1. first-class functions
2. tail call optimization
3. mini-C optimizing compiler continued
first-class functions
A key aspect of functional programming is **first-class functions**, which means that a function is a value like any other.

In particular, we can:

- receive a function as a parameter
- return a function as a result
- store a function in a data structure
- build new functions dynamically
the ability to pass functions as parameters already exists in languages such as Algol, Pascal, Ada, etc.

similarly, the notion of function pointers already exists (Fortran, C, C++, etc.)

but the notion of first-class functions is strictly more powerful

let us illustrate it with OCaml
let us consider this fragment of OCaml

\[
\begin{align*}
e & ::= c \\
   & \mid x \\
   & \mid \text{fun } x \rightarrow e \\
   & \mid e \ e \\
   & \mid \text{let [rec] } x = e \text{ in } e \\
   & \mid \text{if } e \text{ then } e \text{ else } e
\end{align*}
\]

\[
\begin{align*}
d & ::= \text{let [rec] } x = e
\end{align*}
\]

\[
\begin{align*}
p & ::= d \ldots d
\end{align*}
\]
functions can be nested

```
let sum n =
    let f x = x * x in
    let rec loop i =
        if i = n then 0 else f i + loop (i+1)
    in
    loop 0
```

the scoping is usual

(we write `let f x = x * x` for `let f = fun x -> x * x`)

Jean-Christophe Filliâtre
Higher-order functions

We can pass functions as parameters

```ocaml
let square f x = f (f x)
```

and return functions

```ocaml
let f x = if x < 0 then fun y -> y - x else fun y -> y + x
```

Here, the function returned by `f` uses `x` but the stack frame for `f` just disappeared!

So we cannot compile functions in the usual way.
the solution is to use a closure (en français, une fermeture)

this is a heap-allocated data structure (to survive function calls) containing

• a pointer to the code (of the function body)
• the values of the variables that may be needed by this code; this is called the environment

variables in the environment

what are the variables to be stored in the environment of the closure representing \( \text{fun } x \rightarrow e \) ?

these are the **free variables** of \( \text{fun } x \rightarrow e \)

the set \( \text{fv}(e) \) of the free variables of the expression \( e \) is computed as follows:

\[
\begin{align*}
\text{fv}(c) &= \emptyset \\
\text{fv}(x) &= \{x\} \\
\text{fv(\text{fun } x \rightarrow e)} &= \text{fv}(e) \setminus \{x\} \\
\text{fv}(e_1 \ e_2) &= \text{fv}(e_1) \cup \text{fv}(e_2) \\
\text{fv(\text{let } x = e_1 \ \text{in } e_2)} &= \text{fv}(e_1) \cup (\text{fv}(e_2) \setminus \{x\}) \\
\text{fv(\text{let rec } x = e_1 \ \text{in } e_2)} &= (\text{fv}(e_1) \cup \text{fv}(e_2)) \setminus \{x\} \\
\text{fv(\text{if } e_1 \ \text{then } e_2 \ \text{else } e_3)} &= \text{fv}(e_1) \cup \text{fv}(e_2) \cup \text{fv}(e_3)
\end{align*}
\]
let us consider the following program approximating $\int_0^1 x^n dx$

```ocaml
let rec pow i x =  
  if i = 0 then 1. else x *. pow (i-1) x

let integrate_xn n =  
  let f = pow n in  
  let eps = 0.001 in  
  let rec sum x =  
    if x >= 1. then 0. else f x +. sum (x +. eps) in  
  sum 0. *. eps
```
let us make constructions \texttt{fun} explicit and let us consider the closures

\begin{verbatim}
let rec pow =
  fun i ->
    fun x -> if i = 0 then 1. else x *. pow (i-1) x
\end{verbatim}

- in the first closure, \texttt{fun i ->}, the environment is \{\texttt{pow}\}
- in the second closure, \texttt{fun x ->}, it is \{i, pow\}
let integrate_xn = fun n ->
  let f = pow n in
let eps = 0.001 in
let rec sum =
  fun x -> if x >= 1. then 0. else f x +. sum (x+.eps) in
sum 0. *. eps

