Interpretation and Compilation of Languages

Master Programme in Computer Science

Mário Pereira mjp.pereira@fct.unl.pt

Nova School of Science and Technology, Portugal

April 29, 2025

Lecture 8

based on lectures by Jean-Christophe Filliâtre and Léon Gondelman
previous editions by João Costa Seco, Luís Caires, and Bernardo Toninho

Today: Evaluation Strategies + Parameter Passing

1. Evaluation strategies, parameter passing
   - JAVA
   - OCAML
   - Python
   - C
   - C++

2. Compiling call by value and call by reference
   - illustrated with C++

some terminology

when declaring a function

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

variables `x1,...,xn` are called the formal parameters of `f`

and when calling this function

```
f(e1, ..., en)
```

expressions `e1,...,en` are called the actual parameters of `f`

some terminology

in a language with in-place modifications, an assignment

```
e1 := e2
```

modifies a memory location designated by `e1`

the expression `e1` is limited to certain constructs,
and assignments such as

```
42 := 17
true := false
```

do not make sense

an expression that is legal on the left-hand side of an assignment is called a left value

evaluation strategy

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

evaluation strategy

in particular, the evaluation strategy may specify

- when actual parameters are evaluated
- the evaluation order of operands and actual parameters

some aspects of evaluation may be left unspecified

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

evaluation strategy

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

```
  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

exception

an exception is made for Boolean operations && and || in most languages, which is really useful

```
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;
  }
}
```

remark

non-termination is also a side effect

for instance, the Java code

```
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

purely functional programming

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

example

the Haskell program

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

terminates (and prints a)

parameter passing

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.)

call by value

new variables receive the values of actual parameters

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

main() =
  int v := 41
  f(v)
  print(v)    // prints 41
```

call by reference

formal parameters denote the same left values as actual parameters

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

main() =
  int v := 41
  f(v)
  print(v)    // prints 42
```

call by name

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
```

call by need

actual parameters are evaluated only if necessary,
and at most once

```
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

Java uses an eager evaluation, with call by value

evaluation order is left-to-right

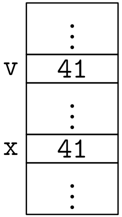
a value is

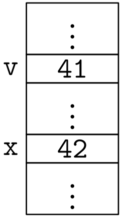
- either of a primitive type (Boolean, character, machine integer, etc.)
- or a pointer to a heap-allocated object

call by value

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

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



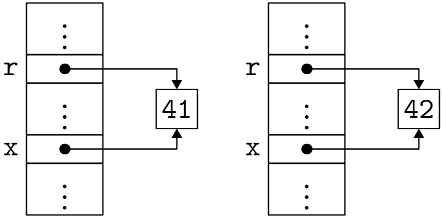


passing an object

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
}
```

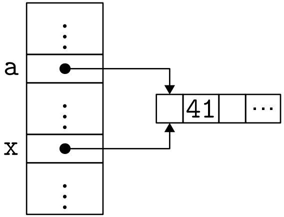


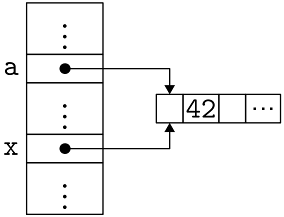
this is still call by value,
with a value that is an (implicit) pointer to an object

passing an array

an array is an object

```
void incr(int[] x) {
  x[1] += 1;
}
void main () {
  int[] a = new int[17];
  a[1] = 41;
  incr(a);
  // a[1] now is 42
}
```





call by name in Java

we can emulate call by name in Java, by replacing parameters with functions; for instance, the function

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

can be turned into

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

and called like this

```
int v = f(() -> 0, () -> { throw new Error(); });
```

call by need in Java

more efficiently, we can simulate call by need in 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

```
  int w = f(new Lazy<Integer>(() -> 1),
            new Lazy<Integer>(() -> { ...takes time... }));
```

a few words on OCaml

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

left values are array elements

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

and mutable record fields

