λ

Control Structures and Abstraction - Part II

Programmazione Funzionale

2023/2024

Università di Trento

Chiara Di Francescomarino

Today

- Recap
- Higher order functions
- Functions as results

Recap

Call by value

- The value is the actual one (r-value) assigned to the formal parameter, that is treated like a local variable
- Transmission from `main` to `proc` ⇨
- Modifications to the formal parameter do not affect the actual one
- On procedure termination, the formal parameter is destroyed (together with the local environment)
- No way to be used to transfer information from the callee to the caller!

Call by reference (or variable)

- A reference (address) to the actual parameter (an expression with l-value) is passed to the function
- The actual parameter must be an expression with l-value
- References to the formal parameter are references to the actual one (aliasing)
- Transmission from and to `main` and `proc` ⇔
- Modifications to the formal parameter are transferred to the actual one
- On procedure termination the link between formal and actual is destroyed

Call by constant/read -only

- Read-only parameter method
- Similar to call by value but procedures are not allowed to change the value of the formal parameter (could be statically controlled by the compiler)
- Implementation could be at the discretion of the compiler (“large” parameters passed by reference, “small" by value)
- It can be thought as a sort of annotation
- In Java: `final`

```
void foo (final int x){ //x cannot be modified
```

- In C/C++: `const`

Call by result

- The actual parameter is an expression that evaluates to an l-value
- No link between the formal and the actual parameter in the body
- The local environment is extended with an association between the formal parameter and a new variable
- When the procedure terminates, the value of the formal parameter is assigned to the location corresponding to l-value of the actual parameter
- Output-only communication: no way to communicate from `main` to `proc` ⇦

Call by value-result

- Bidirectional communication using the formal parameter as a local variable ⇔
- The actual parameter is an expression that can yield an l-value
- At the call, the actual parameter is evaluated and the r-value assigned to the formal parameter.
- At the end of the procedure, the value of the formal parameter is assigned to the location corresponding to the actual parameter

Call by name

- A call to P is the same as executing the body of P after substituting the actual parameters for the formal one
- Bidirectional communication ⇔
- Copy-rule mechanism of the actual parameter to the formal one
  - Every time the formal parameter appears we re-evaluate the actual one
  - If the actual parameter is a variable, it is like passing it by reference
  - If it is an expression it is re-evaluated every time
- “Macro expansion", implemented in a semantically correct way

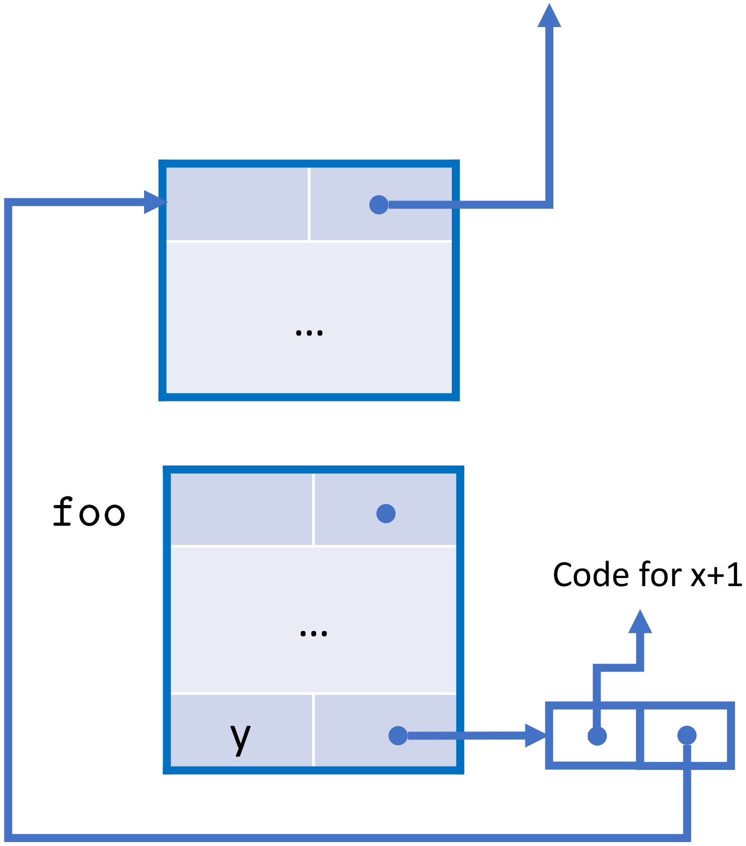
Call by name

- Appears to be simple but … it is not that simple: it has to deal with variables with the same name
- We pass the pair `<exp,env>` (closure)
  - A pointer to the text of `exp`
  - A pointer to the activation record of the calling block
- No longer used by any imperative language

How to implement the call by name?

