Names and Environments

Programmazione Funzionale

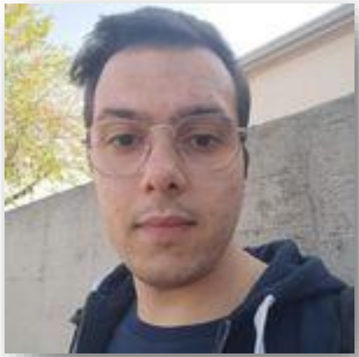
2023/2024

Università di Trento

Chiara Di Francescomarino

Tutoring

- Tue morning 10:30 – 11:30?



Matteo Mariotti

Today

- Names and Denotable Objects
- Environments and Blocks
- Scope Rules
  - Static scoping
  - Dynamic scoping
  - Special cases

Names and Denotable Objects

What is a name?

- Sequence of characters used to *denote* something else
  - `const p = 3.14` (object denoted: the constant 3.14)
  - `x` (object denoted: a variable)
  - `void f()(...)` (object denoted: the definition of `f`)
- In programming languages, the names are often identifiers (alphanumerical tokens)
- The use of a name serves to indicate the object to denote
  - Symbolic identifiers, that are easier to remember
  - Abstractions (data or control)

Names and denotable objects

- A name and the object it denotes are not the same thing.
  - A name is just a character string
  - Denotation can be a complex object (variable, function, type, etc.)
  - A single object can have more than one name (“aliasing”)
  - A single name can denote different objects at different times
- We use "the variable `fie`" or "the function `foo`" as abbreviations for "the variable with the name `fie`" and "the function with the name `foo`"

What is a denotable object?

- An object that can be associated with a name
- Names defined by the user: variables, formal parameters, procedures, types defined by the user, labels, modules, constants defined by the user, exceptions
- Names defined by the language: primitive types, primitive operations, predefined constants
- Binding: association between name and denotable object

Bindings: when are they made?

- Language design time: bindings between primitive constants, types and operations of the language
  - e.g., + means addition, `int` denotes the type of integers
- Program writing time: partial definition of some bindings completed later
  - e.g., the binding of an identifier to a variable, is defined in the program but is only created when the space for the variable is allocated in memory
- Compile time: the compiler allocates memory space for some of the data structures that can be statically processed. The connection between an identifier and the corresponding memory location is formed at this time.
  - e.g., global variables

Static

Execution

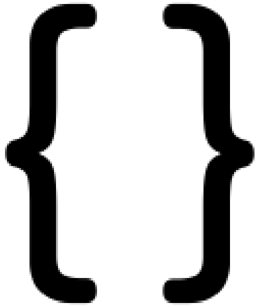
Dynamic

- Runtime: All the associations that have not been previously created must be formed at runtime.
  - e.g., bindings of variable identifiers to memory locations for the local variables in a recursive procedure, or for pointer variables whose memory is allocated dynamically.

Static and dynamic

- "static": everything that happens prior to execution
- "dynamic": everything that happens during execution.
- For example, static memory management is performed by the compiler, while dynamic management is performed by appropriate operations executed at runtime.

Environments and blocks



Environment

Environment

- Not all the associations between names and denotable objects are fixed once and for all.
- Environment: the collection of associations between names and denotable objects that exist at runtime at a specific point in a program and at a specific moment in the execution
- Declaration: a mechanism (implicit or explicit) that creates an association in an environment.

```
int x;
int f(){
    return 0;
}
type T = int;
```

Variable declaration

Function declaration

Type declaration

Names and denoted objects

- The same name can denote different objects at different positions in the program

```
{int fie;
 fie = 2;
     {char fie;
          fie = a;
     }
}
```

Integer

Char

- A single object is denoted by more than one name in different environments
  - A variable passed by reference to a procedure

```
procedure P (var &X:integer);
begin
...
end;

var A:integer;
P(A);
```

X

A

Names and denoted objects

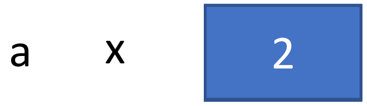
- Aliasing: different names that denote the same object in the same environment
  - Call by reference visible within the procedure

```
int a;
int fie(int &x){
     a = 1;
     printf("a = %d",a);
     printf("x = %d",x);
     x = 2;
     printf("a = %d",a);
     printf("x = %d",x);
     return 0;}
int main(){
     a = 0;
     fie(a);
     printf("a = %d",a);
     return 0;}
```

```
1
1
```

```
2
2
```

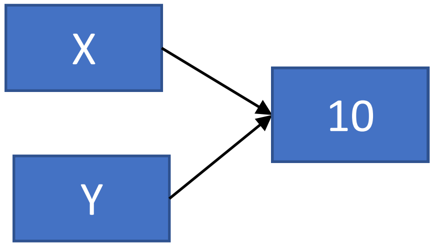
```
2
```



Names and denoted objects

- Aliasing: different names that denote the same object in the same environment
  - Call by reference visible within the procedure
  - Pointers (integer pointers)

```
int *X, *Y; // X,Y pointers to integers
X = (int *) malloc (sizeof (int)); // allocate heap
memory
*X = 5; // * dereference
Y = X; // Y points to the same object as X
*Y = 10;
write(*X);    10
```