```
  x.age <- 42
```

references

OCaml's "mutable variables" (aka references) are records

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

and operations := and ! are defined as

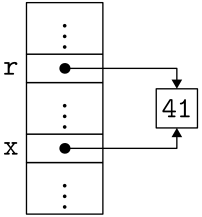
```
  let (!)  r   = r.contents
  let (:=) r v = r.contents <- v
```

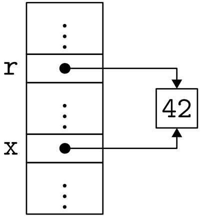
passing a reference

a reference is allocated on the heap

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

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





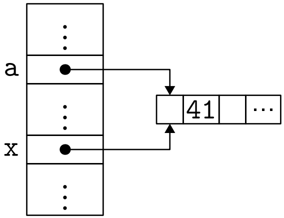
this is still call by value,
with a value that is an (implicit) pointer to a mutable data

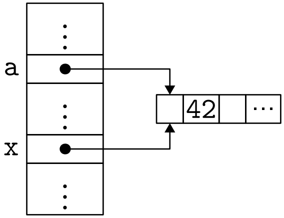
passing an array

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 *)
```

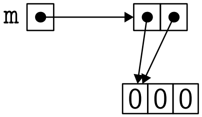




be careful

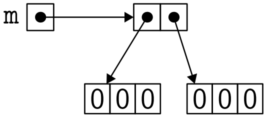
to build a matrix, do not write

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



but

```
let m = Array.make_matrix 2 3 0
```



call by name in OCaml

we can simulate call by name in OCaml, by replacing parameters with functions

for instance, the function

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

can be turned into

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

and called like this

```
let v = f (fun () -> 0) (fun () -> failwith "oups")
```

call by need in OCaml

we can also simulate call by need in OCaml

we first introduce a type to represent lazy computations

```
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

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

(this is memoization)

call by need in OCaml

then we define function `f` as follows

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

and we 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

Python has an eager evaluation, with call by value

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

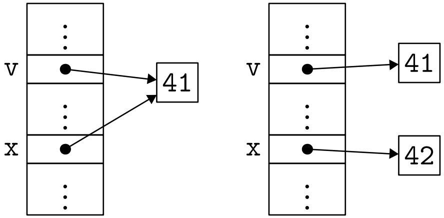
a value is a pointer to a heap-allocated object

passing an integer

an integer is an immutable object

```
def f(x):
    x += 1

v = 41
f(v)
print(v) # prints 41
```



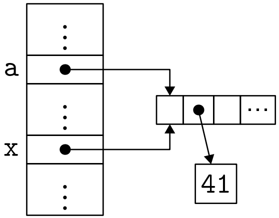
this is still call by value,
with a value that is an (implicit) pointer to an object

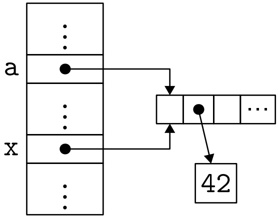
passing an array

an array is a mutable object

```
def incr(x):
    x[1] += 1

a = [0] * 17
a[1] = 41
incr(a)
# a[1] now is 42
```

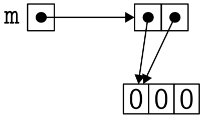




be careful

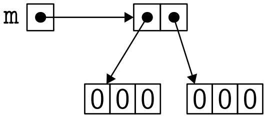
to build a matrix, do not write

```
m = [[0] * 3] * 2
```



but

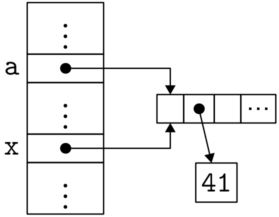
```
m = [[0] * 3 for _ in range(2)]
```

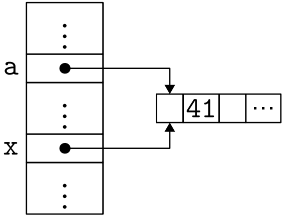


abstraction

the integers being immutable objects, we can forget they are heap-allocated objects

for instance, we can identify the following two representations





remark

the execution models of Java, OCaml, and Python are very close:
call by value only, and atomic (64 bits) values

even if their surface languages (syntax and static typing) are way different

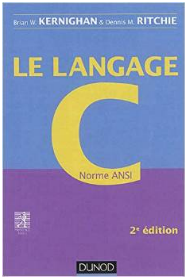
a few words on Python

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



C

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 τ

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

  if e is a left value of type τ, 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 `struct L`, we write `e.head` for a field access

the left values of C

in C, a left value is either

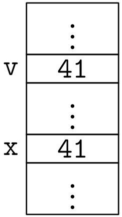
- `x`, a variable
- `*e`, the dereferencing of a pointer
- `e.x`, a structure field access
  if `e` is itself a left value

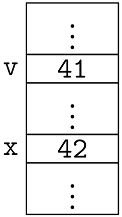
- `t[e]`, that is sugar for `*(t+e)`
- `e->x`, that is sugar for `(*e).x`

call by value

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

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





structures

call by value means that structures are copied when passed to functions or returned

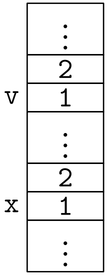
structures are also copied when variables of structure types are assigned, *i.e.* assignments such as `x = y`, where `x` and `y` have type `struct S`

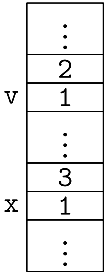
structures