- How do we pass the pair `<exp,env>` (closure)?
  - A pointer to the text of `exp`
  - A pointer to the activation record of the calling block
- This lets us pass functions as arguments to other procedures

```
int x=0;
int foo (name int y){
     int x = 2;
     return x + y;
}
...
int a = foo(x+1);
```



Call by value-result and reference vs call by name

```
void fiefoo (valueresult/reference
int x, valueresult /reference int
y) {
     x = x+1;
     y = 1;
}
...
int i = 1;
int[] A = new int[5];
A[1]=4;
fiefoo(i,A[i]);
```

```
void fiefoo (name int x,
name int y) {
     x = x+1;
     y = 1;
}
...
int i = 1;
int[] A = new int[5];
A[1]=4;
fiefoo(i,A[i]);
```

call- by value-result

- x is 1, y is A[1], i.e., 4
- x is 2
- y is 1
- i is 2, A[1] is 1

call- by reference

- x is 1, y is A[1]
- x and i are 2
- y and A[1] are 1
- i is 2, A[1] is 1

call- by name

- x is i, y is A[i]
- x and i are 2
- y and A[2] are 1
- i is 2, A[1] is 4, A[2]=1

Summing up

Call type	Direction	Link between formal and actual parameters			Actual parameter l-value?	Implementation
		Before	During	After		
Value	⇨	*			NO	Copy
Reference	⬄	*	*	*	YES	Reference
Constant	⇨	*			NO	Copy and/or reference
Result	⇦			*	YES	Copy
Value-result	⬄	*		*	YES	Copy
Name	⬄		Every time it appears		Can be	Closure

Exercise 5.1 *

- Say what will be printed by the following code fragment written in a pseudo-language which uses static scope; the parameters are passed by value.

```
{int x = 2;
 int fie(int y){
    x = x + y;
 }
 {int x = 5;
 fie(x);
 write(x);
 }
 write(x);
}
```

Solution exercise 5.1

- Say what will be printed by the following code fragment written in a pseudo-language which uses static scope; the parameters are passed by value.

```
{int x = 2;
 int fie(int y){
    x = x + y;
 }
 {int x = 5;
 fie(x);
 write(x);
 }
 write(x);
}
```

Solution: 5 7

Exercise 5.2 *

- Say what will be printed by the following code fragment written in a pseudo-language which uses dynamic scope; the parameters are passed by reference.

```
{int x = 2;
 int fie(reference int
y){
    x = x + y;
 }
 {int x = 5;
 fie(x);
 write(x);
 }
 write(x);
}
```

Solution exercise 5.2

- Say what will be printed by the following code fragment written in a pseudo-language which uses dynamic scope; the parameters are passed by reference.

```
{int x = 2;
 int fie(reference int
y){
    x = x + y;
 }
 {int x = 5;
 fie(x);
 write(x);
 }
 write(x);
}
```

Solution: 10 2

Exercise 5.3

- Say what will be printed by the following code fragment written in a pseudo-language which uses static scope; the parameters are passed by reference.

```
{int x = 2;
 void fie(reference int y){
     x = x + y;
     y = y + 1;
}
{int x = 5;
 int y = 5;
 fie(x);
 write(x);
}
write(x);
}
```

Solution exercise 5.3

- Say what will be printed by the following code fragment written in a pseudo-language which uses static scope; the parameters are passed by reference.

```
{int x = 2;
 void fie(reference int y){
    x = x + y;
    y = y + 1;
}
{int x = 5;
 int y = 5;
 fie(x);
 write(x);
}
write(x);
}
```

Solution: 6 7

Exercise 5.4

- Say what will be printed by the following code fragment written in a pseudo-language which uses static scope; the parameters are passed by value (a command of the form `foo(w++)` passes the current value of `w` to `foo` and then increments it by one).

```
{int x = 2;
 void fie(value int y){
    x = x + y;
}
{int x = 5;
 fie(x++);
 write(x);
}
 write(x);
}
```

Solution exercise 5.4

- Say what will be printed by the following code fragment written in a pseudo-language which uses static scope; the parameters are passed by value (a command of the form `foo(w++)` passes the current value of `w` to `foo` and then increments it by one).

```
{int x = 2;
 void fie(value int y){
    x = x + y;
}
{int x = 5;
 fie(x++);
 write(x);
}
 write(x);
}
```

Solution: 6 7

Exercise 5.5*

- State what will be printed by the following fragment of code written in a pseudo-language which uses static scope and call by name.

```
{int x = 2;
 void fie(name int y){
    x = x + y;
 }
 {int x = 5;
    {int x = 7}
    fie(x++);
    write(x);
 }
 write(x);
}
```

Solution exercise 5.5

- State what will be printed by the following fragment of code written in a pseudo-language which uses static scope and call by name.

```
{int x = 2;
 void fie(name int y){
    x = x + y;
 }
 {int x = 5;
    {int x = 7}
    fie(x++);
    write(x);
 }
 write(x);
}
```