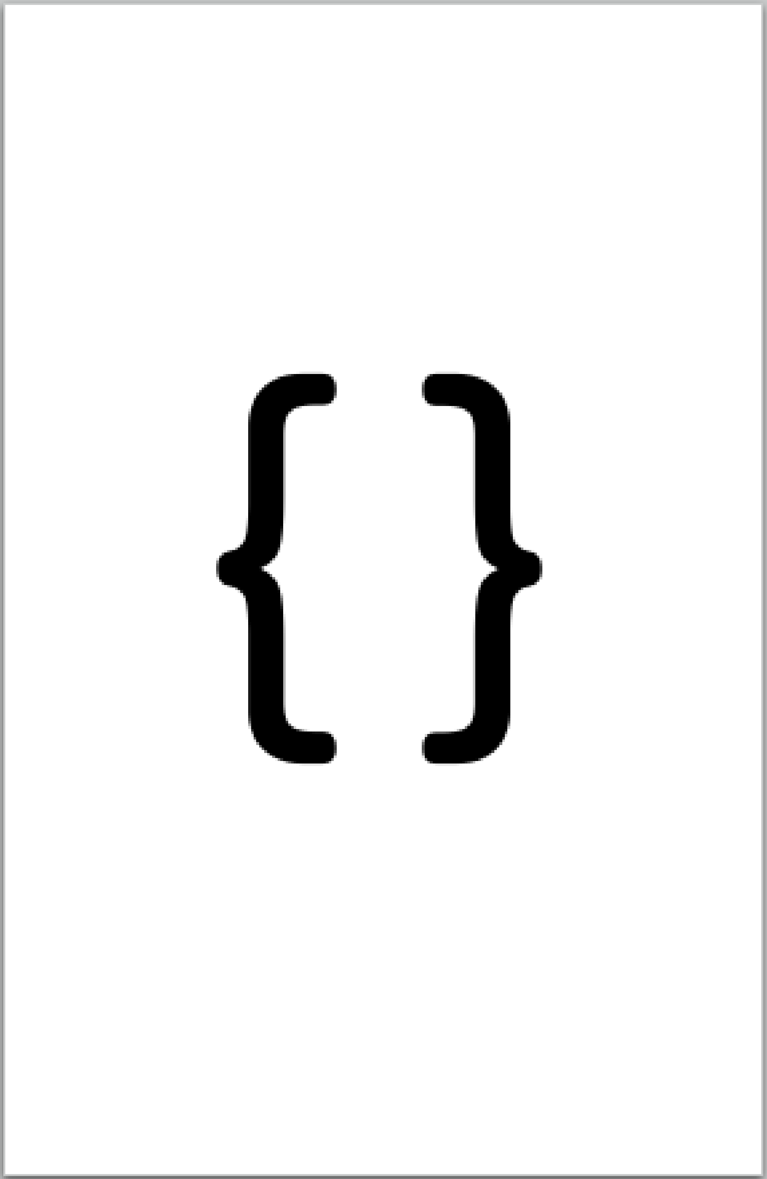


Names and denoted objects

- Aliasing: different names that denote the same object in the same environment
  - Call by reference visible within the procedure
  - Pointers (integer pointers)

```
int *X, *Y; // X,Y pointers to integers
X = (int *) malloc (sizeof (int)); // allocate heap
memory
*X = 5; // * dereference
Y = X; // Y points to the same object as X
*Y = 10;
write(*X);
```

- A single name can denote different objects in a single textual region of the program, according to the execution flow of the program
  - Recursive procedure declaring a local name



Blocks

Blocks

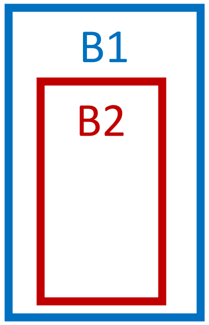
- In modern programming languages, the program can be structured – a block is a key concept for the environment organization
- Block: a section of the program identified by opening and closing signs that contains declarations local to that region
  - `Algol`, Pascal: `begin … end`
  - C, Java: `{ … }`
  - LISP: `( … )`
  - ML: let … in … end
- We distinguish between
  - Block associated with a procedure: body of the procedure with local declarations
  - In-line (anonymous) block: it can appear in any position in which a command appears

Advantages of blocks

- Local use of names
  - Increases clarity
  - Users can choose the names that they prefer
- With an appropriate allocation of memory (see later)
  - Optimizes the use of memory
  - Permits the use of recursion

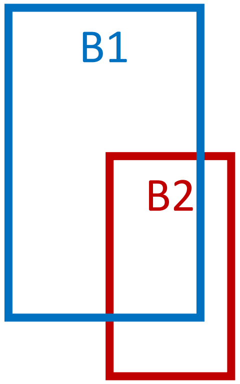
Nesting

- Blocks can (for some languages) be nested



```
int x;
int f(){
    return 0;
}
type T = int;
```

✓



```
open block A;
    open block B;
close block A;
    close block B;
```

✗

- Procedure nesting is not allowed by C but allowed by Pascal and Ada

Visibility of rule declarations among blocks

- Rules of visibility (preliminary)
  - A local declaration in a block is visible in this block, and in all nested blocks, as long as these do not contain another declaration of the same name, that hides or masks the previous declaration.

- Associations declared in an outer block are visible internally but deactivated if redeclared in the internal block
- Associations introduced within a block are not visible outside
- Associations introduced within a block are not visible by sibling or other blocks not containing this block

Subdivision of the environment

- Block: the construct of least granularity to which a constant environment can be associated.
- The environment in a block can be subdivided in:
  - Local environment: associated with the entry into a block
    - Local variables
    - Formal parameters
  - Non-local environment: associations inherited from other blocks external to the current one
  - Global environment: part of the non-local environment that contains associations common to all blocks
    - Explicit declarations of global variables

Example

```
A:{int a =1;
    B:{int b = 2;
       int c = 2;
        C:{int c =3;
           int d;
           d = a+b+c;
           write(d)
        }
        D:{int e;
           e = a+b+c;
        write(e)
        }
    }
}
```

Block A – a: global environment – assuming block A as main

Block B – b,c: local environment
a: global environment

Block C – c,d: local environment
a: global environment
b: non-local environment (B)

6 (d = 1+2+3)

Block D – e: local environment
a: global environment
b,c: non-local environment (B)

5 (e = 1+2+2)

What happens when a block …

- … is entered?
  - Associations between locally declared names and corresponding denotable objects are created
  - Associations with names declared external to and redefined within the block are deactivated
- … is exited?
  - Associations for names declared locally to the block and the objects are destroyed
  - Associations between names that existed external to the block and which were redefined inside it are reactivated

Operations in an environment

- Creation of associations between a name and a denotable object (naming)
  - Local declaration in a block or connection of formal and actual parameter
- Reference to a denoted object via its name (referencing)
  - Usage of the name
- Deactivation of associations between a name and a denoted object
  - Because of the entrance in a block with a declaration that masks the previous one (the old association is just inactive)
- Reactivation of an association between a name and a denotable object
  - Exit from a block that masked the association
- Destruction of an association between a name and a denotable object (unnaming)
  - Exit from a block with a local declaration

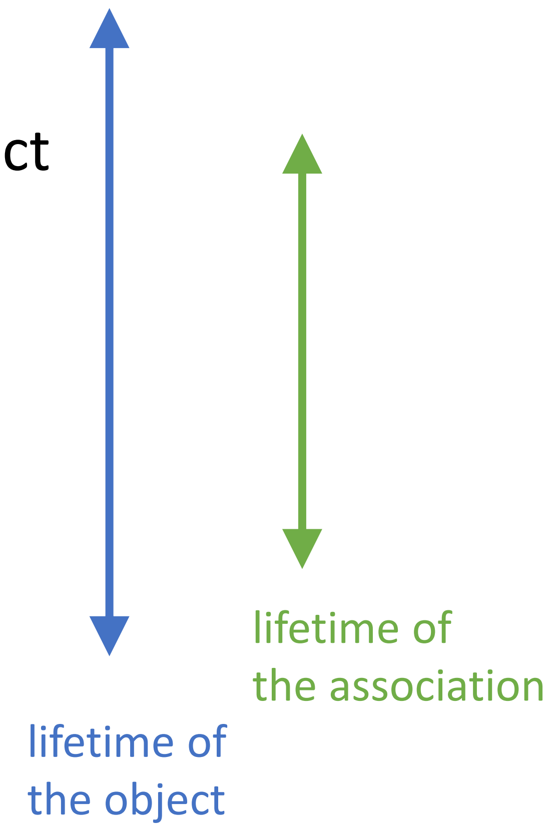
Operations on denotable objects

- Creation
  - Memory allocation – sometimes it also includes the initialization
- Access
  - Reference and usage of the object
- Modification
  - Value modification
- Destruction
  - Reallocation of the reserved memory
- Note that creation and destruction of an object are not the same as creation and destruction of links to it

For example

Let us consider the following events:

1. Creation of an object
2. Creation of an association for the object
3. Reference to an object
4. Deactivation of an association
5. Reactivation of an association
6. Destruction of an association
7. Destruction of an object



Lifetimes

The lifetime of an object does not necessarily coincide with the lifetime of an association for that object

- The lifetime of an object can be longer than that of an association, e.g., a variable passed to a procedure by reference

```
procedure P (var X:integer);
begin
...
end;

var A:integer;
P(A);
```

During the execution of P, there exists an association between X and an object (an integer) that exists before and after this association.