```
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 is still 2
}
```



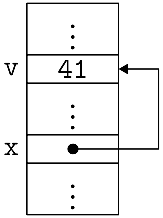


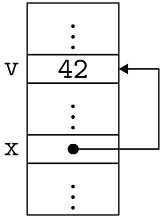
passing a pointer

we can simulate a call by reference by passing an explicit pointer

```
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

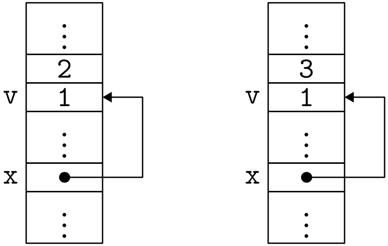
pointers to structures

to avoid copies, we often use pointers to structures

```
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
}
```



dangling reference

explicit pointer manipulation can be dangerous

```
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

arrays

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 4`i` to `t` for an array of 32 bit integers)

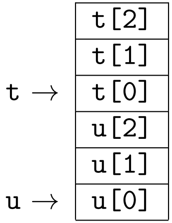
the first element of the array is thus `t[0]`, that is `*t`

arrays and pointers

we cannot assign arrays, only pointers

so we can't write

```
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)

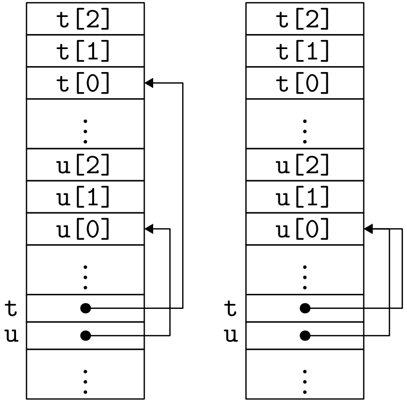
we can write

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

and this is exactly the same as

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

and pointer assignment is possible



a few words on C++

C++

in C++, we have (among other things) all the types and constructs of C with an eager evaluation

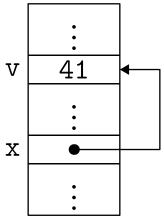
passing is call by value by default

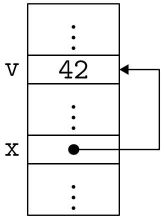
but we also have call by reference
indicated with symbol & at the formal parameter site

example

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

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





this is the compiler that

- passed a pointer to `v` at the call site
- dereferenced the pointer `x` in function `f`

left value

the actual parameter has to be a left value

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

int main() {
  f(41); // <- error (not a left value)
}
```

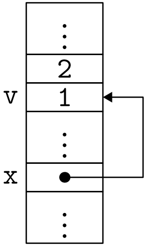
structures by reference

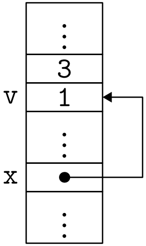
we can pass structures by reference

```
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
}
```





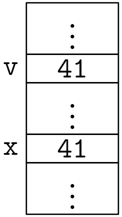
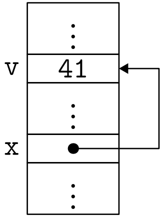
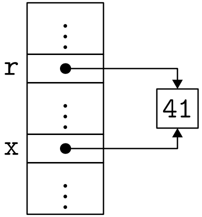
references and pointers

we can pass pointers by reference

for instance to insert an element into a mutable tree

