INF564 – Compilation

Jean-Christophe Filliâtre

evaluation strategies
parameter passing
1. evaluations strategies, parameter passing
   - Java
   - OCaml
   - Python
   - C
   - C++

2. compiling call by value and call by reference
   - illustrated with C++
evaluation strategies, parameter passing
when **declaring** a function

```
function f(x1, ..., xn) =
  ...
```

variables $x_1, ..., x_n$ are called the **formal parameters** of $f$

and when **calling** this function

```
f(e_1, ..., e_n)
```

expressions $e_1, ..., e_n$ are called the **actual parameters** of $f$
in a language with in-place modifications, an assignment

\[ e_1 := e_2 \]

modifies a memory location designated by \( e_1 \)

the expression \( e_1 \) is limited to certain constructs, and assignments such as

\[ 42 := 17 \]
\[ \text{true} := \text{false} \]

do not make sense

an expression that is legal on the left-hand side of an assignment is called a \textit{left value}
the evaluation strategy of a language defines the order in which computations are performed

this can be defined using a formal semantics (see lecture 2)

the compiler must obey the evaluation strategy
in particular, the evaluation strategy \textit{may} specify

\begin{itemize}
  \item \textbf{when} actual parameters are evaluated
  \item the evaluation \textit{order} of operands and actual parameters
\end{itemize}

some aspects of evaluation may be \textbf{left unspecified}

this allows the compiler to perform more aggressive optimizations
(such as reordering computations)
we distinguish

- **eager evaluation**: operands / actual parameters are evaluated before the operation / the call

  examples: C, C++, Java, OCaml, Python

- **lazy evaluation**: operands / actual parameters are evaluated only when needed

  examples: Haskell, Clojure
  but also Boolean operators `&&` and `||` in most languages
evaluation and side effects

an imperative language has to adopt an eager evaluation, to ensure that side effects are performed consistently with the source code for instance, the Java code

```java
int r = 0;
int id(int x) { r += x; return x; }
int f(int x, int y) { return r; }

{ System.out.println(f(id(40), id(2))); }
```

prints 42 since both arguments of \( f \) are evaluated
an exception is made for Boolean operations && and || in most languages, which is really useful

```java
void insertionSort(int[] a) {
    for (int i = 1; i < a.length; i++) {
        int v = a[i], j = i;
        for (; 0 < j && v < a[j-1]; j--)
            a[j] = a[j-1];
        a[j] = v;
    }
}
```
non-termination is also a side effect

for instance, the Java code

```java
int loop() { while (true); return 0; }
int f(int x, int y) { return x+1; }

{ System.out.println(f(41, loop())); }
```

does not terminate, even if argument y is not used
a purely functional language (＝ without imperative features) may adopt any evaluation strategy, since an expression will always evaluate to the same value (this is called referential transparency)

in particular, it may adopt a lazy evaluation
the Haskell program

```haskell
loop () = loop ()
f x y = x
main = putChar (f 'a' (loop ()))
```

terminates (and prints a)
the semantics also defines the way parameters are passed in a function call several approaches:

- call by value
- call by reference
- call by name
- call by need

(we also say passing by value, etc.)
new variables receive the values of actual parameters

```plaintext
function f(x) =
  x := x + 1

main() =
  int v := 41
  f(v)
  print(v)  // prints 41
```
formal parameters denote the same left values as actual parameters

```plaintext
function f(x) =
   x := x + 1

main() =
   int v := 41
   f(v)
   print(v)  // prints 42
```
actual parameters are **substituted** to formal parameters, textually, and thus are evaluated only if necessary

```
function f(x, y, z) =
    return x*x + y*y

main() =
    print(f(1+2, 2+2, 1/0)) // prints 25
    // 1+2 is evaluated twice
    // 2+2 is evaluated twice
    // 1/0 is never evaluated
```
actual parameters are evaluated only if necessary, and \textbf{at most once}

```python
function f(x, y, z) =
    return x*x + y*y

main() =
    print(f(1+2, 2+2, 1/0)) // prints 25
    // 1+2 is evaluated once
    // 2+2 is evaluated once
    // 1/0 is never evaluated
```
a few words on Java
Java uses an eager evaluation, with **call by value**

