Interpretation and Compilation of Language

Master Programme in Computer Science

Mário Pereira mjp.pereira@fct.unl.pt

Nova School of Science and Technology, Portugal

May 20, 2025

Lecture 11

based on lectures by Jean-Christophe Filliâre and LØon Gondelman
previous editions by Joªo Costa Seco, Luís Caires, and Bernardo Toninho

Today: Optimizing Compiler, part 1

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

Goal for the next two lectures: writing 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
```

Compiler Phases = Many AST

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

$$
\begin{array}{c}
\text{Ttree} \\
\downarrow \text{Is} \\
\text{Istree} \\
\downarrow \text{Rtl} \\
\text{Rtltree} \\
\downarrow \text{Ertl} \\
\text{Ertltree} \\
\downarrow \text{Ltl} \\
\text{Ltltree} \\
\downarrow \text{Lin} \\
\text{Mips}
\end{array}
$$

Remark

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 i
(unusual, but standard compliant) so that integers and pointers have th
same size.

```
E   !  n
       j  L
       j  L = E
       j  E op E j - E j ! E
       j  x(E;:::;E)
       j  sizeof(struct x)

L   !  x
       j  E->x

op  !  == j != j < j <= j > j >=
       j  && j || j + j - j * j /

D   !  T x(T x;:::;T x) B
       j  struct x {V :::V};
```

```
S   !  ;
       j  E;
       j  if (E) S else S
       j  while (E) S
       j  return E;
       j  B

B   !  { V :::V  S:::S }

V   !  int x;:::;x;
       j  struct x *x;:::;*x;

T   !  int j struct x *

P   !  D:::D
```

```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 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.

Phase 1: Instruction Selectio

The rst phase is instruction selection.

Goal:

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

Arithmetic Operations

Naively, we can 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

Besides, it is advisable to perform as much evaluation as possible durin
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

Crucial: 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 over ow (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 over ow is an unde ned 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:

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

To implement partial evaluation, we can use smart constructors.

A smart constructor behaves like a syntax tree constructor but it perforr
some simpli cations on the y.

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

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

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

Smart Constructor for Addition

Instruction selection is thus a recursive process over the expressions:

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

and a direct translation for the other constructs.

Instruction selection also introduces explicit memory access, written loa
and store.

A memory address is given by an expression together with a constant o
(so that we make good use of indirect addressing).

In our case, structure elds reads and assignments are turned into mem accesses.

We have a simple schema where each eld is exactly one word long (since type int is assumed to be 64 bits).

So

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

where n is the index for eld x in the structure and 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))
```

Instruction selection is a direct translation over statements (if, while, etc.).

For functions, we erase types (not needed anymore) and we gather all variables at the function level.

