

Abstraction of control

Programmazione Funzionale

2024/2025

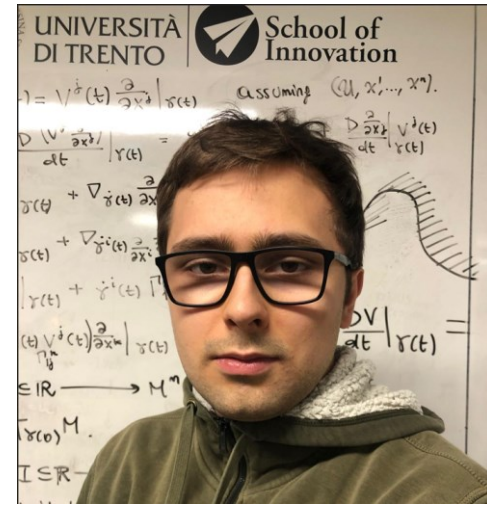
Università di Trento

Chiara Di Francescomarino

Tutors



Davide Dalla Betta



Mattia Maramotti

Tutoring slot

- Four options:
 - Monday early afternoon (14:30 - 15:30)
 - Monday late afternoon (16:30 – 17:30)
 - Wednesday afternoon (15:30 – 16:30) – this overlaps with English classes
 - Friday morning (11:30 – 12:30)

Tutoring slot

How to participate?



1

Go to wooclap.com

2

Enter the event code in the top banner

Event code

OKLORL

Agenda



1.

2.

3.

Today

- Type inference in ML
- Environment
- Scoping rules
- Abstraction of control and methods for parameter passing



Type inference in ML



Type inference in ML

- Types of operands and results of **arithmetic** expressions must agree, e.g., $(a+b)*2.0$
 - The right operand is a `real`
 - Therefore $a+b$ must be a `real`
 - Therefore, so are a and b
- In a **comparison** (e.g., $a \leq 10$), both arguments have the same type, so a is an integer
- In a **conditional**, the types of the `then`, the `else` and the expression itself must all be the same

Type inference in ML

- If an expression used as an **argument of a function** is of a known type, the parameter must be of that type
- If the expression defining the **result of a function** is of a known type, the function returns that type
- If there is no way to determine the types of the arguments of an **overloaded function** (such as +), the type is the default (usually integer)
- If there is no way to determine the types of the arguments and operators are not used (so we do not have any type constraint), we can use the **generic type 'a**
- If there is no way to determine the type of two arguments and there is no relation among them, we can use the **generic types 'a and 'b**

What can be inferred about the types of the arguments in the following function?

```
fun foo (a,b,c,d) =  
  if a=b then c+1 else  
    if a>b then c  
    else b+d;
```

- In the second line we have the expression `c+1`
- Since `1` is an integer, `c` must also be an integer
- In the third line, the expressions following the `then` and `else` must be of the same type
- Since one of these is `c`, so the type of both is integer
- So `b+d` is of integer type
- Therefore, `b` and `d` must also be integers
- Since `a` and `b` are compared on lines 2 and 3, they must be of the same type

Therefore, `a` is also an integer

The function type is hence `val foo = fn: int * int * int * int -> int`



What's the inferred type ... without trying it 😊

How to participate?



1

Go to wooclap.com

2

Enter the event code in the top banner

Event code

GCUXRE

 [Copy participation link](#)

Few more questions

How to participate?



 [Copy participation link](#)



1

Go to wooclap.com

2

Enter the event code in the top banner

Event code

RTJGCD



Environments

Environment

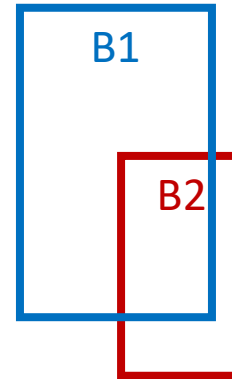
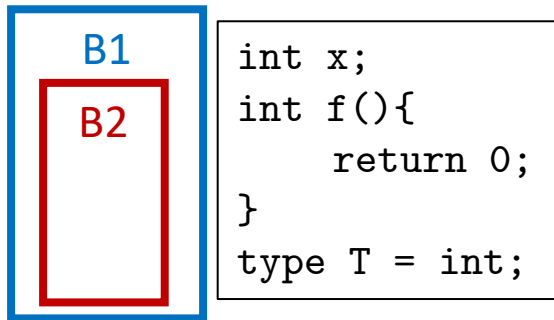
- **Environment:** the collection of associations between names and denotable objects that exist at runtime at a specific point in a program and at a specific moment in the execution
- **Declaration:** a mechanism (implicit or explicit) that creates an association in an environment.

Blocks

- In modern programming languages, 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
 - 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

Nesting

- Blocks can (for some languages) be nested



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



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

Environment

- 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

What defines the environment?

- **Visibility rules** (based on the block structure)
- Scope rules
- Rules for the parameter passing
- Binding policy

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

What defines the environment?