• for fun n ->, the environment is \{pow\}
• for fun x ->, the environment is \{eps, f, sum\}
implementing the closure

the closure is a single heap-allocated block, whose

- first field contains the code pointer
- next fields contains the values of the free variables
let rec pow i x = if i = 0 then 1. else x * pow (i-1) x

let integrate_xn n =
  let f = pow n in
  let eps = 0.001 in
  let rec sum x = if x >= 1. then 0. else f x + sum (x+.eps) in
  sum 0. *. eps

during the execution of `integrate_xn 100`, we have four closures:
a good way to compile closures is to proceed in two steps

1. first, we replace all expressions fun x → e by explicit closure constructions

\begin{equation}
\text{clos } f \ [y_1, \ldots, y_n]
\end{equation}

where the \( y_i \) are the free variables of \( \text{fun } x \rightarrow e \) and \( f \) is the name of a global function

\begin{equation}
\text{letfun } f \ [y_1, \ldots, y_n] x = e'
\end{equation}

where \( e' \) is derived from \( e \) by replacing constructions \( \text{fun} \) recursively (\textit{closure conversion})

2. we compile the obtained code, which only contains \text{letfun} function declarations
on the example, we get

```
letfun fun2 [i,pow] x = 
    if i = 0 then 1. else x *. pow (i-1) x
letfun fun1 [pow] i = 
    clos fun2 [i,pow]
let rec pow = 
    clos fun1 [pow]
letfun fun3 [eps,f,sum] x = 
    if x >= 1. then 0. else f x +. sum (x +. eps)
letfun fun4 [pow] n = 
    let f = pow n in 
    let eps = 0.001 in 
    let rec sum = clos fun3 [eps,f,sum] in 
    sum 0. *. eps
let integrate_xn = 
    clos fun4 [pow]
```
before

\[

e ::= c \\
| x \\
| \text{fun } x \rightarrow e \\
| e \ e \\
| \text{let } [\text{rec}] x = e \text{ in } e \\
| \text{if } e \text{ then } e \text{ else } e \\
\]

\[
d ::= \text{let } [\text{rec}] x = e \\
\]

\[
p ::= d \ldots d \\
\]

after

\[

e ::= c \\
| x \\
| \text{clos } f [x, \ldots, x] \\
| e \ e \\
| \text{let } [\text{rec}] x = e \text{ in } e \\
| \text{if } e \text{ then } e \text{ else } e \\
\]

\[
d ::= \text{let } [\text{rec}] x = e \\
| \text{letfun } f [x, \ldots, x] x = e \\
\]

\[
p ::= d \ldots d \\
\]
variables

in the new syntax trees, an identifier \( x \) can be

- a **global variable** introduced by \texttt{let}
  (allocated in the data segment)

- a **local variable** introduced by \texttt{let in}
  (allocated in the stack frame / a register)

- a **variable contained in a closure**

- the **argument** of a function (the \( x \) of \texttt{fun \( x \) \( \rightarrow \) e})
each function has a single argument, passed in register `%rdi`

the closure is passed in register `%rsi`

the stack frame is as follows, where $v_1, \ldots, v_m$ are the local variables

it is built by the callee

<table>
<thead>
<tr>
<th>return address</th>
<th>%rbp →</th>
</tr>
</thead>
<tbody>
<tr>
<td>saved %rbp</td>
<td></td>
</tr>
<tr>
<td>$v_1$</td>
<td></td>
</tr>
<tr>
<td>$\vdots$</td>
<td></td>
</tr>
<tr>
<td>$v_m$</td>
<td></td>
</tr>
<tr>
<td>$\vdots$</td>
<td></td>
</tr>
<tr>
<td>↓</td>
<td></td>
</tr>
</tbody>
</table>
let us detail how to compile

- the construction of a closure $\text{clos } f [y_1, \ldots, y_n]$
- a function call $e_1 e_2$
- the access to a variable $x$
- a function declaration $\text{letfun } f [y_1, \ldots, y_n] x = e$
to compile