evaluation order is left-to-right

a value is

- either of a primitive type (Boolean, character, machine integer, etc.)
- or a pointer to a heap-allocated object
void f(int x) {
    x = x + 1;
}

int main() {
    int v = 41;
    f(v);
    // v is still 41
}
passing an object

an object is allocated on the heap

class C { int f; }

void incr(C x) {
  x.f += 1;
}

void main () {
  C r = new C();
  r.f = 41;
  incr(r);
  // r.f now is 42
}

this is still call by value,
with a value that is an (implicit) pointer to an object
an array is an object

```java
void incr(int[] x) {
    x[1] += 1;
}
void main () {
    int[] a = new int[17];
    a[1] = 41;
    incr(a);
    // a[1] now is 42
}
```
we can **emulate call by name** in Java, by replacing parameters with functions; for instance, the function

```java
int f(int x, int y) {
    if (x == 0) return 42; else return y + y;
}
```

can be turned into

```java
int f(Supplier<Integer> x, Supplier<Integer> y) {
    if (x.get() == 0)
        return 42;
    else
        return y.get() + y.get();
}
```

and called like this

```java
int v = f(() -> 0, () -> { throw new Error(); });
```
call by need in Java

more efficiently, we can **simulate call by need** in Java

```java
class Lazy<T> implements Supplier<T> {
    private T cache = null;
    private Supplier<T> f;

    Lazy(Supplier<T> f) { this.f = f; }

    public T get() {
        if (this.cache == null) {
            this.cache = this.f.get();
            this.f = null; // allows the GC to reclaim f
        }
        return this.cache;
    }
}
```

(this is memoization)
call by need in Java

and we use it like this

```java
int w = f(new Lazy<Integer>((() -> 1),
               new Lazy<Integer>((() -> {
                   ...takes time...
               })));
```
a few words on OCaml
OCaml has an eager evaluation, with call by value.

Evaluation order is left unspecified.

A value is:
- either of a primitive type (Boolean, character, machine integer, etc.)
- or a pointer to a heap-allocated block (array, record, non constant constructor, etc.)
left values are array elements

```
a.(2) <- true
```

and mutable record fields

```
x.age <- 42
```
OCaml’s “mutable variables” (aka references) are records

```
type 'a ref = { mutable contents: 'a }
```

and operations := and ! are defined as

```
let (!) r = r.contents
let (:=) r v = r.contents <- v
```
passing a reference

A reference is allocated on the heap

```ocaml
let incr x =  
  x := !x + 1

let main () =  
  let r = ref 41 in  
  incr r  
  (* !r now is 42 *)
```

This is still call by value, with a value that is an (implicit) pointer to a mutable data
an array is allocated on the heap

let incr x =  
x.(1) <- x.(1) + 1

let main () =  
let a = Array.make 17 0 in  
a.(1) <- 41;  
incr a  
(* a.(1) now is 42 *)
to build a matrix, do not write

```ocaml
let m = Array.make 2 (Array.make 3 0)
```

but

```ocaml
let m = Array.make_matrix 2 3 0
```
we can simulate call by name in OCaml, by replacing parameters with functions

for instance, the function

```ocaml
let f x y =
  if x = 0 then 42 else y + y
```

can be turned into

```ocaml
let f x y =
  if x () = 0 then 42 else y () + y ()
```

and called like this

```ocaml
let v = f (fun () -> 0) (fun () -> failwith "oups")
```
we can also **simulate call by need** in OCaml

we first introduce a type to represent lazy computations

```ocaml
type 'a value = Value of 'a
    | Frozen of (unit -> 'a)

type 'a by_need = 'a value ref
```

and a function to evaluate a computation when it is not yet done

```ocaml
let force l = match !l with
    | Value v -> v
    | Frozen f -> let v = f () in l := Value v; v
```