Lifetimes

The lifetime of an object does not coincide with the lifetime of an association for that object

- The lifetime of an object can be shorter than the one of an association, e.g., a region of memory dynamically deallocated

```
int *X, *Y;
X = (int *) malloc (sizeof(int));
Y=X;

free(X);
X=null;
```

After the `free` command, the object no longer exists, but there is still a **dangling** reference `Y` to it.

Scope rules

How can we interpret the rules of visibility?

- A local declaration in a block is visible in that block and in all nested blocks, as long as no intervening block contains a new declaration of the same name (that hides the previous one)

Actually this definition is not that clear!

Which value will be printed?

```
A:{int x = 0;
   void fie(){
        x = 1;
   }
   B:{int x;
        fie();
   }
   write(x);
}
```

static

1
since `fie()` is defined in `Block A`

dynamic

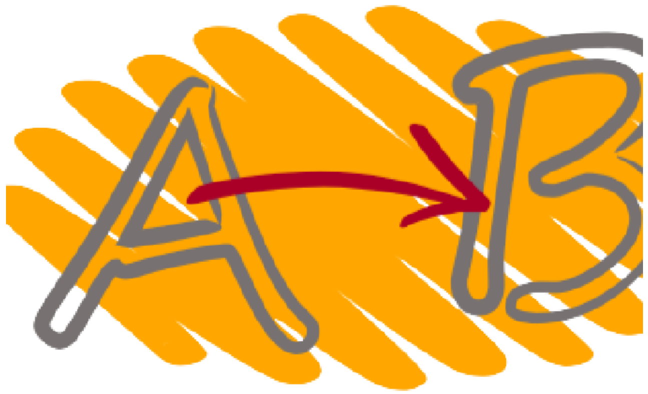
0
since `fie()` is called in `Block B`

It depends on the scope rule used

Scope rule

A non-local reference in a block can be resolved by

- Static scoping: in the block that syntactically includes the block
- Dynamic scoping: in the block that is executed immediately before the block



Static (lexical) scoping

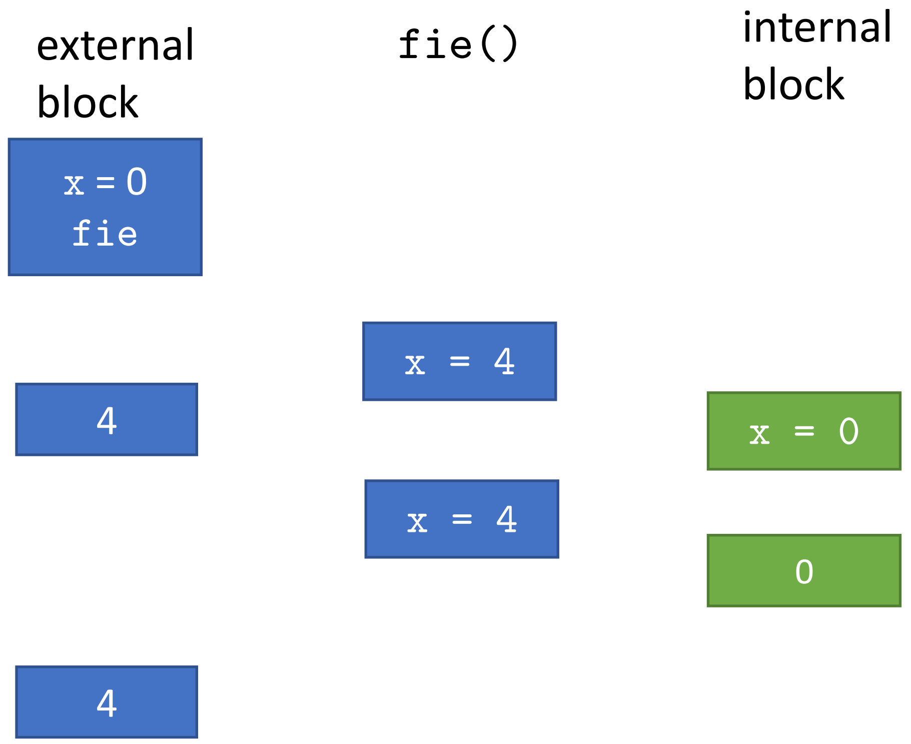
Static scope rule

1. The declarations local to a block define the local environment of that block
2. If a name is used inside a block, the valid association is the one present in the environment local to the block, if it exists; otherwise the association is the one existing in the local environment in the block containing the block. If no association is found for the name moving from the nearest to the furthest block, the association is looked for in the predefined environment and, if not found, an error is retuned
3. A block can be assigned a name, which becomes part of the local environment of the block immediately including the considered block.

Static scoping

- In static scoping a non-local name is resolved in the block that textually includes it

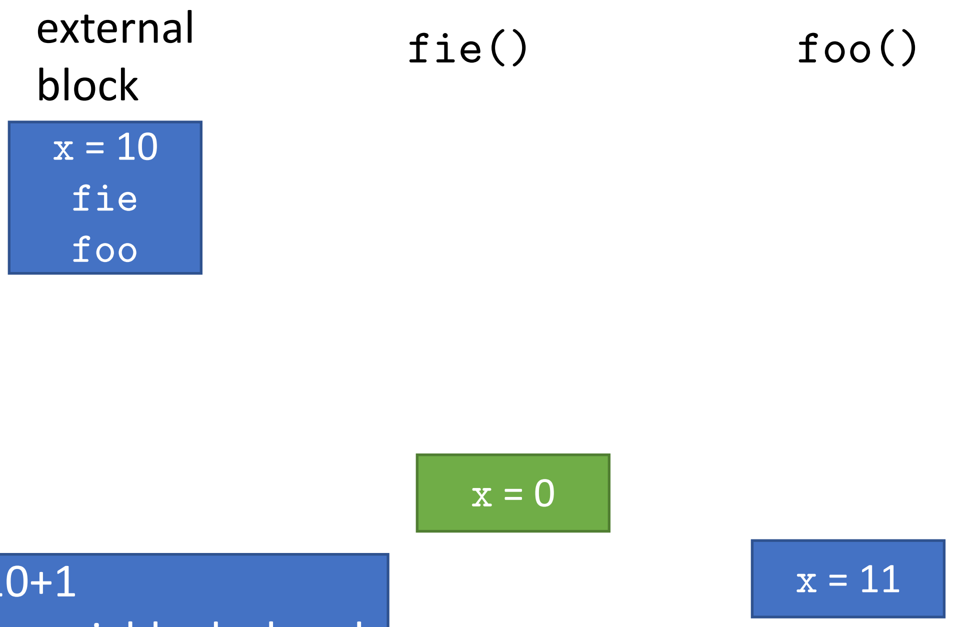
```
{int x=0;
 void fie (int n){
     x = n+1;
 }
 fie(3);
 write (x);
     {int x =0;
      fie(3);
      write (x);
      }
  write (x);
}
```