\texttt{clos \ f \ [y_1, \ldots, y_n]}

we proceed as follows

1. we allocate a block of size \( n + 1 \) on the heap (with a GC)
2. we store the address of \( f \) in field 0
   \( (f \) is a label in the assembly code and we get its address with \( \$f \))
3. we store the values of the variables \( y_1, \ldots, y_n \) in fields 1 to \( n \)
4. we return a pointer to the block

note: we delegate the liberation of the block to the GC (see lecture 9)
to compile a function call

\[ e_1 \ e_2 \]

we proceed as follows

1. we compile \( e_1 \) into register \( %rsi \)
   (its value is a \( p_1 \) to a closure)
2. we compile \( e_2 \) into register \( %rdi \)
3. we call the function whose address is contained in the first field of the closure, with \code{call *(%rsi)}

this is a jump to \textit{dynamic address}
(similar to what we do when compiling OO languages)
to compile the access to the variable $x$, we distinguish four cases

global variable
  the value is stored at the address given by label $x$

local variable
  the value is at $n(\%rbp) /$ in a register

variable contained in a closure
  the value is at $n(\%rsi)$

function argument
  the value is in register $\%rdi$
last, to compile the declaration

\[
\text{letfun } f [y_1, \ldots, y_n] x = e
\]

we proceed as for a usual function declaration

1. save and set \%rbp
2. allocate the frame (for the local variables of \(e\))
3. evaluate \(e\) in register \%rax
4. delete the stack frame and restore \%rbp
5. execute `ret`
today we find closures in
  • Java (since 2014 and Java 8)
  • C++ (since 2011 and C++11)

in these languages, anonymous functions are called **lambdas**
a function is a regular object, with a method apply

```java
LinkedList<B> map(LinkedList<A> l, Function<A, B> f) {
    ... f.apply(x) ...
}
```

an anonymous function is introduced with ->

```java
map(l, x -> { System.out.print(x); return x+y; })
```

the compiler build a closure object (here capturing the value of y) with a method apply
an anonymous function is introduced with []

```cpp
for_each(v.begin(), v.end(), [y](int &x){ x += y; });
```

we specify the variables captured in the closure (here y)

we may specify a capture by reference (here of s)

```cpp
for_each(v.begin(), v.end(), [y,&s](int x){ s += y*x; });
```

the compiler builds a closure (whose type is not accessible)
tail call optimization
Definition

We say that a function call \( f(e_1, \ldots, e_n) \) that appears in the body of a function \( g \) is a **tail call** if this is the very thing that \( g \) computes before it returns.

by extension, we can say that a function is a **tail recursive function** if it is a recursive function whose recursive calls are all tail calls.
tail calls and recursive functions

a tail call is not necessarily a recursive call

```c
int g(int x) {
    int y = x * x;
    return f(y);
}
```

in a recursive function, we may have recursive calls that are tail calls and others that are not

```c
int f91(int n) {
    if (n > 100) return n - 10;
    return f91(f91(n + 11));
}
```
what is the point with tail calls?

we can delete the stack frame of the function performing the tail call before we make the call, since it is not needed afterwards

better, we can reuse it to make the tail call (in particular, the return address is the right one)

said otherwise, we can make a jump rather than a call
int fact(int acc, int n) {
    if (n <= 1) return acc;
    return fact(acc * n, n - 1);
}

traditional compilation

fact:    cmpq  $1, %rsi
         jle   L0
         imulq %rsi, %rdi
         decq %rsi
         call  fact
         ret
L0:      movq  %rdi, %rax
         ret

optimization

fact:    cmpq  $1, %rsi
         jle   L0
         imulq %rsi, %rdi
         decq %rsi
         jmp   fact  # <--
L0:      movq  %rdi, %rax
         ret
the result is a loop

the code is indeed identical to the compilation of

```c
int fact(int acc, int n) {
    while (n > 1) {
        acc *= n;
        n--;
    }
    return acc;
}
```
the compiler gcc optimizes tail calls when we pass option
-foptimize-sibling-calls (included in option -02)

have a look at the code produced by gcc -02 on programs such as fact
or those of slide ??
in particular, we notice that

```c
int f91(int n) {
    if (n > 100) return n - 10;
    return f91(f91(n + 11));
}
```

is compiled **exactly** as if we were compiling

```c
int f91(int n) {
    while (n <= 100)
        n = f91(n + 11);
    return n - 10;
}
```
the OCaml compiler optimizes tail calls by default

the compilation of

```ocaml
let rec fact acc n =
  if n <= 1 then acc else fact (acc * n) (n - 1)
```

is a loop, as with the C program

even if we started with a functional program (variables acc and n are immutable)
with tail call optimization, we get a more efficient code since we have reduced memory access (we do not use \texttt{call} and \texttt{ret} anymore, which require the stack)
on the fact example, the stack space becomes constant

in particular, we avoid any stack overflow due to a too large number of nested calls

Stack overflow during evaluation (looping recursion?).