Solution: 6 7

Exercise 5.6

- State what will be printed by the following code written in a pseudo-language which uses dynamic scope and call by reference.

```
{int x = 1;
 int y = 1;
 void fie(reference int z){
    z =x+y+z;
 }
 {int y = 3;
    {int x = 3;}
    fie(y);
    write(y);
 }
 write(y);
}
```

Solution exercise 5.6

- State what will be printed by the following code written in a pseudo-language which uses dynamic scope and call by reference.

```
{int x = 1;
 int y = 1;
 void fie(reference int z){
    z =x+y+z;
 }
 {int y = 3;
    {int x = 3;}
    fie(y);
    write(y);
 }
 write(y);
}
```

Solution: 7 1

Exercise 5.7*

- State what will be printed by the following code written in a pseudo-language which uses static scope and call by reference.

```
{int x = 0;
 int A(reference int y) {
     int x =2;
     y=y+1;
     return B(y)+x;
}
int B(reference int y){
     int C(reference int y){
          int x = 3;
          return A(y)+x+y;
     }
     if (y==1) return C(x)+y;
     else return x+y;
 }
 write (A(x));
}
```

Solution exercise 5.7

- State what will be printed by the following code written in a pseudo-language which uses static scope and call by reference.

```
{int x = 0;
 int A(reference int y) {
     int x =2;
     y=y+1;
     return B(y)+x;
 }
 int B(reference int y){
     int C(reference int y){
          int x = 3;
          return A(y)+x+y;
     }
     if (y==1) return C(x)+y;
     else return x+y;
 }
 write (A(x));
}
```

Solution: 15

Exercise 5.8*

- State what will be printed by the following code fragment written in a pseudo-language permitting reference parameters (assume Y and J are passed by reference).

```
int X[10];
int i = 1;
X[0] = 0;
X[1] = 0;
X[2] = 0;
void foo (reference int Y,J){
    X[J] = J+1;
    write(Y);
    J++;
    X[J]=J;
    write(Y);
}
foo(X[i],i);
write(X[i]);
```

Solution exercise 5.8

- State what will be printed by the following code fragment written in a pseudo-language permitting reference parameters (assume `Y` and `J` are passed by reference).

```
int X[10];
int i = 1;
X[0] = 0;
X[1] = 0;
X[2] = 0;
void foo (reference int Y,J){
    X[J] = J+1;
    write(Y);
    J++;
    X[J]=J;
    write(Y);
}
foo(X[i],i);
write(X[i]);
```

Solution: 2 2 2

Exercise 5.9*

- State what will be printed by the following code fragment written in a pseudo-language which allows value-result parameters.

```
int X = 2;
void foo (value-result int Y){
    Y++;
    write(X);
    Y++;
 }
foo(X);
write(X);
```

Solution exercise 5.9

- State what will be printed by the following code fragment written in a pseudo-language which allows value-result parameters.

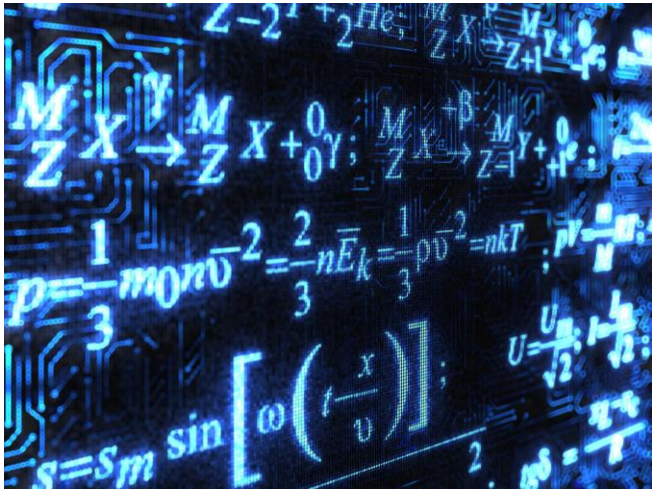
```
int X = 2;
void foo (value-result int Y){
    Y++;
    write(X);
    Y++;
 }
foo(X);
write(X);
```

Solution: 2 4

Higher-order functions

Higher-order functions

- Some language allow
  - passing functions as arguments to procedures
  - returning functions as results of procedures
- In both cases: How do we manage the environment?
- Functions as parameters
  - Simplest case
  - Use a pointer to the activation record in the stack
- Functions as results
  - More complex
  - We must maintain an activation record of the resulting function, but not on the stack



Functions as parameters

Functions as parameters

- A function is passed as a parameter to another function and then called through the actual parameter
- Call by name is a special case of functions as parameters
  - Use a function without arguments

```
{int x = 1;
 int f(int y){
    return x+y;
 }
 void g (int h(int b)){
    int x = 2;
    return h(3) + x;
 }
 ...
 {int x = 4;
  int z = g(f);
 }
}
```

- Three declarations of `x`
- When `f` is called via `h`, which `x` is used?
  - Static scoping: the external `x`
  - Dynamic scoping: both of them would make sense … which one?