Static scoping: independent of the position

- The body of `foo` is part of the scope of the most external occurrence of `x`
- The call to `foo` is part of the scope of the most internal occurrence of `x`
- Static scoping is independent of the position in which the function is called.

```
{int x=10;
 void foo(){
    x++;
 }
 void fie() {
    int x = 0;
    foo();
 }
 fie();
}
```



x=10+1
since `x` refers to the variable declared in the block containing `foo`

Static scoping: independent of local names

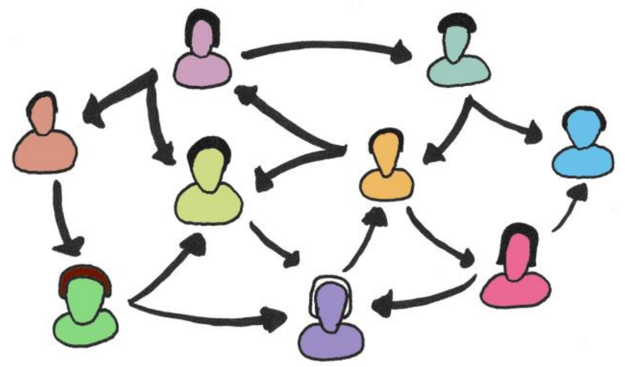
```
{int x=10;
 void foo(){
    x++;
 }
 void fie() {
    int y = 0;
    foo();
 }
 fie();
}
```

- Changing the local declaration from `x` to `y` in `fie`, has no effect with the static scoping (independent of local names).

`x=10+1`
since `x` refers to the variable declared in the block containing `foo`

Advantages of static scoping

- It allows for the determination of all the environments present in a program simply by reading its text.
  1. The programmer has a better understanding of the program
  2. The compiler can connect the name occurrences with its correct declaration, thus enabling several correctness tests and code optimisations
- The compiler has some important information about the storage of the variables (in particular it knows the offsets relative to a fixed position) making more efficient the execution (w.r.t. the dynamic scoping)
- The compiler cannot know in general which memory location will be assigned to the variable with name `x`
- ALGOL, Pascal, C, C++, Ada, Scheme, ML and Java use some form of static scope.

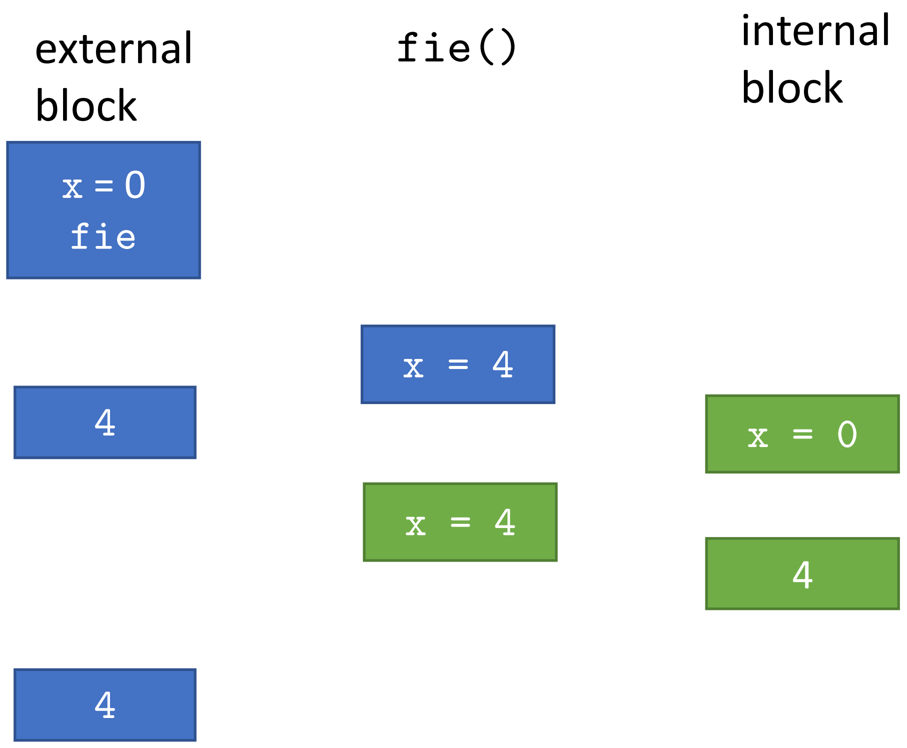


Dynamic scoping

Dynamic scoping

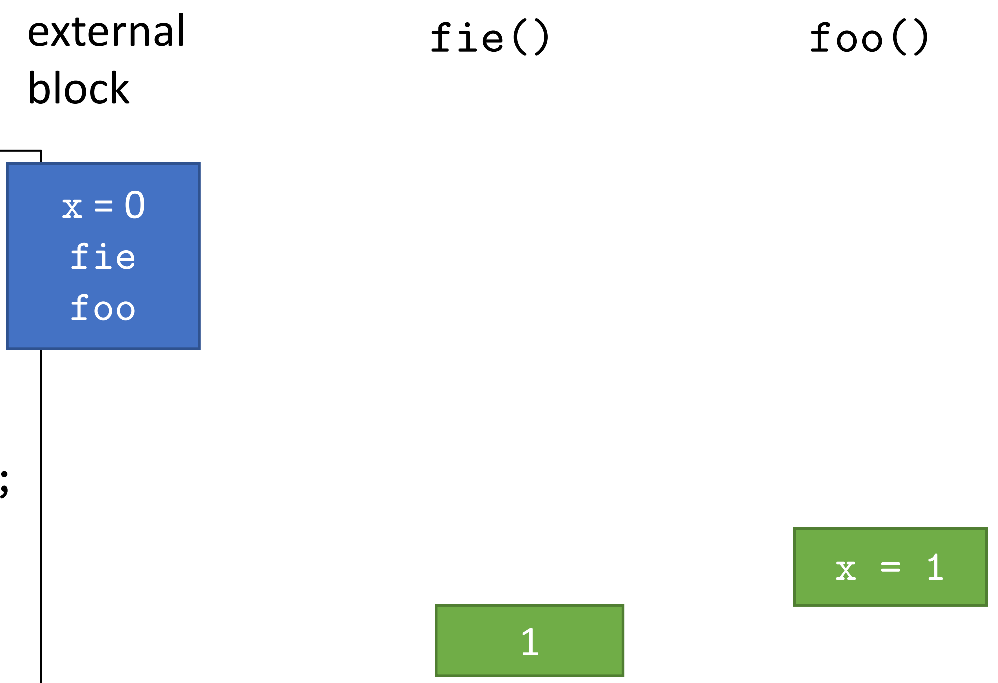
- In dynamic scoping a non-local name is resolved in the block that has been most recently activated and has not yet been deactivated

```
{int x=0;
 void fie (int n){
     x = n+1;
 }
 fie(3);
 write (x);
     {int x=0;
      fie(3);
      write (x);
      }
  write (x);
}
```