Fatal error: exception Stack_overflow

Exception in thread "main" java.lang.StackOverflowError

Segmentation fault

etc.
if we implement quicksort as follows

```c
void quicksort(int a[], int l, int r) {
    if (r - l <= 1) return;
    // partition a[l..r[ in three
    //     l   lo   hi   r
    //     +--------------------------+
    // a|...<p...|...=p...|...>p...|
    //     +--------------------------+
    ...
    quicksort(a, l, lo);
    quicksort(a, hi, r);
}
```

we could overflow the stack
but if we make the first recursive call on the smallest half

```c
void quicksort(int a[], int l, int r) {
    ...
    if (lo - l < r - hi) {
        quicksort(a, l, lo);
        quicksort(a, hi, r);
    } else {
        quicksort(a, hi, r);
        quicksort(a, l, lo);
    }
}
```

the second call is a tail call and a logarithmic stack space is now guaranteed
application: quicksort

what if my compiler does not optimize tail calls (e.g. Java)?

no problem, do it yourself!

```c
void quicksort(int a[], int l, int r) {
    while (r - l > 1) {
        ...
        if (lo - l < r - hi) {
            quicksort(a, l, lo);
            l = hi;
        } else {
            quicksort(a, hi, r);
            r = lo;
        }
    }
}
```
it is important to point out that the notion of tail call

- could be optimized in any language
  (but Java and Python do not, for instance)

- is not related to recursion
  (even if it is likely that a stack overflow is due to a recursive function)
it is not always easy to turn calls into tail calls

eexample: given a type for immutable binary trees, such as

```ocaml
type 'a tree = Empty | Node of 'a tree * 'a * 'a tree
```

implement a function to compute the height of a tree

```ocaml
val height: 'a tree -> int
```
the natural code

\[
\text{let rec height = function}\n\]
\[
| \text{Empty} \rightarrow 0 \n| \text{Node (l, _, r)} \rightarrow 1 + \max (\text{height l}) (\text{height r})
\]

provokes a stack overflow on a tree with a large height
instead of computing the height $h$ of the tree, let us compute $k(h)$ for some arbitrary function $k$, called a **continuation**

```ocaml
val height: 'a tree -> (int -> 'b) -> 'b
```

we call this **continuation-passing style** (or CPS)

the height of a tree is then obtained with the identity continuation

```ocaml
height t (fun h -> h)
```
the code looks like

```ocaml
let rec height t k = match t with
  | Empty ->
    k 0
  | Node (l, _, r) ->
    height l (fun hl ->
      height r (fun hr ->
        k (1 + max hl hr)))
```

we note that all calls to height and k are tail calls

thus height runs in constant stack space
we have replaced stack space with **heap space**

it holds closures

the first closure captures \( r \) and \( k \), the second one captures \( hl \) and \( k \)
of course, there are other, ad hoc, solutions to compute the height of a tree without overflowing the stack (e.g. a breadth-first traversal)

similarly, there are solutions for mutable trees, trees with parent pointers, etc.

but the CPS-based solution is **systematic**
and what if the language optimizes tail calls but does not offer anonymous functions (e.g. C)?

we simply have to build closures by ourselves, manually (a structure with a function pointer and an environment)
we can even introduce some ad hoc data type for closures

```c
enum kind { Kid, Kleft, Knight };

struct Kont {
    enum kind kind;
    union { struct Node *r; int hl; }
    struct Kont *kont;
};
```

together with a function to apply it

```c
int apply(struct Kont *k, int v) { ... }
```

defunctionalization (Reynolds 1972)
mini-C optimizing compiler continued
code production is decomposed into several phases