```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);
}
```

Phase 2: RTL

The next phase transforms the code to the language RTL (Register Transfer Language).

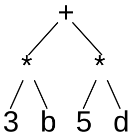
Goal:

- get rid of the tree structure of expressions and statements, in favor a control- ow 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 in nitely many pseudo-registe that will later be either x86-64 registers or stack locations

Let us consider the C expression

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

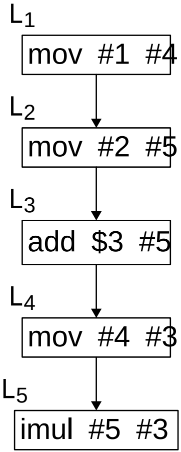
that is the syntax tree



Let us assume that b and d are in pseudo-registers #1 and #2.

And the nal value in #3.

Then we build a CFG such as



mov n r ! L	
load n(r_1) r_2 ! L	
store r_1 n(r_2) ! L	
unop op r ! L	unary operation (neg, etc.)
binop op r_1 r_2 ! L	binary operation (add, mov, etc.)
ubranch br r ! L_1;L_2	unary branching (jz, etc.)
bbranch br r_1 r_2 ! L_1;L_2	binary branching (jle, etc.)
call r f (r_1;:::;r_n) ! L	
goto ! L	

We build a separate CFG for each function, with its own pseudo-registe
(intraprocedural analysis).

We build the CFG from bottom to top, which means we always know th
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 expressio

$$RTL(e; r_d; L_d)$$

The translation is pretty straightforward.

$$\begin{array}{rcll}
RTL(n; r_d; L_d) & = & \text{add } L_1 : \text{mov } n\ r_d\ !\ L_d & \text{with } L_1 \text{ fresh} \\
 & & \text{return } L_1 & \\
 & & & \\
RTL(e_1 + e_2; r_d; L_d) & = & \text{add } L_3 : \text{add } r_2\ r_d\ !\ L_d & \text{with } r_2; L_3 \text{ fresh} \\
 & & L_2 \quad RTL(e_2; r_2; L_3) & \\
 & & L_1 \quad RTL(e_1; r_d; L_2) & \\
 & & \text{return } L_1 &
\end{array}$$

etc.

(Read the code from bottom to top).

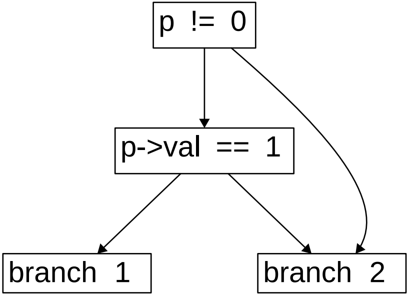
For local variables, we set up a table where each variable is mapped to fresh pseudo-register.

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

$$RTL_c(e; L_t; L_f)$$

$$
\begin{aligned}
RTL_c(e_1 \,\&\&\, e_2; L_t; L_f) &= RTL_c(e_1;\ RTL_c(e_2; L_t; L_f);\ L_f) \\
RTL_c(e_1 \,||,\, e_2; L_t; L_f) &= RTL_c(e_1;\ L_t;\ RTL_c(e_2; L_t; L_f)) \\
RTL_c(e_1 <= e_2; L_t; L_f) &= \text{add } L_3 : \text{bbranch jle } r_2\ r_1\ !\ L_t; L_f \\
&\quad\ L_2\ \ RTL(e_2; r_2; L_3) \\
&\quad\ L_1\ \ RTL(e_1; r_1; L_2) \\
&\quad\ \text{return } L_1 \\
RTL_c(e; L_t; L_f) &= \text{add } L_2 : \text{ubranch jz } r\ !\ L_f; L_t \\
&\quad\ L_1\ \ RTL(e; r; L_2) \\
&\quad\ \text{return } L_1
\end{aligned}
$$

(Of course, we can handle more particular cases).

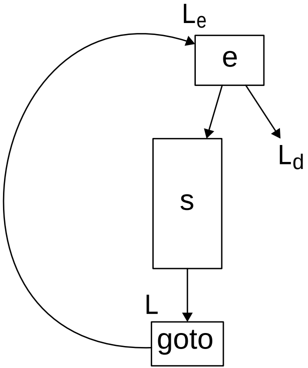
To translate return, we provide a pseudo-register r_{ret} to receive the function result and a label L_{ret} corresponding to the function exit

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

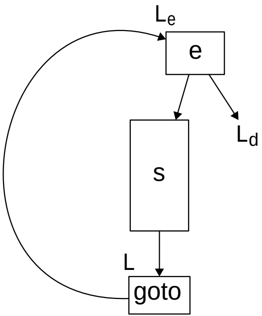
etc.

For a while loop, we have to build a cycle in the CFG.

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



$$\mathrm{RTL}(\mathrm{while}(e)s; L_d) \quad = \quad L_e \quad \mathrm{RTL}_c(e; \ ; \mathrm{RTL}(s; L); \ L_d)$$
$$\text{add } L : \text{goto } L_e$$
$$\text{return } L_e$$



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 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

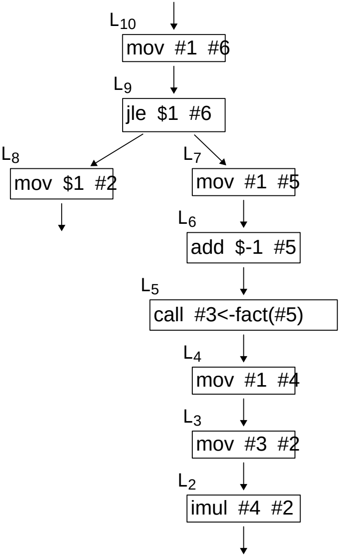
```
#2 fact(#1)
  entry : L10
  exit  : L1
  locals:
  L10: mov #1 #6   --> L9
  L9 : jle $1 #6   --> L8, L7
  L8 : mov $1 #2   --> L1
  L7 : mov #1 #5   --> L6
  L6 : add $-1 #5  --> L5
  L5 : call #3<-fact(#5)--> L4
  L4 : mov #1 #4   --> L3
  L3 : mov #3 #2   --> L2
  L2 : imul #4 #2  --> L1
```

(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



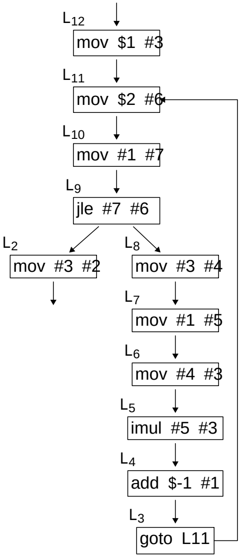
With a loop