Binding rules

```
{int x = 1;
 int f(int y){
     return x+y;
 }
void g (int h(int b)){
     int x = 2;
     return h(3) + x;
}
...
 {int x = 4;
  int z = g(f);
 }
}
```

Shallow binding: environment at this point in time

Deep binding: environment at this point in time

- When a procedure is passed as a parameter, this creates a reference between a name (formal, `h`) and a procedure (actual `f`)
- Which non-local environment applies when `f` is executed, called via `h`?
  - Environment at the moment of creation of the link (deep binding): always used with static scoping
  - Environment at the moment of the call of the function passed as parameter (shallow binding): Can be used with dynamic scoping

Binding policy and scope policy

- Binding policy is independent from scope policy
- Static scoping
  - Deep binding
- Dynamic scoping
  - Deep binding
  - Shallow binding

```
{int x = 1;
 int f(int y){
     return x+y;
 }
void g (int h(int b)){
     int x = 2;
     return h(3) + x;
}
...
 {int x = 4;
  int z = g(f);
 }
}
```

Static and deep	Dyn and deep	Dyn and shallow
x is 1	x is 4	x is 2
h(3) returns 4	h(3) returns 7	h(3) returns 5
z is 6	z is 9	z is 7

Deep binding implementation

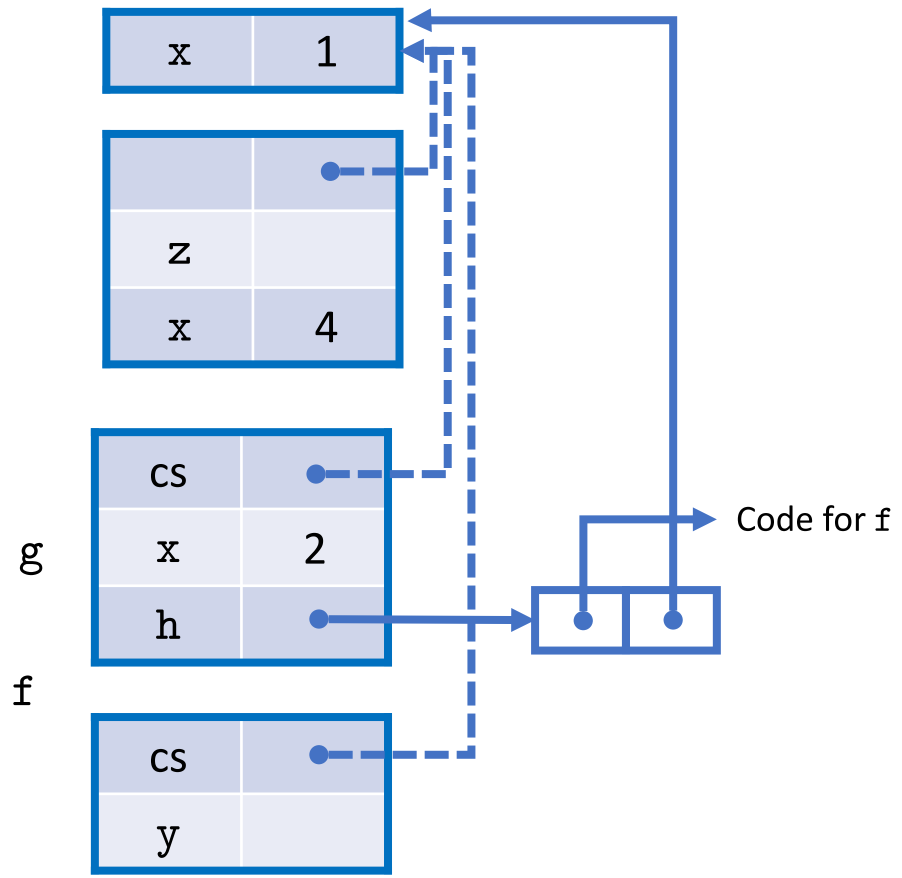
- Shallow binding in dynamic scope
  - No need to do anything special
    - To access `x` use the stack
    - Use the standard structures (A-list, CRT)
- Deep binding is more complex (e.g., static scoping)
  - We need to use some form of closure to freeze the scope so that it can be reactivated later
  - We need to retrieve the environment at the time of the creation of the association
  - We can statically associate the information about the nesting level of the declaration of `f` in the block in which `g(f)` is called
  - At the time of the call, both the code for f and a pointer to the activation record in which f is declared are associated to the formal parameter

Closure (static scoping)

- Both the link to the code of the function, as well as its nonlocal environment are passed to the function
- The procedure passed as parameter
  - Allocates the activation record
  - Takes the pointer to the static chain of the closure