(this is memoization)
then we define function \( f \) as follows

```ocaml
define f as follows
let f x y =
  if force x = 0 then 42 else force y + force y
```

and we call it with

```ocaml
call it with
let v = f (ref (Frozen (fun () -> 1)))
  (ref (Frozen (fun () -> ...takes time...)))
```

note: OCaml has a lazy construct that does something similar (but in a more subtle and more efficient way)
a few words on Python
Python has an eager evaluation, with call by value

evaluation order is left-to-right
(but right-to-left for an assignment)

a value is a pointer to a heap-allocated object
an integer is an **immutable** object

def f(x):
    x += 1
v = 41
f(v)
print(v) # prints 41

this is still **call by value**, with a value that is an (implicit) pointer to an object
def incr(x):
    x[1] += 1

a = [0] * 17
a[1] = 41
incr(a)
# a[1] now is 42
be careful
to build a matrix, do not write

\[ m = \begin{bmatrix} [0] \ast 3 \end{bmatrix} \ast 2 \]

but

\[ m = \begin{bmatrix} [0] \ast 3 \text{ for } _\_ \text{ in range}(2) \end{bmatrix} \]
execution models of Java, OCaml, and Python are very close

even if their surface languages are way different
a few words on C
C is an imperative language that is considered low-level, notably because pointers and pointer arithmetic are explicit

conversely, C can be considered as a high-level assembly language

a book that is still relevant:
*The C Programming Language*
by Brian Kernighan and Dennis Ritchie
the C language has an eager evaluation, with call by value
evaluation order is left unspecified
the types of C

• we have primitive types such as char, int, float, etc.

• a type $\tau*$ for pointers to values of type $\tau$
  
  if $p$ is a pointer of $\tau*$, then $*p$ stands for the value pointed to by $p$, of type $\tau$

  if $e$ is a left value of type $\tau$, then $&e$ is a pointer to its memory location, with type $\tau*$

• we have records, called structures, such as

```
struct L { int head; struct L *next; };
```

if $e$ has type $\text{struct } L$, we write $e$.head for a field access
in C, a left value is either

- \( x \), a variable
- \( *e \), the dereferencing of a pointer
- \( e.x \), a structure field access
  - if \( e \) is itself a left value
- \( t[e] \), that is sugar for \(*(_{t+e})\)
- \( e->x \), that is sugar for \(*e\).x
void f(int x) {
    x = x + 1;
}

int main() {
    int v = 41;
    f(v);
    // v is still 41
}
call by value means that structures are copied when passed to functions or returned

structures are also copied when variables of structure types are assigned, i.e. assignments such as \( x = y \), where \( x \) and \( y \) have type struct \( S \)
struct S { int a; int b; };

void f(struct S x) {
    x.b = x.b + 1;
}

int main() {
    struct S v = { 1, 2 };  // v.b is still 2
    f(v);
}
we can **simulate** a call by reference by passing an explicit pointer

```c
void incr(int *x) {
    *x = *x + 1;
}

int main() {
    int v = 41;
    incr(&v);
    // v now is 42
}
```

but this is still **call by value**
to avoid copies, we often use pointers to structures

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

void f(struct S *x) {
    x->b = x->b + 1;
}

int main() {
    struct S v = { 1, 2 };
    f(&v);
    // v.b now is 3
}
```
explicit pointer manipulation can be dangerous

```c
int* p() {
    int x;
    ...
    return &x;
}
```

this function returns a pointer to a memory location on the stack (the stack frame of \( p \)) that is not meaningful anymore, and that is going to be reused for another stack frame

we call this a **dangling reference**
notation $t[i]$ is syntactic sugar for $*(t+i)$ where

- $t$ is a pointer to a memory location containing consecutive integers
- $+$ stands for **pointer arithmetic** (adding $4i$ to $t$ for an array of 32 bit integers)

the first element of the array is thus $t[0]$, that is $*t$
an array may be allocated on the stack, as follows