```
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);
}
```

			
Java	integer by value	—	pointer by value (object)
OCaml	integer by value	—	pointer by value (`ref`, array, etc.)
Python	—	—	pointer by value (object)
C	integer by value	pointer by value	pointer by value
C++	integer by value	pointer by value integer by reference	pointer by value or by reference

compiling call by value and call by reference

(this might also be useful for your project)

micro C++

let us consider a tiny fragment of C++ with

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

micro C++

$$\begin{array}{rcl}
E & \to & n \\
 & | & x \\
 & | & E\ \texttt{+}\ E \mid E\ \texttt{-}\ E \\
 & | & E\ \texttt{*}\ E \mid E\ \texttt{/}\ E \\
 & | & \texttt{-}\ E
\end{array}
\qquad
\begin{array}{rcl}
C & \to & E\ \texttt{==}\ E \mid E\ \texttt{!=}\ E \\
 & | & E\ \texttt{<}\ E \mid E\ \texttt{<=}\ E \mid E\ \texttt{>}\ E \mid E\ \texttt{>=}\ E \\
 & | & C\ \texttt{\&\&}\ C \\
 & | & C\ \texttt{||}\ C \\
 & | & \texttt{!}\ C
\end{array}$$

$$\begin{array}{rcl}
S & \to & x\ \texttt{=}\ E\texttt{;} \\
 & | & \texttt{if}\ \texttt{(}\ C\ \texttt{)}\ S \\
 & | & \texttt{if}\ \texttt{(}\ C\ \texttt{)}\ S\ \texttt{else}\ S \\
 & | & \texttt{while}\ \texttt{(}\ C\ \texttt{)}\ S \\
 & | & f\texttt{(}E,\dots,E\texttt{);} \\
 & | & \texttt{printf("\%d\textbackslash n",}\ E\texttt{);} \\
 & | & \texttt{int}\ x\texttt{,}\dots\texttt{,}x\texttt{;} \\
 & | & B
\end{array}
\qquad
\begin{array}{rcl}
B & \to & \texttt{\{}\ S\dots\ S\ \texttt{\}} \\
 & & \\
F & \to & \texttt{void}\ f(X\texttt{,}\dots\texttt{,}X)\ B \\
 & & \\
X & \to & \texttt{int}\ x \\
 & | & \texttt{int}\ \texttt{\&}x \\
 & & \\
P & \to & F\dots F \\
 & & \texttt{int main()}\ B
\end{array}$$

example

```
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);
}
```

scope

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

example

```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);
}
```

scope

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

scope

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

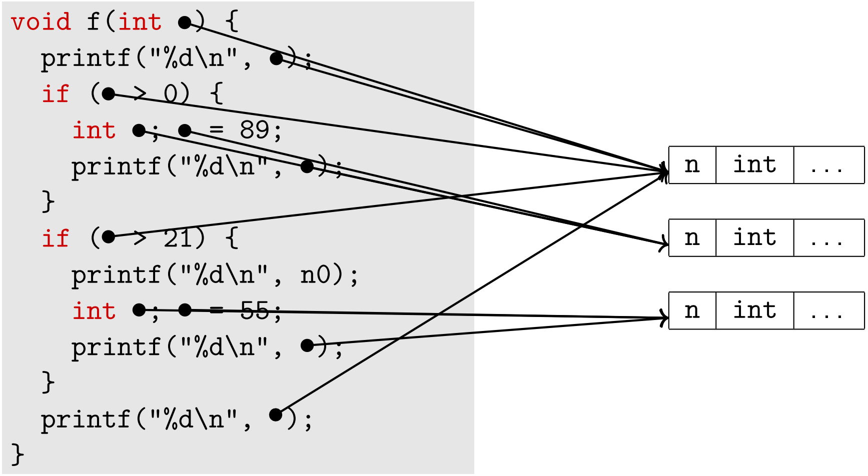
example

the abstract syntax tree now corresponds to something like this:

```
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);
}
```

example

or more precisely like this:



```
void f(int ●) {
  printf("%d\n", ●);
  if (● > 0) {
    int ●; ● = 89;
    printf("%d\n", ●);
  }
  if (● > 21) {
    printf("%d\n", n0);
    int ●; ● = 55;
    printf("%d\n", ●);
  }
  printf("%d\n", ●);
}
```

note

there are languages where scoping is dynamic i.e. depends on the execution of the program

example: `bash`

allocating variables

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

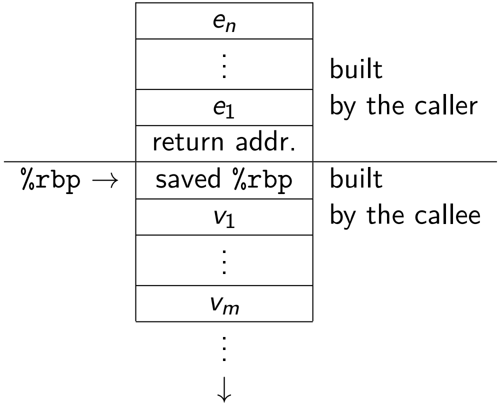
here we allocate all the variables on the stack

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

stack frame

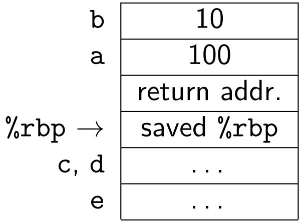
the stack frame of a call $f(e_1, \ldots, e_n)$ of a function f with n parameters



example

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

int main() {
  g(100, 10);
}
```