Deep binding implementation (static scoping)

```
{int x = 1;
 int f(int y){
     return x+y;
 }
 void g (int h(int b)){
     int x = 2;
     return h(3) + x;
 }
...
 {int x = 4;
  int z = g(f);
 }
}
```



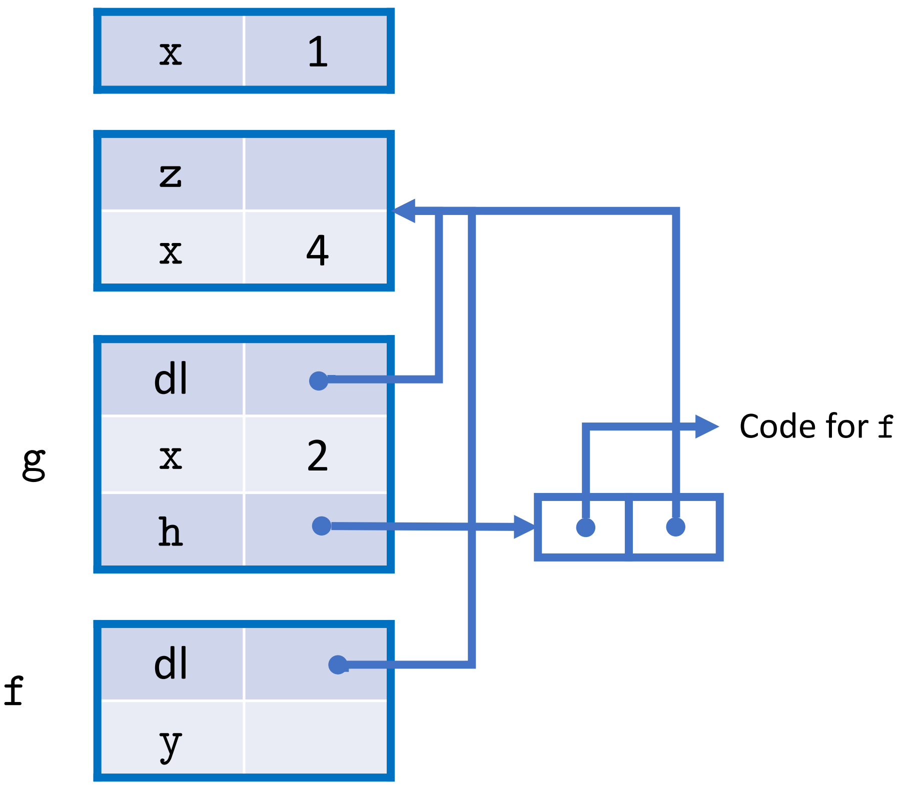
- When g is called with actual parameter f, a closure is linked to `h`.
- f is declared at distance 1 (SD(f)=1, SD(call)=1) from the place in which it appears as actual parameter → one step along the static chain
- When `f` is called through the name h, the corresponding activation record is pushed into the stack. The values of the static chain pointer is taken from the second component of the closure

Deep binding (static scoping)

- Closure to maintain pointers to the code and to the nonlocal environment
- When called, the pointer to the static chain is determined via the closure
- Activation records can be “jumped over” to access nonlocal variables through the static chain
- Deallocation of activation records using stack LIFO

Deep binding with dynamic scoping

```
{int x = 1;
 int f(int y){
     return x+y;
 }
 void g (int h(int b)){
     int x = 2;
     return h(3) + x;
 }
...
 {int x = 4;
  int z = g(f);
 }
}
```



Deep vs shallow binding

- Dynamic scope
  - Possible with deep binding
    - Implementation with closure
  - Or shallow binding
    - No special implementation needed
- Static scope
  - Always uses deep binding
    - Implemented with closure
  - At first glance no difference between deep and shallow binding
    - The static scoping rules determine which nonlocal value to use
  - That is not the case: There may be dynamically several instances of a block that declare a non-local name (for example with recursion)