```c
void f() {
    int t[10];
}
```

and it will be deallocated when the function exits

or allocated on the heap, as follows

```c
int *t = malloc(10 * sizeof(int));
```

and it has to be deallocated with `free` (see lecture 9)
we cannot assign arrays, only pointers

so we can’t write

```c
void p() {
    int t[3];
    int u[3];
    t = u;    // <- error
}
```

since \(t\) and \(u\) are (stack-allocated) arrays and arrays assignment is not possible
when passing an array, we only pass a pointer (by value, as always)

we can write

```c
void q(int t[3], int u[3]) {
    t = u;
}
```

and this is exactly the same as

```c
void q(int *t, int *u) {
    t = u;
}
```

and pointer assignment is possible
a few words on C++
in C++, we have (among other things) all the types and constructs of C with an eager evaluation

passing is **call by value** by default

but we also have **call by reference** indicated with symbol & at the formal parameter site
```c
void f(int &x) {
    x = x + 1;
}

int main() {
    int v = 41;
    f(v);
    // v now is 42
}
```

diagram showing values of variables before and after function call.

this is the compiler that
- passed a pointer to \( v \) at the call site
- dereferenced the pointer \( x \) in function \( f \)
the actual parameter has to be a left value

```c
void f(int &x) {
    x = x + 1;
}

int main() {
    f(41); // <- error (not a left value)
}
```
we can pass structures by reference

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

void f(struct S &x) {
    x.b = x.b + 1;
}

int main() {
    struct S v = { 1, 2 };  
    f(v);
    // v.b now is 3
}
```
we can pass pointers by reference

for instance to insert an element into a mutable tree

```c
struct Node { int elt; Node *left, *right; };

void add(Node* &t, int x) {
    if (t == NULL ) t = create(NULL, x, NULL);
    else if (x < t->elt) add(t->left, x);
    else if (x > t->elt) add(t->right, x);
}
```
<table>
<thead>
<tr>
<th>Language</th>
<th>Java</th>
<th>OCaml</th>
<th>Python</th>
<th>C</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>integer by value</td>
<td>integer by value</td>
<td>—</td>
<td>integer by value</td>
<td>integer by value</td>
</tr>
<tr>
<td>Access</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>pointer by value (object)</td>
<td>pointer by value</td>
</tr>
<tr>
<td>Access</td>
<td>pointer by value</td>
<td>—</td>
<td>pointer by value</td>
<td>pointer by value</td>
<td></td>
</tr>
<tr>
<td>Access</td>
<td>integer by value</td>
<td>—</td>
<td>—</td>
<td>pointer by value (object)</td>
<td>—</td>
</tr>
<tr>
<td>Access</td>
<td>pointer by value</td>
<td>pointer by value</td>
<td>integer by reference</td>
<td>pointer by value</td>
<td></td>
</tr>
<tr>
<td>Access</td>
<td>pointer by value</td>
<td>—</td>
<td>—</td>
<td>pointer by value or by reference</td>
<td>—</td>
</tr>
</tbody>
</table>
compiling call by value and call by reference
let us consider a tiny fragment of C++ with

- integers
- functions (without return value)
- call by value and call by reference
micro C++

\[
E \rightarrow n \quad C \rightarrow E == E \ | \ E != E \\
\quad \ | \ x \quad \ | \ E < E \ | \ E <= E \ | \ E > E \ | \ E >= E \\
\quad \ | \ E + E \ | \ E - E \ | \ C \ && \ C \\
\quad \ | \ E * E \ | \ E / E \ | \ C \ || \ C \\
\quad \ | \ - E \ | \ ! C
\]

\[
S \rightarrow x = E; \\
\quad \ | \ if \ ( C ) \ S \\
\quad \ | \ if \ ( C ) \ S \ else \ S \\
\quad \ | \ while \ ( C ) \ S \\
\quad \ | \ f(E,...,E); \\
\quad \ | \ printf("%d\n", \ E); \\
\quad \ | \ int \ x,...,x; \\
\quad \ | \ B
\]

