# INF108: Compilation

Louis Jachiet

# Order of evaluation

```c
int i = 0 ;
printf("%d, %d and %d\n", i++, i++);
//what does ^ output (it is the C programming language)?
```

# Left values

A *Left value* is a value that can appear on the left of an $=$, *i.e.* a "value that can be modified". In C, it can be:

- a variable
- an array cell
- a struct member
- ...

# Evaluation strategy

An evaluation strategy defines the way the arguments of a function are evaluated.

# Evaluation strategy

An evaluation strategy defines the way the arguments of a function are evaluated.

---

There exist many types of strategies, depending on:

- the order of evaluation
- the *eagerness*
- whether we pass by value/reference/sharing/etc.

# Evaluation strategy: eagerness

Two types of strategies:

**Lazy (Haskell/Clojure)**

Evaluate arguments when they are used

**Eager (most languages)**

Evaluate arguments before calling the function.

# Evaluation strategy: eagerness

Two types of strategies:

**Lazy (Haskell/Clojure)**

Evaluate arguments when they are used

**Eager (most languages)**

Evaluate arguments before calling the function.

Note that logical constructs are lazy in most programming languages...

# Call by $X$

## Call by reference:

- calls need left-values
- the evaluation passes a reference to the value
- any modification to the value modify the value everywhere

---

```
void swap(int & a, int & b) {
  int tmp = a ;
  a = b ; b = tmp ;
}
int f() {
  int a = 42, b = 17 ;
  swap(a,b); ; // now a==17, b==42
  swap(1,a); // fails, 1 is not a left-value
}
```

# Call by value:

- eager
- evaluate to an r-value all arguments
- stored in fresh memory cells

---

```
void swap(int a, int b) {
  int tmp = a ;
  a = b ; b = tmp ;
}
int f() {
  int a = 42, b = 17 ;
  swap(a,b); ; // still a==42, b==17
  swap(1,a); // works but a and b are unchanged
}
```

# Call by name / need / macro

- arguments are not evaluated
- when calling a function arguments are substituted

---

```c
#include <stdio.h>
int x = 42, y = 8;
int f() { printf("calling f!\n"); return x ; }
int g() { printf("calling g!\n"); return y ; }
#define min(a,b) a<b?a:b
int main() {
  int x = 0, y = 100;
  int f() { printf("calling f2!\n"); return 0 ; }
  return min(f(),g());
}
```

# Compilation of functions

# General scheme

## Caller needs to

- Evaluate and put arguments where expected (A0-3 or stack)
- Save temporary registers if needed
- Jump and link to the function
- Restore temporary registers

## Callee needs to

- Save all registers that needs to be saved (e.g. RA!)
- Allocate space for local variables (decreasing SP)
- Do the computation
- Deallocate space for local variables (increasing SP)
- Restore and then jump to RA

# Factorial

```
fact:
    blez $a0 fact_cas_0 # cas n=0

    # appel fact(n-1)
    sub $sp, $sp, 8
    sw $ra, 0($sp)
    sw $a0, 4($sp)

    sub $a0, $a0, 1
    jal fact

    # recupere n
    lw $a0, 4($sp)

    # fait le produit
    mul $v0, $v0, $a0

    #desalloue et retourne
    lw $ra, 0($sp)
    add $sp, $sp, 8
    jr $ra

fact_cas_0:
    li $v0, 1
    jr $ra
```

# Fibonacci

```
fibo:                   # calcul fibo(a0)
    sub $sp, $sp, 12 # allocation pile
    sw $ra, 8($sp) # on sauve ra
    sw $a0, 4($sp) # on sauve n

    # on gère les petits cas (n=0 ou 1)
    blez $a0 fibo_cas_0
    sub $a0 $a0 1
    blez $a0 fibo_cas_1

    # appel fibo(n-1)
    jal fibo

    # stockage du resultat
    sw $v0, 0($sp)

    # appel fibo(n-2)
    lw $a0, 4($sp)
    sub $a0, $a0, 2
    jal fibo

    # on recupere le calcul de fibo(n-1)
    # sur la pile
    lw $v1, 0($sp)
    # on met fibo(n-1) + fibo(n-2) dans v0
    add $v0, $v0, $v1

fin_fibo:
    lw $ra, 8($sp)
    add $sp, $sp, 12
    jr $ra

fibo_cas_0:  # cas n<=0
    li $v0, 0
    j fin_fibo
fibo_cas_1:  # cas n==1
    li $v0, 1
    j fin_fibo
```

