INF564 – Compilation

Jean-Christophe Filliâtre

object-oriented languages
functional languages
today we focus on the compilation of

1. object-oriented languages
   - object layout
   - dynamic dispatch

2. functional languages
   - first-class functions
   - tail call optimization
compiling OO languages
let us explain

- how an object is represented
- how a method call is implemented

let us use Java as an example (for the moment)
class Vehicle {
    static int start = 10;
    int position;
    Vehicle() { position = start; }
    void move(int d) { position += d; }
}

class Car extends Vehicle {
    int passengers;
    void await(Vehicle v) {
        if (v.position < position)
            v.move(position - v.position);
        else
            move(10);
    }
}

class Truck extends Vehicle {
    int load;
    void move(int d) {
        if (d <= 55) position += d; else position += 55;
    }
}
representing objects

an object is an heap-allocated block, containing

- its class (and a few other items of information)
- the values of its fields

the value of an object is a pointer to the block

**key idea:** simple inheritance allows us to store the value of some field $x$ at some fixed position in the block: own fields are placed after inherited fields

<table>
<thead>
<tr>
<th>Vehicle</th>
<th>Car</th>
<th>Truck</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
<td>position</td>
<td>position</td>
</tr>
<tr>
<td></td>
<td>passengers</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>load</td>
</tr>
</tbody>
</table>

note the absence of field `start`, which is static and thus allocated elsewhere (for instance in the data segment)
Truck t = new Truck();
Car c = new Car();
c.passengers = 2;
c.move(60);
Vehicle v = c;
v.move(70);
c.await(t);
for each field, the compiler knows its position, that is the offset to add to the object pointer

if for instance field position is at offset +16, then expression e.position is compiled to

```assembly
...  # we compile e in %rcx
movl 16(%rcx), %eax  # field position at +16
```

this is sound, even if the compiler only knows the static type of e, which may differ from the dynamic type (the class of the object)

it could even be a sub-class of Vehicule that is not yet defined!
**overriding** is the ability to redefine a method in a subclass (so that objects in that subclass behave differently)

example: in class Truck

```java
class Truck extends Vehicle {
    ...
    void move(int d) { ... }
}
```

the method move, inherited from class Vehicle, is **overridden**
the essence of OO languages lies in **dynamic method call** $e.m(e_1, \ldots, e_n)$ (aka dynamic dispatch / message passing)

to do this, we build **class descriptors** containing addresses to method codes (aka **dispatch table**, **vtable**, etc.)

as for class fields, simple inheritance allows us to store the address of (the code of) method $m$ at a fixed offset in this descriptor

class descriptors can be allocated in the data segment; each object points to its class descriptor
class Vehicule { void move(int d) {...} }
class Car extends Vehicule { void await(Vehicule v) {...}}
class Truck extends Vehicule { void move(int d) {...} }

descr. Vehicule
   Vehicule_move

descr. Car
   Vehicule_move
   Car_wait

descr. Truck
   Truck_move
Truck t = new Truck();
Car c = new Car();
c.passengers = 2;
c.move(60);
Vehicle v = c;
v.move(70);
c.await(t);
to compile a call such as e.move(10)

1. we compile e; its value is a pointer to an object
2. this object contains a pointer to its class descriptor
3. inside, the code for method move is located at some offset known from the compiler (for instance +8)

```asm
... # compile e into %rdi
movq $10, %rsi # parameter
movq (%rdi), %rcx # get the class descriptor
call *8(%rcx) # call method move
```

as for field access, the compiler has no need to know the actual class of the object (the dynamic type)
if we write

```java
Truck v = new Truck();
((Vehicule)v).move();
```

this is the method `move` from class `Truck` that is called
since the call is always compiled the same way

the cast only has an influence on the static type
(existence of the method + overloading resolution; see lecture 4)
in practice, the class descriptor for \( C \) also contains points to the class that \( C \) inherits from, called the \textbf{super class} of \( C \)

this can be a pointer to the descriptor of the super class (for instance stored in the very first slot of the descriptor)

this allows subtyping tests at runtime (\textit{downcast} or \textit{instanceof})
a few words on C++
let us reuse the vehicles example

```cpp
class Vehicle {
    static const int start = 10;

public:
    int position;
    Vehicle() { position = start; }
    virtual void move(int d) { position += d; }
};
```