\[
B \rightarrow \{ \ S...S \} \\
F \rightarrow \text{void } f(X,...,X) \ B \\
X \rightarrow \text{int } x \\
\quad \ | \ \text{int } &x \\
P \rightarrow F...F \\
\quad \ | \ \text{int main() } B
\]
```c
void fib(int n, int &r) {
    if (n <= 1)
        r = n;
    else {
        int tmp;
        fib(n - 2, tmp);
        fib(n - 1, r);
        r = r + tmp;
    }
}

int main() {
    int f;
    fib(10, f);
    printf("%d\n", f);
}
```
**scoping** defines the places in the code where a variable is visible here, if the body of function $f$ mentions a variable $x$, then

- either $x$ is a parameter of $f$
- or $x$ is declared upper in a block (including the current block)

beside, a variable can **shadow** another variable with the same name
```c
void f(int n) {
    printf("%d\n", n); // prints 34
    if (n > 0) {
        int n; n = 89;
        printf("%d\n", n); // prints 89
    }
    if (n > 21) {
        printf("%d\n", n); // prints 34
        int n; n = 55;
        printf("%d\n", n); // prints 55
    }
    printf("%d\n", n); // prints 34
}

int main() {
    f(34);
}
```
here, scoping only depends on the program source (this is called **lexical scoping**) and it can be solving during type checking

the abstract syntax keeps track of this analysis, by identifying each variable in a unique way
before

abstract syntax out of the parser

abstract class Expr {...}
class UseVar extends Expr
    { String name; ... }
...
abstract class Stmt {...}
class DeclVar extends Stmt
    { String name; ... }
...

variables are strings (names)

after

abstract syntax after type checking

abstract class TExpr {...}
class TUseVar extends TExpr
    { Var x; ... }
...
abstract class TStmt {...}
class TDeclVar extends TStmt
    { Var x; ... }
...

now Var is a unique identifier object
the abstract syntax tree now corresponds to something like this:

```c
void f(int n0) {
    printf("%d\n", n0);
    if (n0 > 0) {
        int n1; n1 = 89;
        printf("%d\n", n1);
    }
    if (n0 > 21) {
        printf("%d\n", n0);
        int n2; n2 = 55;
        printf("%d\n", n2);
    }
    printf("%d\n", n0);
}
```
or more precisely like this:

```c
void f(int n) {
    printf("%d\n", n);
    if (n > 0) {
        int n; n = 89;
        printf("%d\n", n);
    }
    if (n > 21) {
        printf("%d\n", n0);
        int n; n = 55;
        printf("%d\n", n);
    }
    printf("%d\n", n);
}
```
there are languages where scoping is **dynamic** i.e. depends on the execution of the program

example: bash
we need to allocate variables in memory and to be able to access them at runtime

here we allocate all the variables on the stack

each on-going function call is implemented with a portion of the stack, called a stack frame, that contains

- actual parameters
- the return address
- local variables
the stack frame of a call $f(e_1, \ldots, e_n)$ of a function $f$ with $n$ parameters

$$
\begin{array}{c|c|c}
\%rbp & \text{saved} \%rbp & \text{built} \\
\hline
%rbp & & \\
\hline
& e_n & \text{built by the caller} \\
\hline
& \vdots & \\
\hline
& e_1 & \\
\hline
\text{return addr.} & & \text{built by the callee} \\
\hline
\vdots & v_1 & \\
\hline
\vdots & \vdots & \\
\hline
\vdots & v_m & \\
\hline
\end{array}
$$

Jean-Christophe Filliâtre
```c
void g(int a, int b) {
    if (...) {
        int c;
        ...
    }
    if (...) {
        int d;
        ...
        int e;
        ...
    }
}

int main() {
    g(100, 10);
    return addr.
    \%rbp → saved \%rbp
    \%rbp → return addr.
    a 100
    b 10
    c ...
    d ...
    e ...

}
setting %rbp this way allows us to easily retrieve the location of a variable, with a constant offset (e.g. %rbp + 16 or %rbp − 8)

indeed, the top of the stack may change when

- we store temporary values
- we prepare a function call
for each variable, the compiler chooses a position in the stack frame