1. instruction selection
2. RTL (Register Transfer Language)
3. ERTL (Explicit Register Transfer Language)
4. LTL (Location Transfer Language)
5. linearization
int fact(int x) {
    if (x <= 1) return 1;
    return x * fact(x-1);
}

phase 1: instruction selection

int fact(int x) {
    if (Mjlei 1 x) return 1;
    return Mmul x fact((Maddi -1) x);
}
phase 2: RTL (Register Transfer Language)

#2 fact(#1)
- entry : L10
- exit : L1
- locals:
  - L10: mov #1 #6 --> L9
  - L9 : jle $1 #6 --> L8, L7
  - L8 : mov $1 #2 --> L1

L7: mov #1 #5 --> L6
L6: add $-1 #5 --> L5
L5: #3 <- call fact(#5) --> L4
L4: mov #1 #4 --> L3
L3: mov #3 #2 --> L2
L2: imul #4 #2 --> L1
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:

\[
\begin{array}{c|c}
%rbp & \rightarrow \\
\hline
\vdots & \vdots \\
\text{param.} \ n & \\
\vdots & \\
\text{param.} \ 7 & \\
\text{return addr.} & \\
\hline
\text{saved} \ %rbp & \\
\hline
\text{local} \ 1 & \\
\vdots & \\
\hline
%rsp & \rightarrow \\
\hline
\vdots & \\
\hline
\text{local} \ m & \\
\vdots & \\
\end{array}
\]

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 \( m \)
ERTL instructions (1/3)

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

- **mov** $n \ r \rightarrow L$
- **load** $n(r_1) \ r_2 \rightarrow L$
- **store** $r_1 \ n(r_2) \rightarrow L$
- **unop** $op \ r \rightarrow L$
- **binop** $op \ r_1 \ r_2 \rightarrow L$
- **ubranch** $br \ r \rightarrow L_1, L_2$
- **bbranch** $br \ r_1 \ r_2 \rightarrow L_1, L_2$
- **goto** $\rightarrow L$

unary operation (neg, etc.)
binary operation (add, mov, etc.)
unary branching (jz, etc.)
binary branching (jle, etc.)
in RTL, we had

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

in ERTL, we now have

\[
\text{call } f(k) \rightarrow L
\]

\textit{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 \( n(\%r\_bp) \))
- `push_param r → 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

\[ L_1 : \text{binop div } r_1 \ r_2 \rightarrow L \]

becomes three ERTL instructions

\[ L_1 : \text{binop mov } r_2 \ %rax \rightarrow L_2 \]
\[ L_2 : \text{binop div } r_1 \ %rax \rightarrow L_3 \]
\[ L_3 : \text{binop mov } %rax r_2 \rightarrow L \]

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

\[ L_1 : \text{call } r \leftarrow f(r_1, \ldots, r_n) \rightarrow L \]

into a sequence of ERTL instructions

1. copy \( \min(n, 6) \) parameters \( r_1, r_2, \ldots \) into \%rdi,\%rsi,\ldots
2. if \( n > 6 \), pass other parameters on the stack with \text{push\_param}
3. execute \text{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: #3 <- call fact(#5) --> L4

is translated into the ERTL code

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

**RTL**

\[ r \text{ } f(r, \ldots, r) \]

- locals : \{ r, \ldots \}
- entry : L
- exit : L
- \text{cfg} : \ldots

**ERTL**

\[ f(n) \]

- locals : \{ r, \ldots \}
- entry : L
- \text{cfg} : \ldots
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
(to make things simpler, we here assume that callee-saved registers are limited to \%rbx and \%r12)
at function exit, we

• copy the pseudo-register holding the result into \%rax
• restore the saved registers
• delete the stack frame
<table>
<thead>
<tr>
<th>RTL</th>
<th>ERTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>#2 fact(#1)</td>
<td>fact(1)</td>
</tr>
<tr>
<td>entry : L10</td>
<td>entry : L17</td>
</tr>
<tr>
<td>exit : L1</td>
<td></td>
</tr>
<tr>
<td>locals:</td>
<td>locals: #7, #8</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>L8 : mov $1 #2 --&gt; L1</td>
<td></td>
</tr>
<tr>
<td>L2 : imul #4 #2 --&gt; L1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>L1 : mov #2 %rax --&gt; L21</td>
<td></td>
</tr>
<tr>
<td>L21: mov #7 %rbx --&gt; L20</td>
<td></td>
</tr>
<tr>
<td>L20: mov #8 %r12 --&gt; L19</td>
<td></td>
</tr>
<tr>
<td>L19: delete_frame --&gt; L18</td>
<td></td>
</tr>
<tr>
<td>L18: return</td>
<td></td>
</tr>
</tbody>
</table>
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 to allocate the stack frame
the next phase translates ERTL to \textbf{LTL} (Location Transfer Language)