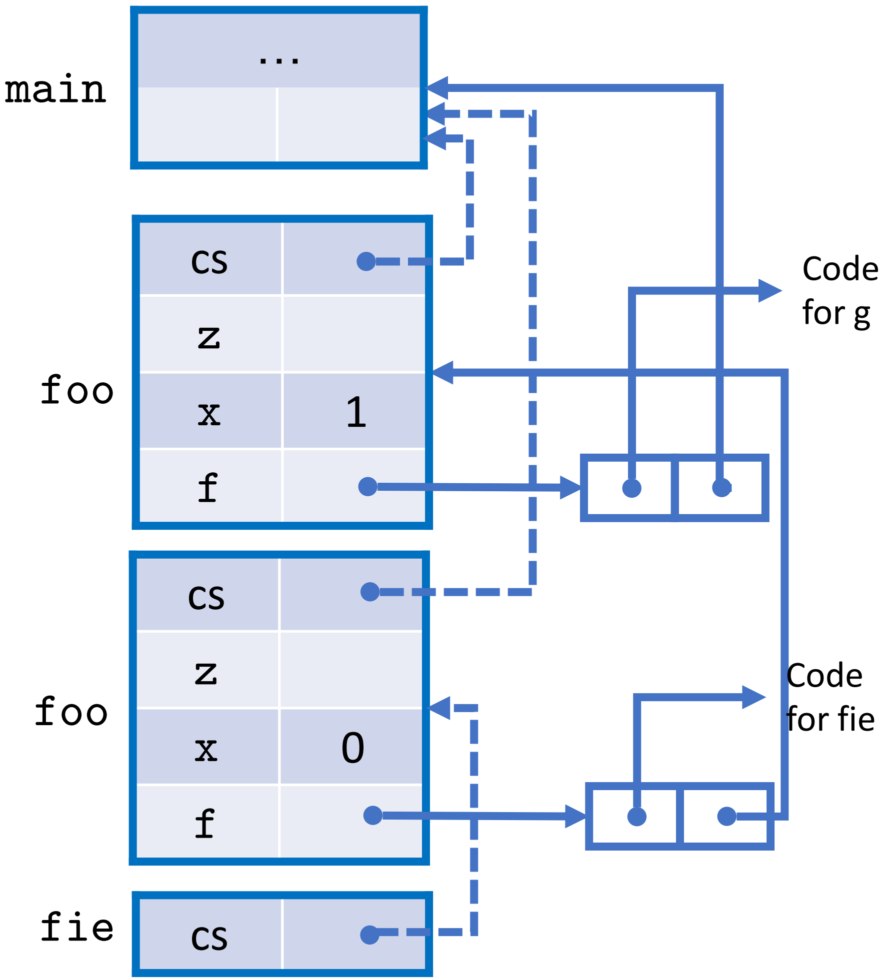
An example

```
{
 void foo (int f(), int x){
      int fie(){
          return x;
      }
      int z;
      if (x==0) z=f();
          else foo(fie,0);
  }
  int g(){
     return 1;
  }
  foo(g,1);
}
```

- Static scoping
  - `x` inside `fie` refers to formal parameter of `foo`
- However when fie is called there are two active instances of foo (and environment)
  - `foo(g,1)` with `x=1`
  - `foo(fie,0)` with `x=0`
- Deep binding: when the association between `fie` and `f` is created
  - `x=1`
- In case of shallow binding (which is not possible), the environment would be determined at the time `f` is invoked and `z` would be assigned 0.

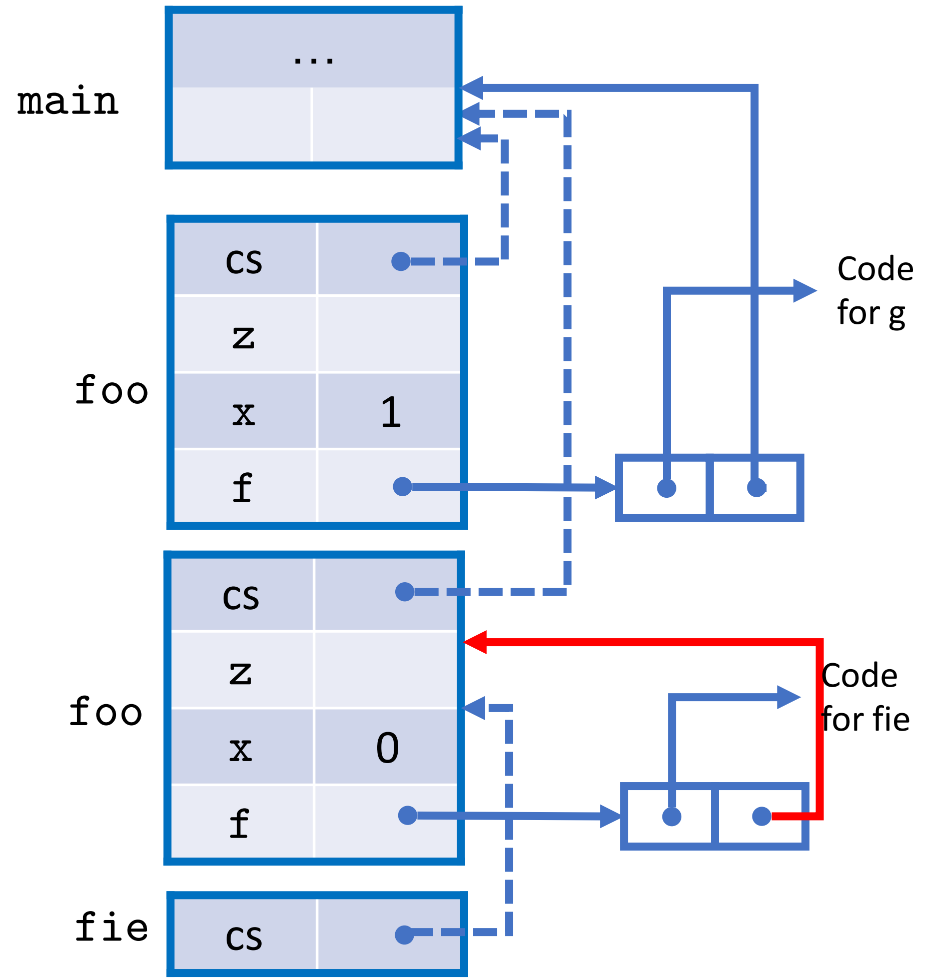
An example

```
{
 void foo (int f(), int x){
      int fie(){
          return x;
      }
      int z;
      if (x==0) z=f();
      else foo(fie,0);
  }
  int g(){
     return 1;
  }
  foo(g,1);
}
```