```java
class Var {
    String name;
    int ofs; // position wrt %rbp
}
```

```json
type var = {
    name: string;
    ofs: int;
    (* position wrt %rbp *)
}
```

assuming 64-bit integers (to make things simpler),

- for parameters, these are $+16$, $+24$, etc.
- for local variables, these are $-8$, $-16$, etc.,
  with several options, some of which more economical
let us show now how to compile micro C++ to x86-64

let us focus on call by value only for the moment
we adopt a simple compilation scheme, where the results of intermediate computations are stored on the stack (we’ll talk about register allocation in lectures 7&8)

we note $C(e)$ the assembly code produced by the compiler for an expression $e$

principles: after the execution of $C(e)$,

- the value of expression $e$ is in register `%rdi` (arbitrary choice)
- the top of the stack is unchanged
- registers other than `%rsp` and `%rbp` can be clobbered
compiling expressions

constants

\[ C(n) \overset{\text{def}}{=} \text{movq } n \ %r di \]

operations

\[ C(e_1 + e_2) \overset{\text{def}}{=} C(e_1) \]
\[ \text{pushq } %r di \]
\[ C(e_2) \]
\[ \text{popq } %r si \]
\[ \text{addq } %r si, %r di \]
of course, this is extremely inefficient; for 1+2, we get

```
  movq  $1, %rdi
  pushq %rdi
  movq  $2, %rdi
  popq  %rsi
  addq  %rsi, %rdi
```

even though we have 16 registers!
for a \textbf{variable}, we use indirect addressing, since the position \text{wrt} %rbp is a constant that the compiler knows

\[
C(x) \overset{\text{def}}{=} \text{movq ofs}(%rbp), \%rdi
\]

(reminder: we only consider call by value for the moment)
Boolean expressions are compiled in a very similar way

\[
C(e_1 = e_2) \overset{\text{def}}{=} C(e_1) \\
\text{pushq } %rdi \\
C(e_2) \\
\text{popq } %rsi \\
\text{cmpq } %rdi, %rsi \\
\text{sete } %dil \\
\text{movzbq } %dil, %rdi
\]

caveat: more complex for operators && and ||, that must be evaluated lazily i.e. \(e_2\) is not evaluated in \(e_1 \&\& e_2\) (resp. \(e_1 \| e_2\)) if \(e_1\) is false (resp. true)
a statement $s$ is compiled into a piece of assembly code $C(s)$

principles: after the execution of $C(s)$,

- the top of the stack is unchanged
- registers other than `%rsp` and `%rbp` can be clobbered
e.g. $C(\text{print}(e)) \overset{\text{def}}{=} C(e)$

call print_int

print_int:
  pushq %rbp
  movq %rsp, %rbp
  andq $-16, %rsp  # 16-byte stack alignment
  movq %rdi, %rsi
  movq $.Sprint_int, %rdi
  movq $0, %rax
  call printf
  movq %rbp, %rsp
  popq %rbp
  ret

.data
.Sprint_int:
  .string "\%d\n"
for a call to function $f$, we need to

1. push actual parameters
2. call the code at label $f$
3. pop the parameters

\[
C(f(e_1, \ldots, e_n)) \overset{\text{def}}{=} C(e_n)
\]

```assembly
pushq %rdi
... 
C(e_1)
pushq %rdi
call f
addq $8n, %rsp
```
in an assignment $x = e$;, the left value is limited to a variable $x$ and we know where this variable is located on the stack

\[
C(x = e) \overset{\text{def}}{=} C(e) \\
\text{movq } %rdi, ofs(%rbp)
\]
up to now, parameters were passed by value

i.e. the formal parameter is a new variable that receives the value of the actual parameter

in C++, the qualifier & indicates a call by reference

in this case, the formal parameter stands for the same variable as the actual parameter, which must be a variable (a left value, in the general case)
void fib(int n, int &r) {
    if (n <= 1)
        r = n;
    else {
        int tmp;
        fib(n - 2, tmp);
        fib(n - 1, r);
        r = r + tmp;
    }
}

int main() {
    int f;
    fib(10, f); // updates the value of f
    printf("%d\n", f); // prints 55
}
to account for call by reference, we extend the type of variables to indicate whether it is passed by reference