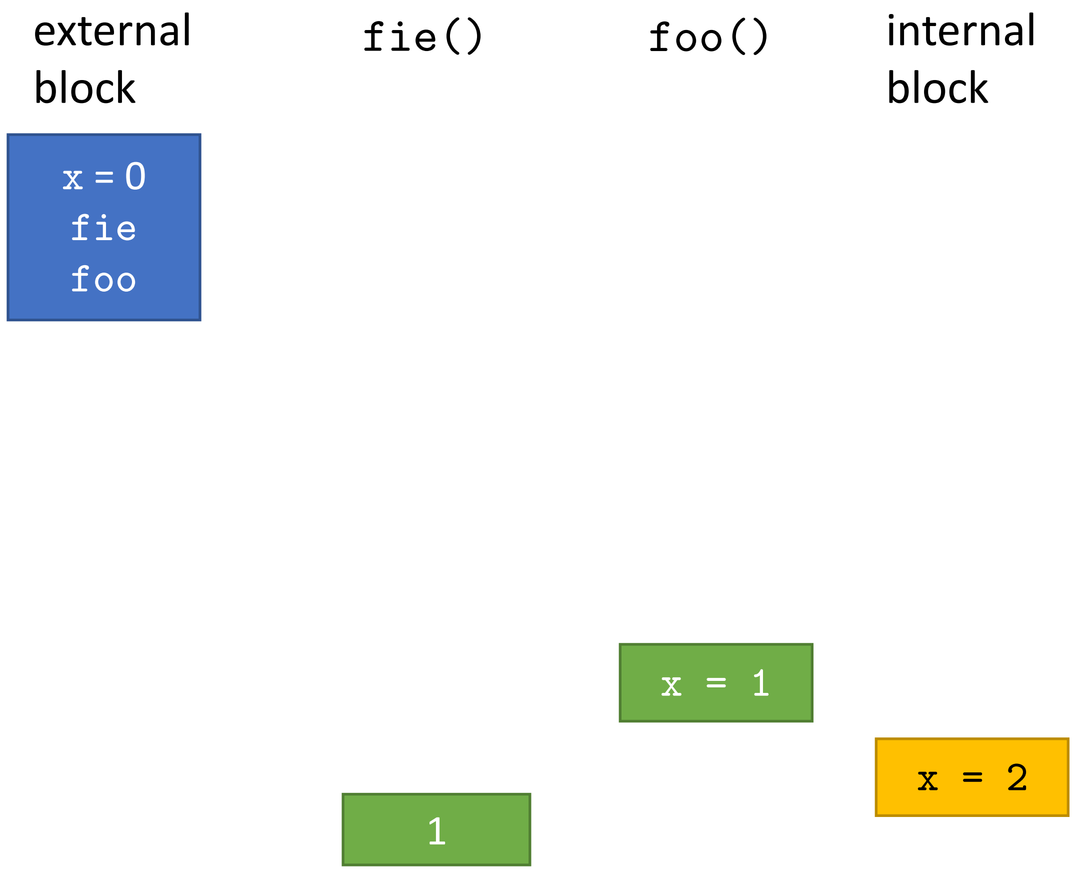
Dynamic scoping: another example

```
{const int x = 0;
 void fie(){
    write(x);
 }
 void foo(){
    const int x =1;
    fie();
 }
 foo();
}
```



Dynamic scoping

```
{const int x = 0;
 void fie(){
    write(x);
 }
 void foo(){
    const int x = 1;
        {const int x = 2;
        }
    fie();
 }
 foo();
}
```



Dynamic scoping: specializing a function

- Dynamic scope allows the modification of the behaviour of procedures or subprograms without using explicit parameters but only by redefining some of the non-local variables used
- `visualize` is a function that displays a text in a certain colour

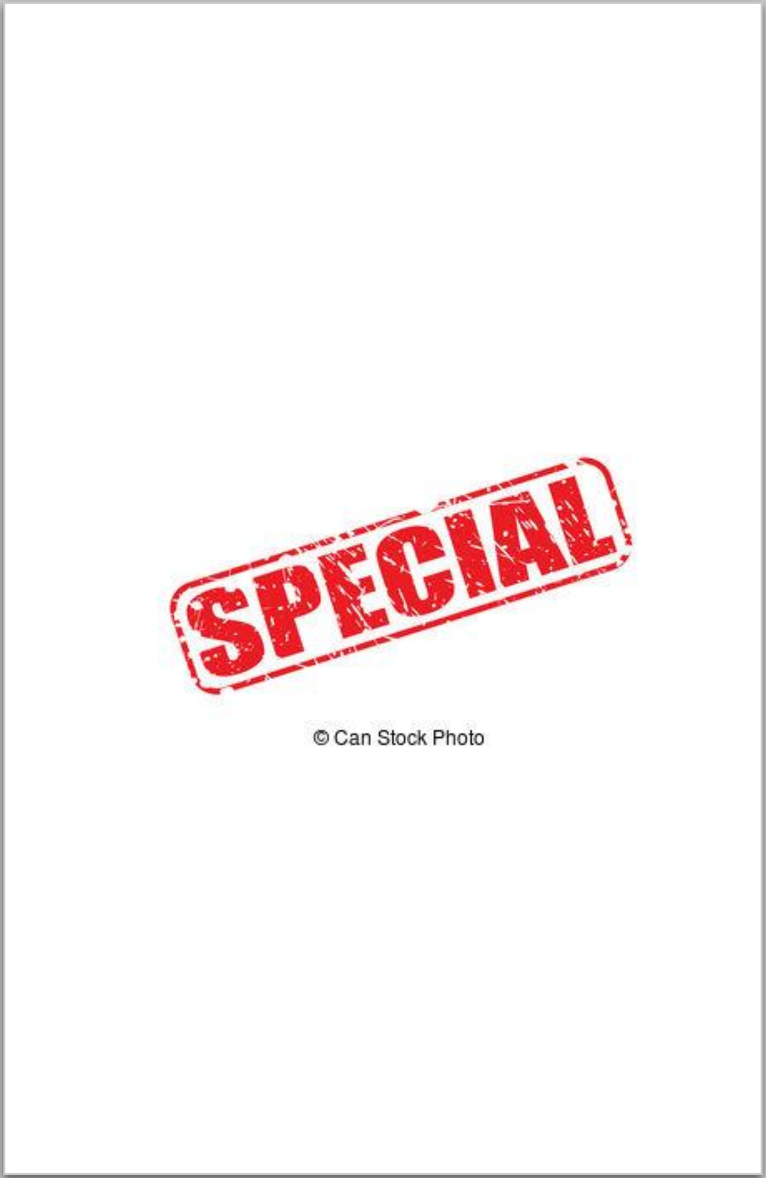
```
{var colour = red;
  visualize (text);
}
```