`virtual` means that method `move` can be overridden
class Car : public Vehicle {
public:
  int passengers;
  Car() {}
  void await(Vehicle &v) { // call by reference
    if (v.position < position)
      v.move(position - v.position);
    else
      move(10);
  }
};
class Truck : public Vehicle {
public:
    int load;
    Truck() {}
    void move(int d) {
        if (d <= 55) position += d; else position += 55;
    }
};
```cpp
#include <iostream>
using namespace std;

int main() {
    Truck t; // objects are stack-allocated
    Car c;
    c.passengers = 2;
    c.move(60);
    Vehicle *v = &c; // alias
    v->move(70);
    c.await(t);
}
```
on this example, object representation is not different from Java’s

<table>
<thead>
<tr>
<th>descr. Vehicle</th>
<th>descr. Car</th>
<th>descr. Truck</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
<td>position</td>
<td>position</td>
</tr>
<tr>
<td>passengers</td>
<td></td>
<td>load</td>
</tr>
</tbody>
</table>
but in C++, we also **multiple inheritance**

consequence: we cannot use anymore the principle that

- the object layout for the super class is a prefix of the object layout of the subclass
- the descriptor for the super class is a prefix of the descriptor for the subclass
multiple inheritance

class Rocket {
public:
  float thrust;
  Rocket() { }
  virtual void display() {}  }
}

class RocketCar : public Car, public Rocket {
public:
  char *name;
  void move(int d) { position += 2*d; }
};

represenations of Car and Rocket are appended
in particular, a cast such as

```c
RocketCar rc;
... (Rocket)rc ...
```

is compiled using pointer arithmetic

```c
... rc + 16 ...
```

this is not a no-op anymore
let us now assume that Rocket also inherits from Vehicle

```cpp
class Rocket : public Vehicle {
public:
    float thrust;
    Rocket() {}
    virtual void display() {}
};

class RocketCar : public Car, public Rocket {
public:
    char *name;
    ...
};
```

descr. RocketCar
  position
  passengers

descr. Rocket
  position
  thrust
  name

we now have two fields position
multiple inheritance

and thus a possible ambiguity

class RocketCar : public Car, public Rocket {
public:
    char *name;
    void move(int d) { position += 2*d; }
};

vehicles.cc: In member function ‘virtual void RocketCar::move(int)’
vehicles.cc:51:22: error: reference to ‘position’ is ambiguous
multiple inheritance

we have to say which one we refer to

class RocketCar : public Car, public Rocket {
public:
    char *name;
    void move(int d) { Rocket::position += 2*d; }
};
to have a single instance of `Vehicle` inside `RocketCar`, we need to modify the way `Car` and `Rocket` inherit from `Vehicle`; this is **virtual inheritance**

```cpp
class Vehicle { ... };

class Car : public virtual Vehicle { ... };

class Rocket : public virtual Vehicle { ... };

class RocketCar : public Car, public Rocket {

there is no ambiguity anymore:

```cpp
public:
    char *name;
    void move(int d) { position += 2*d; }
};
```
three class diagrams

```cpp
class Vehicle { ... };
class Car : Vehicle { ... };
class Rocket { ... };
class RocketCar : Car, Rocket { ... };
```

```cpp
class Vehicle { ... };
class Car : Vehicle { ... };
class Rocket : Vehicle { ... };
class RocketCar : Car, Rocket { ... };
```

```cpp
class Vehicle { ... };
class Car : virtual Vehicle { ... };
class Rocket : virtual Vehicle { ... };
class RocketCar : Car, Rocket { ... };
```

(the diamond)
if you are curious

g++’s command line option -fdump-lang-class outputs a text file containing objects and tables layout
though Java only features simple inheritance, interfaces make method call more complex, in a way analogous to multiple inheritance