```java
class Var {
    String name;
    int ofs; // position wrt %rbp
    boolean byref;
}
```

```json
type var = {
    name: string;
    ofs: int;
    byref: bool;
}
```

(is false for a local variable)
in a call $f(e)$ the actual parameter $e$ is not typed nor compiled the same way anymore when passed by reference

indeed, the type checker

1. checks that this is a left value
2. recalls it is passed by reference
a nice way to proceed consists in adding a new construct “compute a left value” in the abstract syntax of expressions

...  
  class Addr extends TExpr {
      Var x;
  }

then we replace $f(e)$ with $f(\text{Addr}(e))$ when $e$ is passed by reference

note: this is exactly the C++ operator &, even if it is not part of our fragment
we have to extend the compilation of expressions:

\[
C(\&x) \overset{\text{def}}{=} \text{leaq ofs}(\%rbp), \%rdi \\
\quad \text{movq}(\%rdi), \%rdi \quad \text{if } x.\text{byref}
\]

note: the case \(x.\text{byref}=\text{true}\) accounts for a variable \(x\) that is itself passed by reference
void z(int &x) { x = 0; }
void h(int &s) { z(s); while (s < 100) s = 2*s+1; }
int main() { int tmp; h(tmp); printf("%d\n", tmp); }

```
ret
0

main       tmp
  s
ret

h
  x
  ret

z
  ...
```
call by reference

we also need to update the case of a variable access:

\[ C(x) \overset{\text{def}}{=} \text{movq ofs(\%rbp), \%rdi} \]
\[ \text{movq (%rdi), \%rdi} \quad \text{if } x.\text{byref} \]
as well as that of an assignment:

\[ C(x = e) \overset{\text{def}}{=} C(e) \]

\[
\begin{align*}
\text{movq } &\text{ofs}(\%rbp), \%rsi \quad \text{if } x\text{.byref} \\
\text{leaq } &\text{ofs}(\%rbp), \%rsi \quad \text{otherwise} \\
\text{movq } &\text{(}%rsi\text{,) }\%rdi
\end{align*}
\]

on the contrary, we do not have to update the compilation of a function call, thanks to the new operator &
we are left with the compilation of functions

```c
void f(x1, ..., xn) {
    // local variables y1,...,ym
    body
}
```
compute

$$fs = \max_{y_i} |y_i . ofs|$$

then

```c
f: pushq %rbp  # save %rbp
    movq %rsp, %rbp  # and set it
    subq $fs, %rsp  # allocate the frame
C(body)
    movq %rbp, %rsp  # deallocate the frame
    popq %rbp  # restore %rbp
    ret  # return to caller
```
```c
void swap(int &x, int &y) {
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

**Example**

```
void swap(int &x, int &y) {
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

**swap**:

```
pushq %rbp
movq %rsp, %rbp
subq $8, %rsp
movq 16(%rbp), %rdi
movq 0(%rdi), %rdi
leaq -8(%rbp), %rsi
movq %rdi, 0(%rsi)
movq 24(%rbp), %rdi
movq 0(%rdi), %rdi
movq 16(%rbp), %rsi
movq %rdi, 0(%rsi)
movq 24(%rbp), %rdi
movq -8(%rbp), %rdi
movq 0(%rdi), %rdi
movq %rdi, 0(%rsi)
movq %rbp, %rsi
movq %rsp, %rbp
popq %rbp
ret
```
• lab 5
  • static typing of Mini Python continued

• next lecture
  • OO and functional languages