi`,`%rsi`,...
2. if $n > 6$, pass other parameters on the stack with `push_param`
3. execute `call` $f(\min(n, 6))$
4. copy `%rax` into r
5. if $n > 6$, pop $8 \times (n - 6)$ bytes

that starts at the same label L_1 and transfers the control at the end to the same label L

the RTL code

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

is translated into the ERTL code

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

RTL

```
r f(r,..,r)
  locals : { r, ... }
  entry  : L
  exit   : L
  cfg    : ...
```

ERTL

```
f(n)
  locals : { r, ... }
  entry  : L

  cfg    : ...
```

for each callee-saved register, we allocate a fresh pseudo-register to save it, that we add to the local variables of the function

note: for the moment, we do not try to figure out which callee-saved registers will be used by the function at the function entry, we

- allocate the stack frame with `alloc_frame`
- save the callee-saved registers
- copy the parameters into their pseudo-registers

RTL

```
#2 fact(#1)
  entry : L10
  exit  : L1
  locals:




  L10: mov #1 #6     --> L9
  ...
```

ERTL

```
fact(1)
  entry : L17

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

to make things simpler, we here assume that callee-saved registers are limited to %rbx and %r12 (in practice, we also have %r13, %r14, %r15) at function exit, we

- copy the pseudo-register holding the result into `%rax`
- restore the saved registers
- delete the stack frame

RTL

```
#2 fact(#1)
  entry : L10
  exit  : L1
  locals:
  ...
  L8 : mov $1 #2     --> L1
  ...
  L2 : imul #4 #2    --> L1
```

ERTL

```
fact(1)
  entry : L17

  locals: #7, #8
  ...
  L8 : mov $1 #2     --> L1
  ...
  L2 : imul #4 #2    --> L1
  L1 : mov #2 %rax   --> L21
  L21: mov #7 %rbx   --> L20
  L20: mov #8 %r12   --> L19
  L19: delete_frame --> L18
  L18: return
```

altogether, we get the following ERTL code:

```
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
```

this is far from being what we think is a good x86-64 code for the factorial

at this point, we have to understand that

- register allocation (phase 4) will try to match physical registers to pseudo-registers to minimize the use of the stack and the number of `mov`

  if for instance we map `#8` to `%r12`, we remove the two instructions at `L15` and `L20`

- the code is not linearized yet (the graph is simply printed in some arbitrary order); this will be done in phase 5, where we will try to minimize jumps if we intend to optimize tail calls, it has to be done during the RTL to ERTL translation

indeed, the ERTL instructions will differ, and this change influences the next phase (register allocation) there is a difficulty, however, if the called function in a tail call does not have the same number of stack parameters or of local variables, since the stack frame has to be modified

at least two solutions

- limit tail call optimization to cases where the stack frame has the **same layout**; this is the case for recursive calls!
- the caller patches the stack frame and transfers the control **after** the instructions that allocate the stack frame

the next phase translates ERTL to **LTL** (Location Transfer Language)

the goal is to get rid of pseudo-registers, replacing them with

- physical registers preferably
- stack locations otherwise

this is called **register allocation**

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 **graph coloring**
   - it maps pseudo-registers to physical registers or stack locations

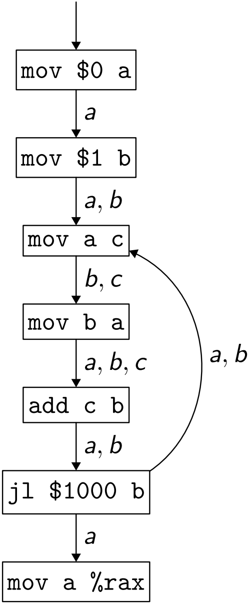
in the following, a *variable* stands for a pseudo-register or a physical register

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 live variables are drawn on edges

```
    mov $0 a
    mov $1 b
L1: mov a c
    mov b a
    add c b
    jl  $1000 b L1
    mov a %rax
```



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

- $\mathit{def}(\ell)$ *for the set of variables defined by this instruction,*
- $\mathit{use}(\ell)$ *for the set of variables used by this instruction.*

example: for the instruction `add` r_1 r_2 we have

$$\mathit{def}(\ell) = \{r_2\} \quad \text{and} \quad \mathit{use}(\ell) = \{r_1, r_2\}$$

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

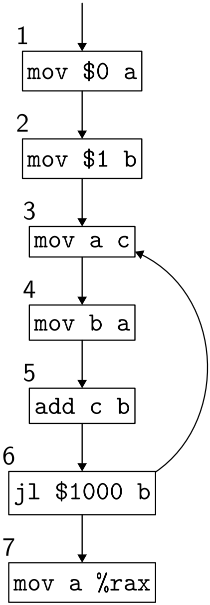
- *in*(ℓ) *for the set of live variables on the set of incoming edges to* ℓ,
- *out*(ℓ) *for the set of live variables on the set of outcoming edges from* ℓ.

the equations defining $in(\ell)$ and $out(\ell)$ are the following

$$\left\{ \begin{array}{rcl} in(\ell) & = & use(\ell) \cup (out(\ell) \backslash def(\ell)) \\ out(\ell) & = & \bigcup_{s \in succ(\ell)} in(s) \end{array} \right.$$

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 3)



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

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

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

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

$$\left\{\begin{array}{rcl} out(\ell) & = & \bigcup_{s\in succ(\ell)} in(s) \\ in(\ell) & = & use(\ell) \cup (out(\ell) \backslash def(\ell)) \end{array}\right.$$

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 the sets $def(\ell)$ and $use(\ell)$ is straightforward for most instructions

examples:

	def	*use*
`mov` n r	$\{r\}$	$\emptyset$
`mov` r_1 r_2	$\{r_2\}$	$\{r_1\}$
`unop` op r	$\{r\}$	$\{r\}$
`goto`	$\emptyset$	$\emptyset$
...		

this is more subtle for function calls

for a call, we express that any caller-saved register may be erased by the call

	def	*use*
`call` $f(k)$	*caller-saved*	the first k registers of `%rdi`,`%rsi`,...,`%r9`

last, for `return`, we express that `%rax` and all callee-saved registers may be used

	def	*use*
`return`	$\emptyset$	$\{$`%rax`$\} \cup$ *callee-saved*

this was the ERTL code for `fact`

```
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
```

```
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
```

- lab 7
  - mini-Java continued
- lecture 8
  - optimizing compiler 2/2