Pros and cons of dynamic scoping

- Through the redefinition of non-local variables it is possible to change the behaviour of the program
- However, it often makes programs more difficult to read
- It is less efficient

Static versus dynamic scoping

- Difference between static and dynamic scope only for not local and not global
- Static scoping (also lexical scoping)
  - All information is included in the program text
  - Associations can be derived at compile-time
  - Principle of independence
  - Easier to implement and more efficient
  - Used in Algol, Pascal, C, Java
- Dynamic scoping
  - Information derived during execution
  - It allows for changing the behaviour of the program
  - Often results in programs hard to read
  - Harder to implement and less efficient
  - Some versions of Lisp, Perl



Special cases

Special case: C

- `Algol`, Pascal, Ada, and Java allow nesting blocks of subprograms
- This is not possible in C
  - Functions can only be defined in the most external block
  - Therefore, the environment of a function is divided into a local and a global part and we do not have the problem of non-local definitions within functions

Determining the environment

- The environment is determined by
  - Scoping rules (static or dynamic)
  - Specific rules
    - For example, rules that clarify when a declaration is visible
- We will discuss later
  - Rules for parameter passing
  - Rules for binding (shallow or deep)
    - These apply when a procedure P is passed as a parameter to another procedure

The Pascal case

- Static scoping + 3 addition specific rules:
  1. Declarations can appear only at the start of a block.
  2. The scope of a name extends from the start to the end of the block in which the declaration of the name itself appears (excluding possible holes in scope) independently of the position of that declaration.
  3. All names not predefined by the language must be declared before use.

```
begin
    const fie = value;
    const value = 0;
...
end
```

Wrong!
value is used before its definition
(rule 3)

The Pascal case

```
begin
    const value = 1;
    procedure foo;
        const fie = value;
        const value = 0;
        ...
        begin
        ...
        end
...
end
```

The compiler should raise a semantic error! The local declaration of `value` should be the right one – hence the variable is used before its definition (rule 3)

What about C and Ada?

- Where is a declaration visible in the block where it appears?
  - From the declaration until the end of the block

```
int value = 1;
int foo(){
 int fie = value;
 int value = 0;
 return 0;
}
int main(){
    foo();
    return 0;
}
```

The local declaration of value is only visible from here on

What about Java?

- Where is a declaration visible in the block where it appears?
  - From the declaration until the end of the block
    - Java: Declaration of a variable

```
{a=1;
 int a;
}
```

Illegal!

How to deal with recursive types and mutual recursion?

  - For methods. It is also visible before the declaration
    - Java: declaration of a method. Mutually recursive methods are allowed

```
{void f(){
     g();
 }
 void g() {
     f();
 }
}
```

Mutual recursion

- How can we do mutual recursion (functions, types) in languages where a name must be declared before use?
  1. Relax this rule for functions and/or types
     - Java via methods

```
{void f(){
    g();
}
{void g(){
    f();
 }
}
```

     - Pascal by pointer types

```
type list = ^elem;
type elem = record
     info: integer;
     next: list;
end
```

^T denotes the type of pointers to objects of type T

Mutual recursion

2. Incomplete definitions
   - Ada

```
type elem;
type list is access elem;
type elem is record
     info: integer;
     next: list;
end
```

   - C

```
struct elem;
struct elem {
     int info;
     elem *next }
end
```

   - Pascal

```
procedure fie(A:integer); forward;
procedure foo(B: integer);
     begin ... fie(3); ... end
procedure fie;
     begin ... foo(4); ... end
```

Exercise 2.1

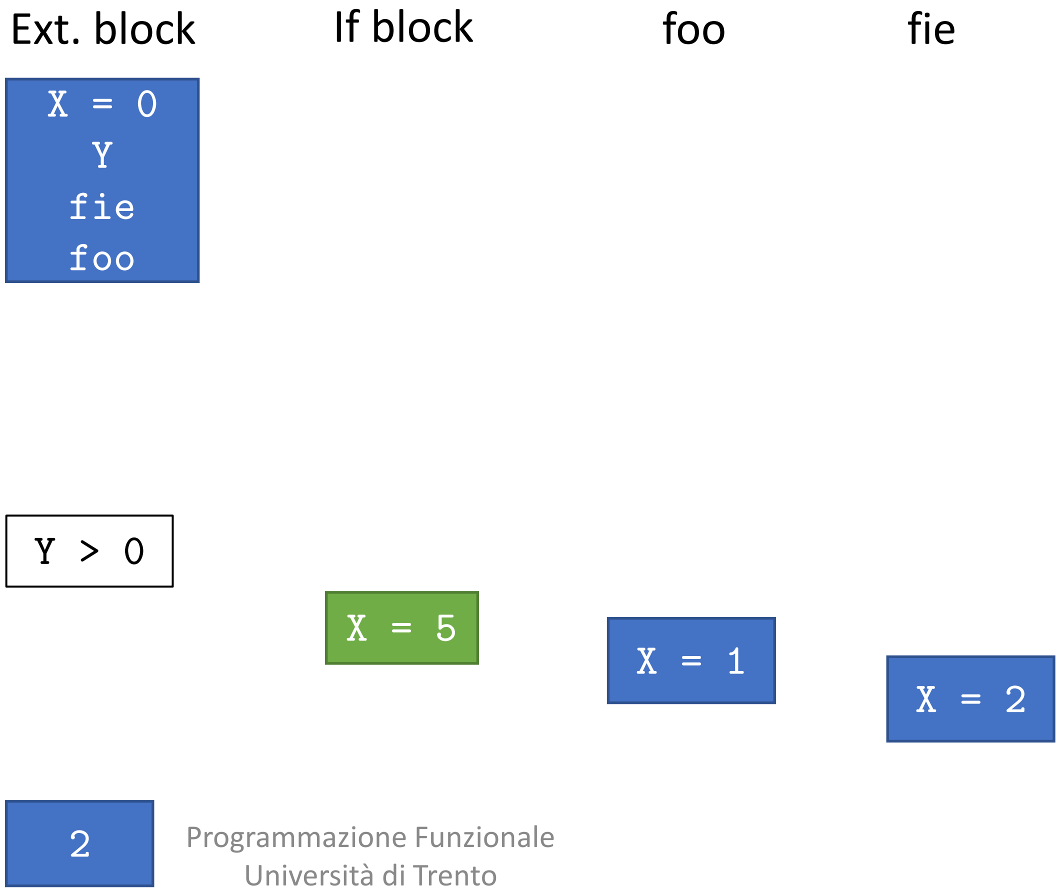
- Consider the following program fragment written in a pseudo-language which uses static scope and where the primitive `read(Y)` allows the reading of the variable `Y` from standard input. State what the printed value(s) is (are).

```
...
int X = 0;
int Y;
void fie(){
    X++;
}
void foo(){
    X++;
    fie();
}
read(Y);
if Y > 0{
    int X = 5;
    foo();}
else
    foo();
write(X);
```

Solution Exercise 2.1

- Consider the following program fragment written in a pseudo-language which uses static scope and where the primitive `read(Y)` allows the reading of the variable `Y` from standard input. State what the printed value(s) is (are).