```java
interface I {
    void m();
}

class C {
    void foo(I x) { x.m(); }
}
```

when compiling `x.m()`, we have no idea what the class of object `x` will be
instead of dispatching according to the type of the object, we can use the types of all the actual parameters; this is called **multiple dispatch**

an example: Julia, a mathematically-oriented language

```plaintext
function +(x::Int64 , y::Int64 ) ... end
function +(x::Float64, y::Float64) ... end
function +(x::Date , y::Time ) ... end
```

another example: CLOS (Common Lisp Object System)
**pattern matching**, as we find in OCaml for instance, e.g.,

```ocaml
let rec eval = function
| Const n -> ...  
| Call ("print", [e]) -> ...  
| Call (f, el) -> ...  
```

is a form of dynamic dispatch: the branch is selected according to some runtime information

the polycopié (section 7.3) describes how the compiler turns pattern matching into elementary operations

see also the comparison functional/object in lecture 2
compiling functional languages
first-class functions
on 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

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

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

\[
p ::= d \ldots d
\]
functions can be nested

```ocaml
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`)
higher-order functions

we can pass functions as parameters

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

and return functions

```
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**

P. J. Landin, *The Mechanical Evaluation of Expressions*,
The Computer Journal, 1964
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
define 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 \to}, the environment is \{pow\}
- in the second closure, \texttt{fun x \to}, 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 (the environment)
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 \( \text{fun} \ x \rightarrow e \) by explicit closure constructions

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

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

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

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

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

```ocaml
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]
```
abstract syntax

before

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

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

\[
\begin{align*}
p & ::= \ d \ \ldots \ d
\end{align*}
\]

after

\[
\begin{align*}
e & ::= \ c \\
  & \mid \ x \\
  & \mid \ clos f [x, \ldots, x] \\
  & \mid \ e\ e \\
  & \mid \ \text{let [rec] } x = e \ \text{in } e \\
  & \mid \ \text{if } e \ \text{then } e \ \text{else } e
\end{align*}
\]

\[
\begin{align*}
d & ::= \ \text{let [rec] } x = e \\
  & \mid \ \text{letfun } f [x, \ldots, x] \ x = e
\end{align*}
\]

\[
\begin{align*}
p & ::= \ d \ \ldots \ d
\end{align*}
\]
in the new syntax trees, an identifier $x$ can be

- a **global variable** introduced by `let`
  (allocated in the data segment)

- a **local variable** introduced by `let in`
  (allocated in the stack frame / a register)

- a **variable contained in a closure**

- the **argument** of a function (the $x$ of `fun x -> 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

```
\begin{array}{|c|}
\hline
\text{return address} \\
%rbp \rightarrow \\
\hline
\text{saved %rbp} \\
\hline
v_1 \\
\hline
\vdots \\
\hline
v_m \\
\hline
\vdots \\
\hline
\end{array}
```
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 \( \text{letfun} \ f [y_1, \ldots, y_n] \ x = e \)
construction of a closure

to compile

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

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 deallocation 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 call \( *(\%rsi) \)

this is a jump to \textbf{dynamic address}
(similar to what we did earlier to compile 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 builds a closure object (here capturing the value of `y`) with a method apply
an anonymous function is introduced with []

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

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

the default behavior is to capture by value

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

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

the compiler builds a closure (whose type is not accessible ⇒ use auto)
tail call optimization
We say that a function call $f(e_1, \ldots, e_n)$ that appears in the body of a function $g$ is a \textbf{tail call} if this is the very thing that $g$ computes before it returns.

by extension, we can say that a function is a \textbf{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**
```c
int fact(int acc, int n) {
    if (n <= 1) return acc;
    return fact(acc * n, n - 1);
}
```

traditional compilation

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

optimization

```assembly
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 -O2)

have a look at the code produced by gcc -O2 on programs such as fact
or those of slide 63
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 manipulate 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
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

example: 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

```ocaml
let rec height = function
| Empty       -> 0
| Node (l, _, r) -> 1 + max (height l) (height r)
```

causes 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 \( h1 \) 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) {
    ... }
```

this is called **defunctionalization** (Reynolds 1972)
• lab 6
  • Mini Python compiler continued
  • lab rooms 30 & 32

• next lecture
  • optimizing compiler 1/2