# Fibonacci

```
fibo:                    # calcul fibo(a0)
    # on gère les petits cas (n=0 ou 1)
    blez $a0 fibo_cas_0
    sub $a0 $a0 1
    blez $a0 fibo_cas_1

    sub $sp, $sp, 12 # allocation pile
    sw $ra, 8($sp) # on sauve ra
    sw $a0, 4($sp) # on sauve n-1


    # appel fibo(n-1)
    jal fibo

    # stockage du resultat
    sw $v0, 0($sp)

    # appel fibo(n-2)
    lw $a0, 4($sp)
    sub $a0, $a0, 1
    jal fibo
```

```
    # on recupere le calcul de fibo(n-1)
    # sur la pile
    lw $v1, 0($sp)
    # on met fibo(n-1) + fibo(n-2) dans v0
    add $v0, $v0, $v1

    # on désalloue ce qui doit l'être
    lw $ra, 8($sp)
    add $sp, $sp, 12
    jr $ra


fibo_cas_0:  # cas n<=0
    li $v0, 0
    jr $ra

fibo_cas_1:  # cas n==1
    li $v0, 1
    jr $ra
```

# What does this OCaml program do?

```
let rec liste_infinie = 1::liste_infinie

let rec explore =  function
  | [] -> 42
  | a::q -> 1 + explore q


let _ = explore liste_infinie
```

# What does this OCaml program do?

```
let rec liste_infinie = 1::liste_infinie

let rec explore =  function
  | [] -> 42
  | a::q -> explore q


let _ = explore liste_infinie
```

# What does this OCaml program do?

```
let rec liste_infinie = 1::liste_infinie

let rec explore =  function
  | [] -> 42
  | a::q -> explore q


let _ = explore liste_infinie
```

**Tail-call optimization!**

# Fast Exponentiation

```
exp: # a^n
    blez $a1, exp_cas_0
    and $t1, $a1, 1
    srl $a1, $a1, 1
    bneqz $t1, cas_impair

cas_pair:
    mul $v0 $a0 $a0
    # we do a^n = (a^2)^(n/2)
    j exp

exp_cas_0:
    li $v0, 1
    jr $ra
```

```
cas_impair:
    sub $sp, $sp, 8
    sw $ra, 0($sp)
    sw $a0, 4($sp)
    # we saved a and ra
    mul $v0 $a0 $a0
    jal exp
    #we restore a and ra
    lw $a0, 4($sp)
    lw $ra, 0($sp)
    add $sp, $sp, 8
    # we do (a^2)^(n/2) * a
    mul $v0, $v0, $a0
    jr $ra
```

# Can we always derecursify?

Not really...

# Should you program in assembly?

In real life: NO:

- harder to write
- harder to read
- faster code but compilers are good and improving the algorithm is usually a better idea

---

Here: **YES**

- learning how a CPU works
- understanding better the performance bottlenecks of high-level constructs
- useful for the compiler project :)

# Project P1

# Association map

**An associative structure is a data-structure that supports:**

- storing association from some key to some value
- removing an association
- get the value associated with some key
- testing whether some key has an association

**In OCaml you can implement them with:**

- simple list
- maps
- hashtable

# Association map

**An associative structure is a data-structure that supports:**

- storing association from some key to some value
- removing an association
- get the value associated with some key
- testing whether some key has an association

**In OCaml you can implement them with:**

- simple list simple and persistent but not efficient
- maps **efficient and persistent, for local variables**
- hashtable **efficient but not persistent, for global variables**

# Association map

**Association list:**

```
[] (* the empty association list *)

let nouv_l = ("key",42)::l
(* add a key value *)

let val = List.assoc "key" l
(* find the value for "key" *)

List.mem_assoc "key" l
(* tests if a value exists for "key" *)
```

# Association map

**Map:**

```
module StrMap = Map.Make(String)

StrMap.empty (* Empty association *)

StrMap.add "key" 42 myMap  (* add a mapping *)

StrMap.find "key" myMap  (* find the value *)

StrMap.mem "key" myMap  (* test the key presence *)
```

# Association map

**Hashtbl:**

```
let myT = Hashtbl.create 17

Hashtbl.add myT "key" 42

Hashtbl.find myT "key"

Hashtbl.mem myT "key"
```

# P1: the language

**A simple language:**

- local variables (introduced by let-in)
- global variables (introduced by read)
- simple arithmetical expressions (+,-,*,/)
- parenthesis
- print

**Possible extensions:**

- Dealing with optimization (storing sub-results in registers instead of the stack)
- Dealing with functions (one integer parameter, returning an int)

# P1: the language

```
print (let x = 10 in x) + (let x = 20 in let y = 30 in x+y)
print 100 / 2
print 2 / 100
read x
print x
read y
print y
print (let x = 10 in x) + (let x = x in let y = 30 in x+y)
read x
print x
read z
read x
print x
```

# P1: the language

```
print (let x = 10 in x) + (let x = 20 in let y = 30 in x+y)
print 100 / 2
print 2 / 100
read x
print x
read y
print y
print (let x = 10 in x) + (let x = x in let y = 30 in x+y)
read x
print x
read z
read x
print x
```

Let us look at the parser and lexer!

# P1: what to do?

- download the files from moodle
- fill out `compile.ml`
- add some tests and use `test_all.sh` to test your program!
- submit on moodle before the 26/09 at 18:00