purpose of `%rbp`

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

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

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

```
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

compilation

let us show now how to compile micro C++ to x86-64

let us focus on call by value only for the moment

compiling expressions

we adopt a simple compilation scheme, where the results of intermediate computations are stored on the stack

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
- caller-saved registers can be clobbered

compiling expressions

constants

$C(n) \overset{\text{def}}{=}$ `movq` n `%rdi`

operations

$C(e_1 + e_2) \overset{\text{def}}{=}$ $C(e_1)$
```
pushq %rdi
```
$C(e_2)$
```
popq %rsi
addq %rsi, %rdi
```

(in)efficiency

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 variable, we use indirect addressing, since the position wrt `%rbp` is a constant that the compiler knows

$$C(x) \overset{\text{def}}{=} \texttt{movq}\ \mathit{ofs}\texttt{(\%rbp), \%rdi}$$

(reminder: we only consider call by value for the moment)

Boolean expressions

Boolean expressions are compiled in a similar way

$C(e_1 = e_2) \overset{\text{def}}{=}$ $C(e_1)$

```
pushq %rdi
```

$C(e_2)$

```
popq %rsi
cmpq %rdi, %rsi
sete %dil
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)

statements

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
- caller-saved registers can be clobbered

example

e.g. $C(\texttt{print}(e)) \overset{\text{def}}{=}$ $C(e)$
`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"
```

function call

for a call to function `f`, we need to

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

$C(f(e_1,\ldots,e_n)) \stackrel{\text{def}}{=}$ $C(e_n)$

```
pushq %rdi
⋮
```

$C(e_1)$

```
pushq %rdi
call f
addq $8n, %rsp
```

assignment

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

$$C(x = e) \overset{\text{def}}{=} \quad C(e)$$

```
                movq %rdi, ofs(%rbp)
```

call by reference

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)

example

```
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
}
```

call by reference

to account for call by reference, we extend the type of variables to indicate whether it is passed by reference

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

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

(is `false` for a local variable)

call by reference

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

call by reference

a nice way to proceed is to add 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(Addr(e))` when `e` is passed by reference

note: this is exactly the C operator `&`, even if it is not part of our fragment

call by reference

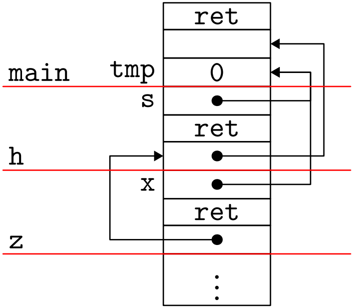
we have to extend the compilation of expressions:

$$C(\&x) \overset{\text{def}}{=} \begin{array}{ll} \texttt{leaq } \mathit{ofs}\texttt{(\%rbp), \%rdi} & \\ \texttt{movq (\%rdi), \%rdi} & \texttt{if } x\texttt{.byref} \end{array}$$

note: the case x`.byref`=`true` accounts for a variable x that is itself passed by reference

example

```
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); }
```



call by reference

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

$C(x) \overset{\text{def}}{=}$ `movq` ofs`(%rbp), %rdi`
`movq (%rdi), %rdi` if x`.byref`

call by reference

as well as that of an assignment:

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

```
C(e)
movq ofs(%rbp), %rsi      if x.byref
leaq ofs(%rbp), %rsi      otherwise
movq %rdi, (%rsi)
```

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

compiling functions

we are left with the compilation of functions

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

compiling a function

compute

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

then

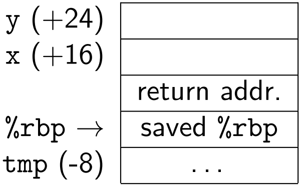
```
f:      pushq %rbp              # save %rbp
        movq  %rsp, %rbp        # and set it
        subq  $fs, %rsp         # allocate the frame
```

$C(\texttt{body})$

```
        movq  %rbp, %rsp        # deallocate the frame
        popq  %rbp              # restore %rbp
        ret                     # return to caller
```

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 -8(%rbp), %rdi
        movq 24(%rbp), %rsi
        movq %rdi, 0(%rsi)
        movq %rbp, %rsp
        popq %rbp
        ret
```