the goal is to get rid of pseudo-registers, replacing them with
\begin{itemize}
\item physical registers preferably
\item stack locations otherwise
\end{itemize}

this is called \textit{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 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

**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

- \( \text{def}(l) \) for the set of variables defined by this instruction,
- \( \text{use}(l) \) for the set of variables used by this instruction.

**example:** for the instruction \( \text{add} \ r_1 \ r_2 \) we have

\[
\text{def}(l) = \{ r_2 \} \quad \text{and} \quad \text{use}(l) = \{ r_1, r_2 \}
\]
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*

- $\text{in}(l)$ for the set of live variables on the set of incoming edges to $l$,
- $\text{out}(l)$ for the set of live variables on the set of outcoming edges from $l$. 
the equations defining \( \text{in}(l) \) and \( \text{out}(l) \) are the following

\[
\begin{align*}
\text{in}(l) &= \text{use}(l) \cup (\text{out}(l) \setminus \text{def}(l)) \\
\text{out}(l) &= \bigcup_{s \in \text{succ}(l)} \text{in}(s)
\end{align*}
\]

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)
fixpoint computation

\[
\begin{align*}
\text{in}(l) &= \text{use}(l) \cup (\text{out}(l) \setminus \text{def}(l)) \\
\text{out}(l) &= \bigcup_{s \in \text{succ}(l)} \text{in}(s)
\end{align*}
\]

we get the fixpoint with 7 iterations
assuming the control-flow graph has $N$ nodes and $N$ variables, a brute force computation has complexity $O(N^4)$ 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 \( \text{in}(l) \) changes, then we only need to redo the computation for the predecessors of \( l \)

\[
\begin{align*}
\text{out}(l) &= \bigcup_{s \in \text{succ}(l)} \text{in}(s) \\
\text{in}(l) &= \text{use}(l) \cup (\text{out}(l) \setminus \text{def}(l))
\end{align*}
\]

here is the algorithm:

let \( WS \) be a set containing all nodes
while \( WS \) is not empty
  remove a node \( l \) from \( WS \)
  \( \text{old_in} \leftarrow \text{in}(l) \)
  \( \text{out}(l) \leftarrow \ldots \)
  \( \text{in}(l) \leftarrow \ldots \)
  if \( \text{in}(l) \) is different from \( \text{old_in}(l) \) then
    add all predecessors of \( l \) in \( WS \)
computing the sets $\text{def}(l)$ (definitions) and $\text{use}(l)$ (uses) is straightforward for most instructions.

**Examples**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>$\text{def}$</th>
<th>$\text{use}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{mov } n \ r$</td>
<td>${r}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\text{mov } r_1 \ r_2$</td>
<td>${r_2}$</td>
<td>${r_1}$</td>
</tr>
<tr>
<td>$\text{unop } \text{op } r$</td>
<td>${r}$</td>
<td>${r}$</td>
</tr>
<tr>
<td>$\text{goto}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
this is more subtle for function calls

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

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

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
```
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
lab 7

ERTL code production
liveness analysis
some code is provided (for OCaml and Java)

- ERTL abstract syntax
- for each ERTL instruction, the sets $\text{def}/\text{use}/\text{successors}$
- ERTL interpreter to test
- ERTL pretty-printer to debug
- `main.ml / Main.java` for the command line

```
./mini-c --debug --interp-ertl test.c
```
the module/class Register also implements physical registers (with the same type) and provides

- `Register.parameters` the first six parameters (list)
- `Register.result` for the function result (%rax)
- `Register.rax` for the division
- `Register.callee_saved` list of callee-saved registers
- `Register.rsp` for stack manipulation