In case of shallow binding …

```
{
 void foo (int f(), int x){
      int fie(){
          return x;
      }
      int z;
      if (x==0) z=f();
      else foo(fie,0);
  }
  int g(){
     return 1;
  }
  foo(g,1);
}
```



However, it is not allowed with static scoping!

What defines the environment

- Visibility rules (based on the block structure)
- Exceptions to the visibility rules (e.g., usage of a name before its declaration)
- Scoping rules
- Rules for the parameter passing
- Binding policy

Functions as results

Functions as results

- Generating functions as the result of other functions allows the dynamic creation of functions at runtime

```
{int x = 1;
 void->int F () {
     int g () {
          return x+1;
     }
     return g;
 }
 void->int gg = F();
 int z = gg();
}
```

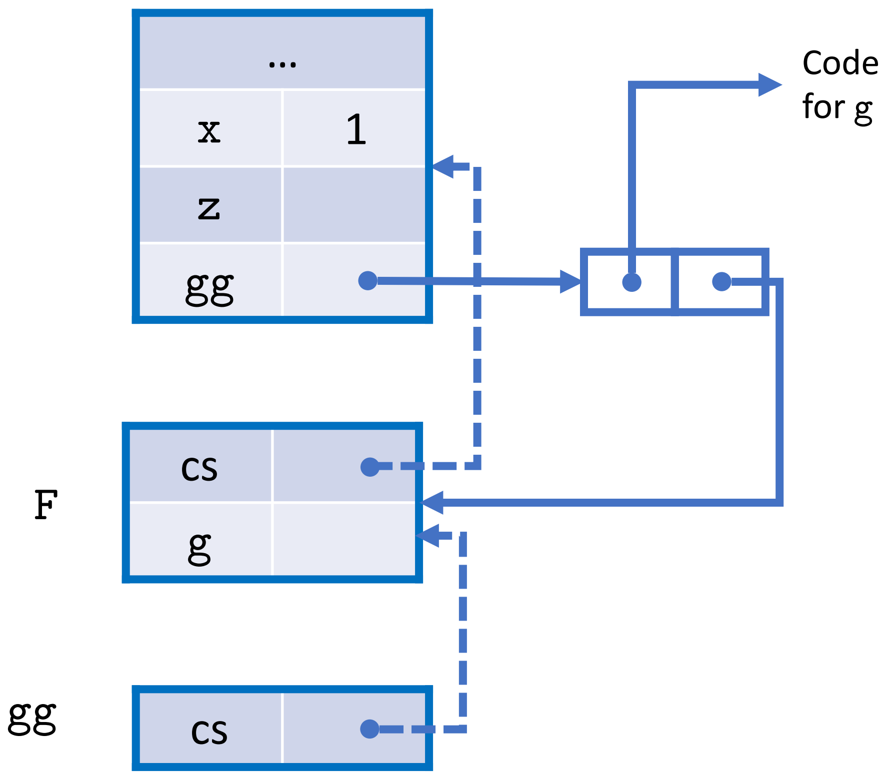
- **`void-> int`** denotes the type of the functions that take no argument and return an int
- **`void->int`** `F()` is the declaration of a function which returns a function of no argument and return value int
- **`return g`** returns the function and not its application
- gg is dynamically associated with the result of the evaluation of `F`
- The function gg returns the successor of the value of `x`

Returning a function: simple case

- A function returned as a result requires besides its code the environment in which the function will be evaluated.
- When a function returns a function as result, the result is a closure
- Manage a “call by closure”: the static chain pointer of the activation record is determined using the associated closure (and not the canonical rules)