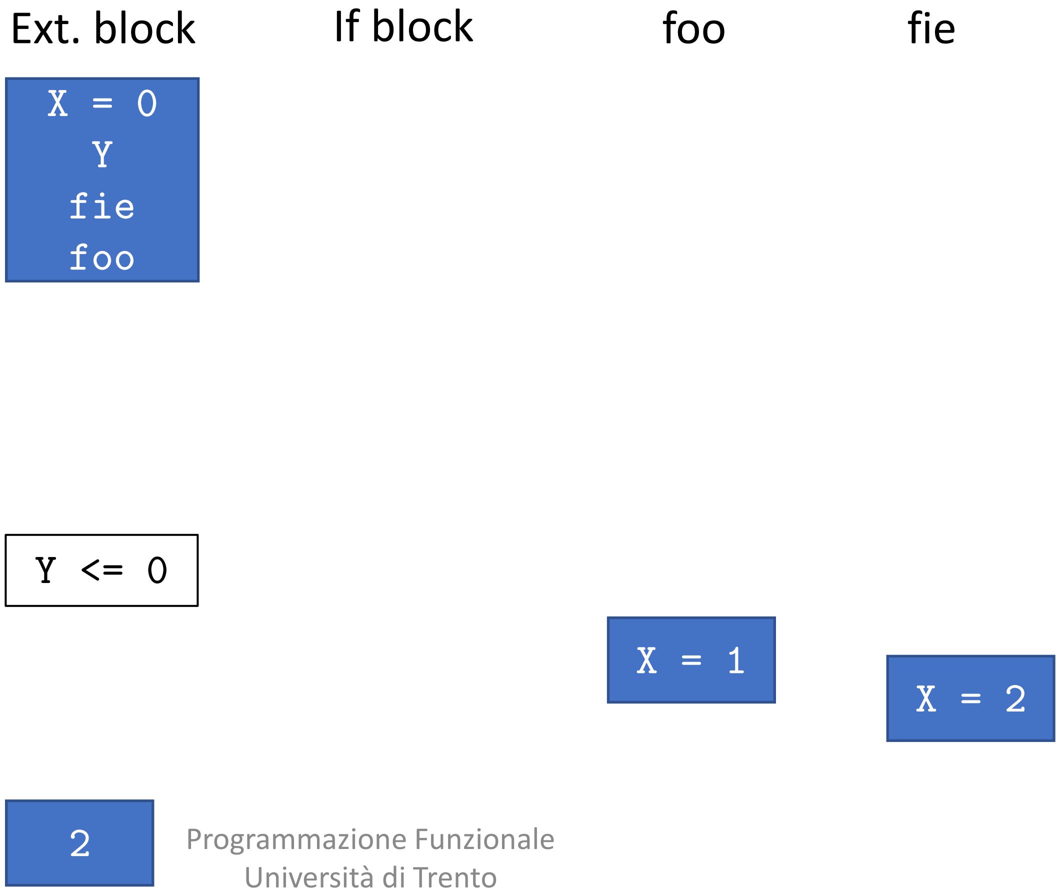
```
...
int X = 0;
int Y;
void fie(){
     X++;
}
void foo(){
     X++;
     fie();
}
read(Y);
if Y > 0{
     int X = 5;
     foo();}
else
     foo();
write(X);
```



Solution Exercise 2.1

- Consider the following program fragment written in a pseudo-language which uses static scope and where the primitive `read(Y)` allows the reading of the variable `Y` from standard input. State what the printed value(s) is (are).

```
...
int X = 0;
int Y;
void fie(){
     X++;
}
void foo(){
     X++;
     fie();
}
read(Y);
if Y > 0{
     int X = 5;
     foo();}
else
     foo();
write(X);
```



Solution Exercise 2.1

- Consider the following program fragment written in a pseudo-language which uses static scope and where the primitive `read(Y)` allows the reading of the variable `Y` from standard input. State what the printed values are.

```
...
int X = 0;
int Y;
void fie(){
    X++;
}
void foo(){
    X++;
    fie();
}
read(Y);
if Y > 0{
    int X = 5;
    foo();}
else
    foo();
write(X);
```

2

Exercise 2.2

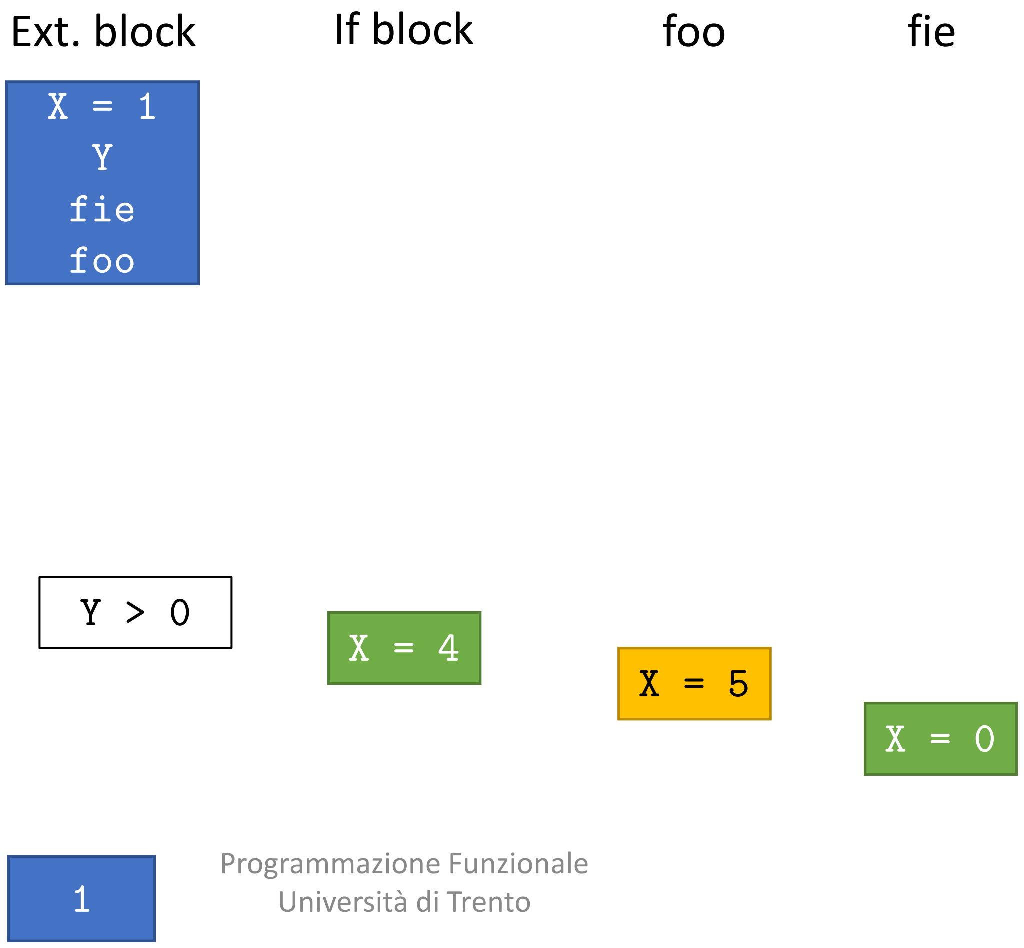
- Consider the following program fragment written in a pseudo-language that uses dynamic scoping. State what the printed value(s) is (are).

```
...
int X;
X = 1;
int Y;
void fie() {
    foo();
    X = 0; }
void foo(){
    int X;
    X = 5; }
read(Y);
if Y > 0 {
    int X;
    X = 4;
    fie();
}
else fie();
write(X);
```

