goal for the last three 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

\[
\text{add} \quad $3, \%rdi
\]
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/](http://compcert.inria.fr/))
the starting point the abstract syntax tree output by the type checker

```
Ttree
  ↓ Is
Istree
  ↓ Rtl
Rtltree
  ↓ Ertl
Ertltree
  ↓ Ltl
Ltltree
  ↓ Lin
X86_64
```
this compiler architecture is independent of the programming paradigm (imperative, function, object oriented, etc.)

it is illustrated on a small fragment of C (that of the project)
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 type `int` (unusual, but standard compliant) so that integers and pointers have the same size
mini-C (reminder)

\[
\begin{align*}
E & \rightarrow n \\
& \quad \mid L \\
& \quad \mid L = E \\
& \quad \mid E \ op \ E \mid - E \mid ! E \\
& \quad \mid x(E,\ldots,E) \\
& \quad \mid \text{sizeof} (\text{struct } x) \\
L & \rightarrow x \\
& \quad \mid E \rightarrow x \\
op & \rightarrow == \mid != \mid < \mid <= \mid > \mid >= \\
& \quad \mid && \mid || \mid + \mid - \mid * \mid / \\
D & \rightarrow T \ x(T \ x,\ldots,T \ x) \ B \\
& \quad \mid \text{struct } x \ \{V \ldots V\}; \\
S & \rightarrow ; \\
& \quad \mid E; \\
& \quad \mid \text{if } (E) \ S \ \text{else} \ S \\
& \quad \mid \text{while } (E) \ S \\
& \quad \mid \text{return } E; \\
B & \rightarrow \{ \ V \ldots V \ S \ldots S \ \} \\
V & \rightarrow \text{int } x,\ldots,x; \\
& \quad \mid \text{struct } x \ \ast x,\ldots,\ast x; \\
T & \rightarrow \text{int } | \ \text{struct } x \ \ast \\
P & \rightarrow D \ldots D
\end{align*}
\]
int fact(int x) {
    if (x <= 1) return 1;
    return x * fact(x-1);
}

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
phase 1: instruction selection

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

examples: 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

\[
\begin{align*}
pure(n) & = \text{true} \\
pure(x) & = \text{true} \\
pure(e_1 + e_2) & = pure(e_1) \land pure(e_2) \\
& \vdots \\
pure(e_1 = e_2) & = \text{false} \\
pure(f(e_1, \ldots, e_n)) & = \text{false} \quad (\text{we don't know})
\end{align*}
\]
smart constructors

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
```
smart constructor for addition

here are some simplifications for addition:

\[
\begin{align*}
mkAdd(n_1, n_2) & = n_1 + n_2 \\
mkAdd(0, e) & = e \\
mkAdd(e, 0) & = e \\
mkAdd(\text{add } n_1 e, n_2) & = \text{mkAdd}(n_1 + n_2, e) \\
mkAdd(n, e) & = \text{add } n \ e \\
mkAdd(e, n) & = \text{add } n \ e \\
mkAdd(e_1, e_2) & = \text{add } e_1 \ e_2 \ 	ext{otherwise}
\end{align*}
\]
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)
instruction selection is then a recursive process over the expressions

\[
\begin{align*}
    IS(e_1 + e_2) &= \text{mkAdd}(IS(e_1), IS(e_2)) \\
    IS(e_1 - e_2) &= \text{mkSub}(IS(e_1), IS(e_2)) \\
    IS(\neg e_1) &= \text{mkNot}(IS(e_1)) \\
    IS(-e_1) &= \text{mkSub}(0, IS(e_1)) \\
    \vdots
\end{align*}
\]

and a morphism for the other constructs
memory access

instruction selection also introduces explicit memory access, written \texttt{load} and \texttt{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 \texttt{int} is assumed to be 64 bits)

so

\[
\begin{align*}
\text{IS}(e_1->x) &= \text{load IS}(e_1)(n \times \text{wordsize}) \\
\text{IS}(e_1->x = e_2) &= \text{store IS}(e_1)(n \times \text{wordsize}) \text{ IS}(e_2)
\end{align*}
\]

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

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

the instruction selection for expression

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

is