```
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;
}
```



Phase 3: ERTL

The third phase turns RTL into ERTL (Explicit Register Transfer

The stack frame is as follows:

	⋮
	param. n
	⋮
	param. 7
	return addr.
%rbp !	saved %rbp
	local 1
	⋮
%rsp !	local m
	⋮

The m local variables area will hold all the pseudo-registers that could 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 ! L
load n(r_1) r_2 ! L
store r_1 n(r_2) ! L
unop op r ! L — unary operation (neg, etc.)
binop op r_1 r_2 ! L — binary operation (add, mov, etc.)
ubranch br r ! L_1;L_2 — unary branching (jz, etc.)
bbranch br r_1 r_2 ! L_1;L_2 — binary branching (jle, etc.)
goto ! L

In RTL, we had

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

In ERTL, we now have

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

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

We do not change the structure of the control- ow graph; we simply inse
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 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

$$L_1 : \text{ binop div } r_1 \; r_2 \; ! \; L$$

becomes three ERTL instructions

$$\begin{array}{l} L_1 : \text{ binop mov } r_2 \text{ \%rax } ! \; L_2 \\ L_2 : \text{ binop div } r_1 \text{ \%rax } ! \; L_3 \\ L_3 : \text{ binop mov \%rax } r_2 \; ! \; L \end{array}$$

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

$$L_1: \text{ call } r \leftarrow f(r_1, \dots, r_n) \to L$$

into a sequence of ERTL instructions

1. copy $\min(n, 6)$ parameters $r_1, r_2, \dots$ 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	ERTL
r f(r,..,r) locals : { r, ... } entry : L exit : L cfg : ...	f(n) locals : { r, ... } entry : L cfg : ...

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

Note: for the moment, we do not try to gure out which callee-saved registers will be used by the function.

Function Entry

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
```

An Interesting Optimization

If we intend to optimize tail calls, it has to be done during the RTL to ERTL translation.

indeed, the ERTL instructions will di er, and this change in uences the next phase (register allocation)

There is a di culty, however, if the called function in a tail call does not have the same number of stack parameters or of local variables, since stack frame has to be modi ed.

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 th 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 t[he]
     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.

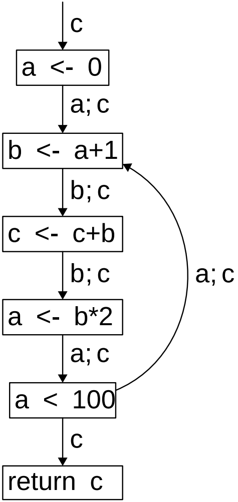
De nition (live variable)

Given a program point, a variable is said to be live if the value it contai 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 sour 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 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 add r_1 r_2 we have

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

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

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

De nition

For an instruction at label l in the control-fow 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.

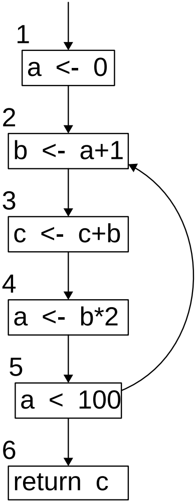
The equations defining $in(l)$ and $out(l)$ are the following

$$\left\{\begin{array}{lcl} in(l) & = & use(l) \cup (out(l) \setminus def(l)) \\ out(l) & = & \bigcup_{s \in succ(l)} in(s) \end{array}\right.$$

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 th
we can use Tarski's theorem (see lecture 4).

here we go again...



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

	use	def	in	out	in	out		in	out
1		a					...		a
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(l) changes, then we only need to redo the computation for th
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 the sets def(l) (definitions) and use(l) (uses) 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	$\emptyset$	$\emptyset$
...		

Computing def and use

This is more subtle for function calls.

For a call, we express that any caller-saved register may be erased by
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 ma
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
```

Liveness for `fact`

```
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
```