In the example

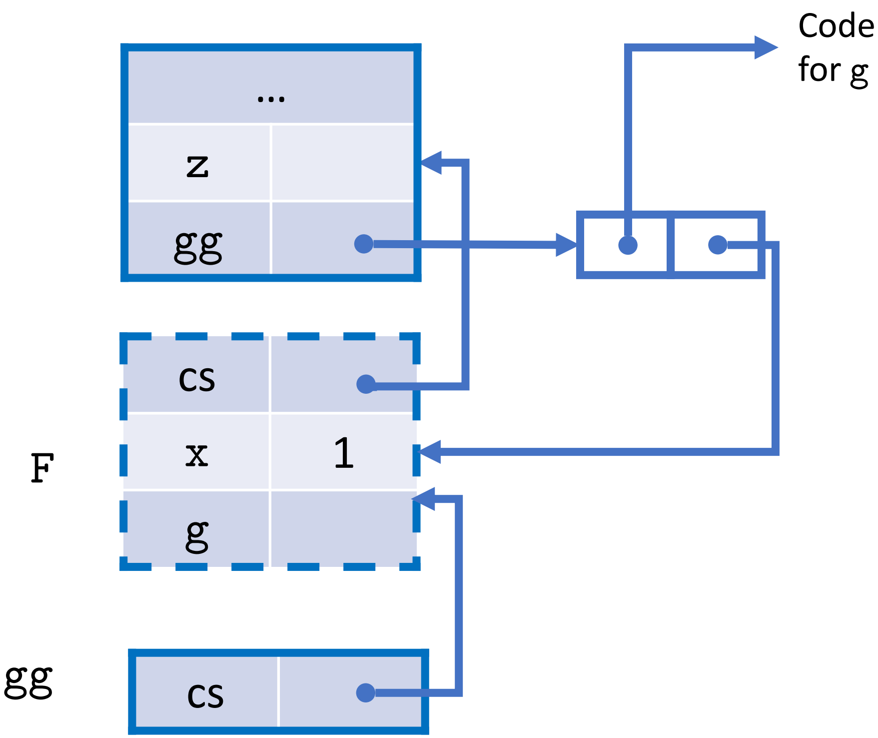
```
{int x = 1;
 void->int F () {
     int g () {
          return x+1;
     }
     return g;
 }
 void->int gg = F();
 int z = gg();
}
```



- Using the static scope regime, x is fixed by the structure of the program and not by the position of the call to gg, which could appear in an environment in which another definition of the name x occurs

Complex case

```
void->int F () {
  int x = 1;
  int g () {
      return x+1;
  }
  return g;
}
void->int gg = F();
int z = gg();
```



- When the result of F() is assigned to gg, the closure points to the environment local to F() that contains x
- The environment of F() is destroyed after its termination

How to solve the issue?

- Use closure
- But the activation record has to remain forever
  - The stack LIFO property fails
- How do we then implement a "stack"?
  - No automatic deallocation
  - Activation record on heap
  - Static or dynamic chain connects the record
  - Call garbage collector when needed

- In imperative languages
  - Several restrictions to avoid creating a reference to an environment that has been deactivated

In ML

- An environment (except the top-level) in ML:
  - A list of name-value pairs (the bindings in the environment)
  - A pointer to the parent environment (except for the case of the top-level environment) that refers to the textually enclosing environment

Referencing external variables

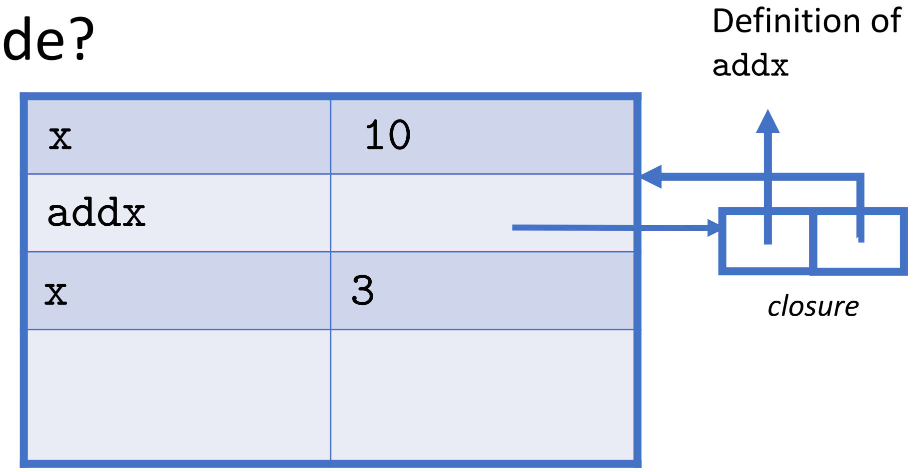
- What is the result of this code?

```
> val x=3;
> fun addx(a) = a+x;
> val x=10;
> addx(2);
```

Referencing external variables

- What is the result of this code?

```
> val x=3;
> fun addx(a) = a+x;
> val x=10;
> addx(2);
```



- The value of x is the value when the function is defined

```
> addx(2);
val it = 5: int
```

Summary

- Higher order functions
- Functions as results

Readings

- Chapter 6 and 7 of the reference book
  - Maurizio Gabbrielli and Simone Martini "Linguaggi di Programmazione - Principi e Paradigmi", McGraw-Hill

Next time

- Data types and abstract data types