Solution Exercise 2.2

- Consider the following program fragment written in a pseudo-language that uses dynamic scoping. State what the printed value(s) is (are).

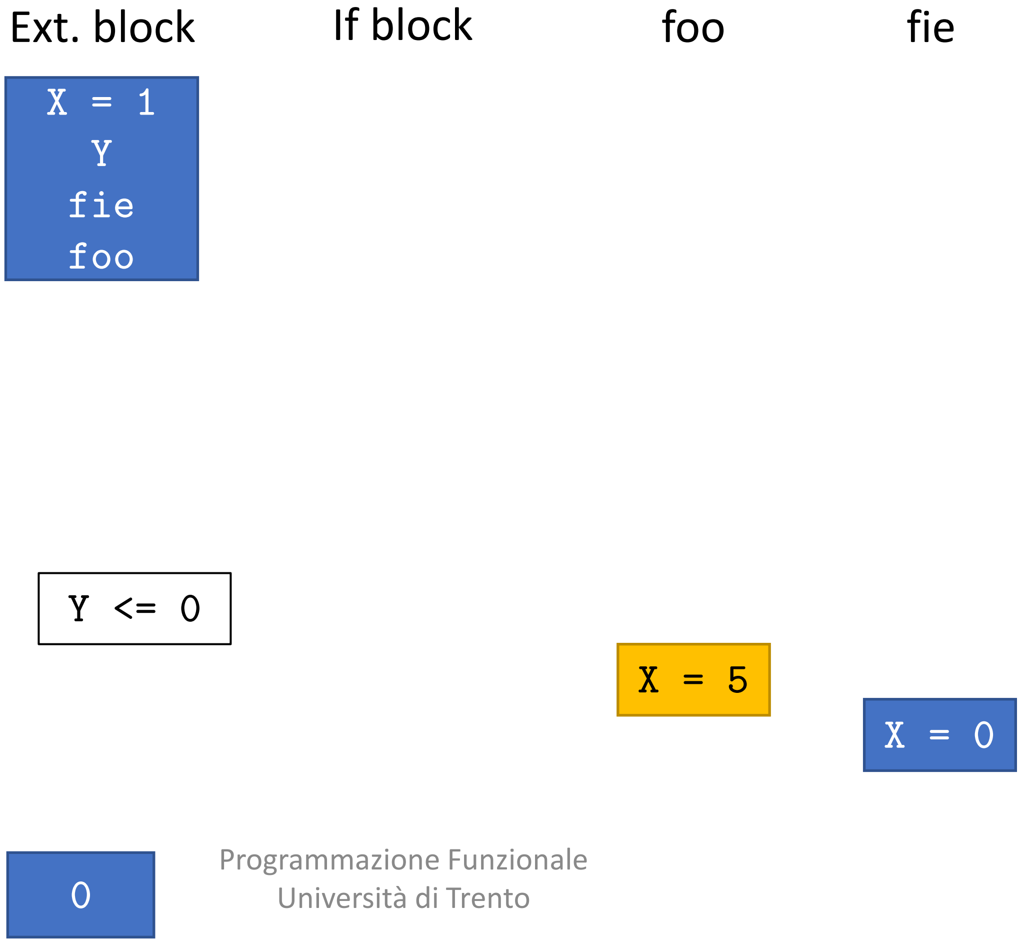
```
...
int X;
X = 1;
int Y;
void fie() {
     foo();
     X = 0; }
void foo(){
     int X;
     X = 5; }
read(Y);
if Y > 0 {
     int X;
     X = 4;
     fie();
}
else fie();
write(X);
```



Solution Exercise 2.2

- Consider the following program fragment written in a pseudo-language that uses dynamic scoping. State what the printed value(s) is (are).

```
...
int X;
X = 1;
int Y;
void fie() {
     foo();
     X = 0; }
void foo(){
     int X;
     X = 5; }
read(Y);
if Y > 0 {
     int X;
     X = 4;
     fie();
}
else fie();
write(X);
```



Solution Exercise 2.2

- Consider the following program fragment written in a pseudo-language that uses dynamic scoping. State what the printed value(s) is (are).

```
...
int X;
X = 1;
int Y;
void fie() {
     foo();
     X = 0; }
void foo(){
     int X;
     X = 5; }
read(Y);
if Y > 0 {
     int X;
     X = 4;
     fie();
}
else fie();
write(X);
```

Y > 0	Y <= 0
1	0

Exercise 2.3

- Consider the following code fragment in which there are gaps indicated by (*) and (**). Provide code to insert in these positions in such a way that:
  - If the language being used employs static scope, the two calls to the procedure `foo` assign the same value to `x`
  - If the language being used employs dynamic scope, the two calls to the procedure `foo` assign different values to `x`
- The function `foo` must be appropriately declared at (*). A loop is also a block.

```
{int i;
     (*)
     for (i=0; i<=1; i++){
          int x;
          (**)
          x = foo();
     }
}
```

Solution Exercise 2.3

- Consider the following code fragment in which there are gaps indicated by (*) and (**). Provide code to insert in these positions in such a way that:
  - If the language being used employs static scope, the two calls to the procedure `foo` assign the same value to `x`
  - If the language being used employs dynamic scope, the two calls to the procedure `foo` assign different values to `x`
- The function `foo` must be appropriately declared at (*). A loop is also a block.

```
{int i;
    (*)
    for (i=0; i<=1; i++){
        int x;
        (**)
        x = foo();
    }
}
```

A possible solution:

```
(*)
int x = 3;
int foo(){
    return x;
}
(**) x = i;
```

Summary

- Names and Denotable Objects
- Environments and Blocks
- Scope Rules
  - Static scoping
  - Dynamic scoping
  - Special cases

Readings

- Chapter 4 of the reference book
  - Maurizio Gabbrielli and Simone Martini "Linguaggi di Programmazione - Principi e Paradigmi", McGraw-Hill

Next time

- Memory management