```c
store p 0 (addi 2 (load p 8))
```
instruction selection is a morphism over statements (if, while, etc.)
as 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;
}
```

```c
// 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 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

\[ b \times (3 + d) \]

that is the syntax tree

![Syntax tree diagram]

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

\[
\begin{align*}
L_1 & : \text{mov #1 #4} \\
L_2 & : \text{mov #2 #5} \\
L_3 & : \text{add $3 #5} \\
L_4 & : \text{mov #4 #3} \\
L_5 & : \text{imul #5 #3}
\end{align*}
\]
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

\[
\text{mov } n \; r \; \rightarrow \; L
\]

means “load constant \( n \) int pseudo-register \( r \) and transfer control to label \( L \)”
RTL instructions

- `mov n r → L`
- `load n(r₁) r₂ → L`
- `store r₁ n(r₂) → L`
- `unop op r → L`
- `binop op r₁ r₂ → L`
- `ubranch br r → L₁, L₂`
- `bbranch br r₁ r₂ → L₁, L₂`
- `call r ← f(r₁, ..., rₙ) → L`
- `goto → L`

**unary operation** (neg, etc.)

**binary operation** (add, mov, etc.)

**unary branching** (jz, etc.)

**binary branching** (jle, etc.)
we build a separate CFG for each function, with its own pseudo-registers

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
the translation is pretty straightforward

\[
\text{RTL}(n, r_d, L_d) = \quad \text{add } L_1 : \text{mov } n \ r_d \to L_d \quad \text{with } L_1 \text{ fresh}
\]

\[
\text{return } L_1
\]

\[
\text{RTL}(e_1 + e_2, r_d, L_d) = \quad \text{add } L_3 : \text{add } r_2 \ r_d \to L_d \quad \text{with } r_2, L_3 \text{ fresh}
\]

\[
L_2 \leftarrow \text{RTL}(e_2, r_2, L_3)
\]

\[
L_1 \leftarrow \text{RTL}(e_1, r_d, L_2)
\]

\[
\text{return } L_1
\]

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 \texttt{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:

```c
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
translating a condition

\[
RTL_c(e_1 \&\& e_2, L_t, L_f) = RTL_c(e_1, RTL_c(e_2, L_t, L_f), L_f)
\]

\[
RTL_c(e_1 \| e_2, L_t, L_f) = RTL_c(e_1, L_t, RTL_c(e_2, L_t, L_f))
\]

\[
RTL_c(e_1 \leq e_2, L_t, L_f) = \text{add } L_3 : \text{bbranch jle } r_2 r_1 \rightarrow L_t, L_f \\
L_2 \leftarrow RTL(e_2, r_2, L_3) \\
L_1 \leftarrow RTL(e_1, r_1, L_2) \\
\text{return } L_1
\]

\[
RTL_c(e, L_t, L_f) = \text{add } L_2 : \text{ubranch jz } r \rightarrow L_f, L_t \\
L_1 \leftarrow RTL(e, r, L_2) \\
\text{return } L_1
\]

(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.

\[
\begin{align*}
RTL( ; , L_d) & = \text{return } L_d \\
RTL(\text{return } e ; , L_d) & = RTL(e, r_{ret}, L_{ret}) \\
RTL(\text{if}(e) s_1 \ \text{else} \ s_2 , L_d) & = RTL_c(e, RTL(s_1, L_d), RTL(s_2, L_d)) \\
\text{etc.}
\end{align*}
\]
for a `while` loop, we have to build a cycle in the CFG

```plaintext
while (e) {
    ...s... 
}
```
\[
RTL(\text{while}(e)s, L_d) = \ \text{Le} \leftarrow RTL_c(e, , RTL(s, L), L_d) \\
\text{add } L: \text{goto } \text{Le} \\
\text{return } \text{Le}
\]
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

#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

```c
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: #3 <- call fact(#5) --> L4
L4: mov #1 #4 --> L3
L3: mov #3 #2 --> L2
L2: imul #4 #2 --> L1
```

(the graph is printed arbitrarily)
• lab 6
  • translation to RTL

• lecture 7
  • code production (2/3)
lab 6

translation to RTL
since mini-C is quite simple, we can fuse instruction selection and translation to RTL into a single pass

beside, we can make a very naive instruction selection (one C operation \(\approx\) one x86-64 operation) and improve that later
some code is provided (for OCaml and Java)

- x86-64 operations
- RTL abstract syntax
- RTL interpreter to test
- RTL pretty-printer to debug
- `main.ml / Main.java` for the command line

`.mini-c --debug --interp-rtl test.c`
proceed *incrementally*, translating C constructs one at a time

the very first goal must be to translate

```c
int main() {
    return 42;
}
```

which must output something like that:

```
#1 main()
    entry : L2
    exit  : L1
    locals:
    L2: mov $42 #1  --> L1
```