- Visibility rules (based on the block structure)
- Scope rules
- Rules for the parameter passing
- Binding policy



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

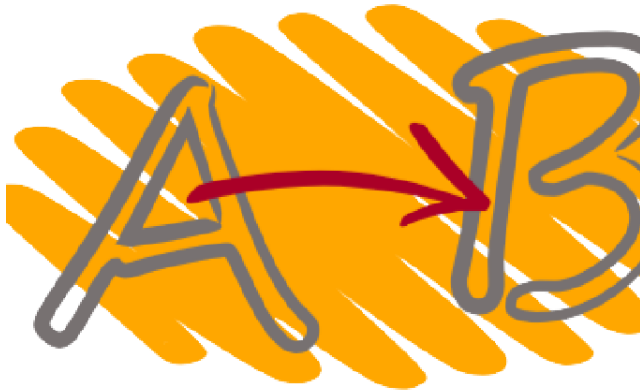


It depends on the scope rule used

Scope rule

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

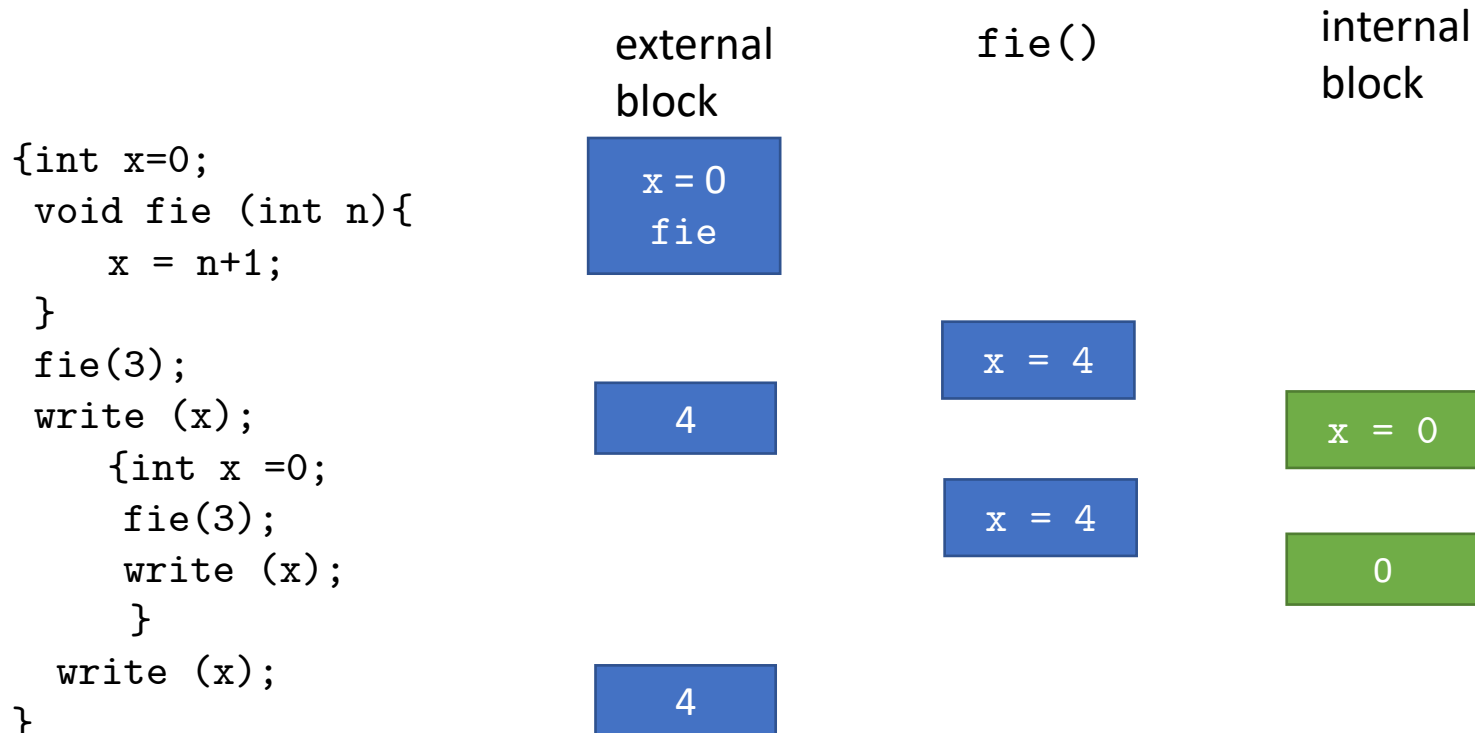
- **Static scoping**: in the block that syntactically includes the block
- **Dynamic scoping**: in the block that is executed immediately before the block



Static (lexical)
scoping

Static scoping

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



Static scoping: independent of local names

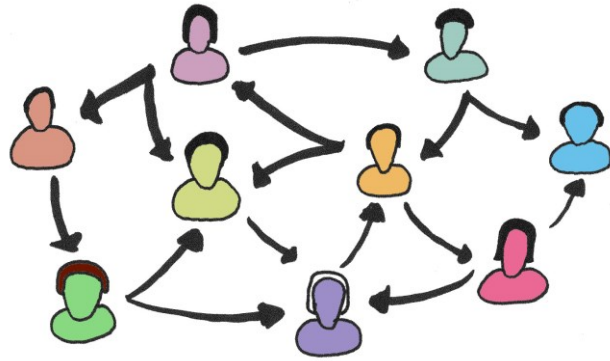
- Changing the local declaration from `x` to `y` in `fie`, has no effect with the static scoping (**independent of local names**).

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

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

Advantages of static scoping

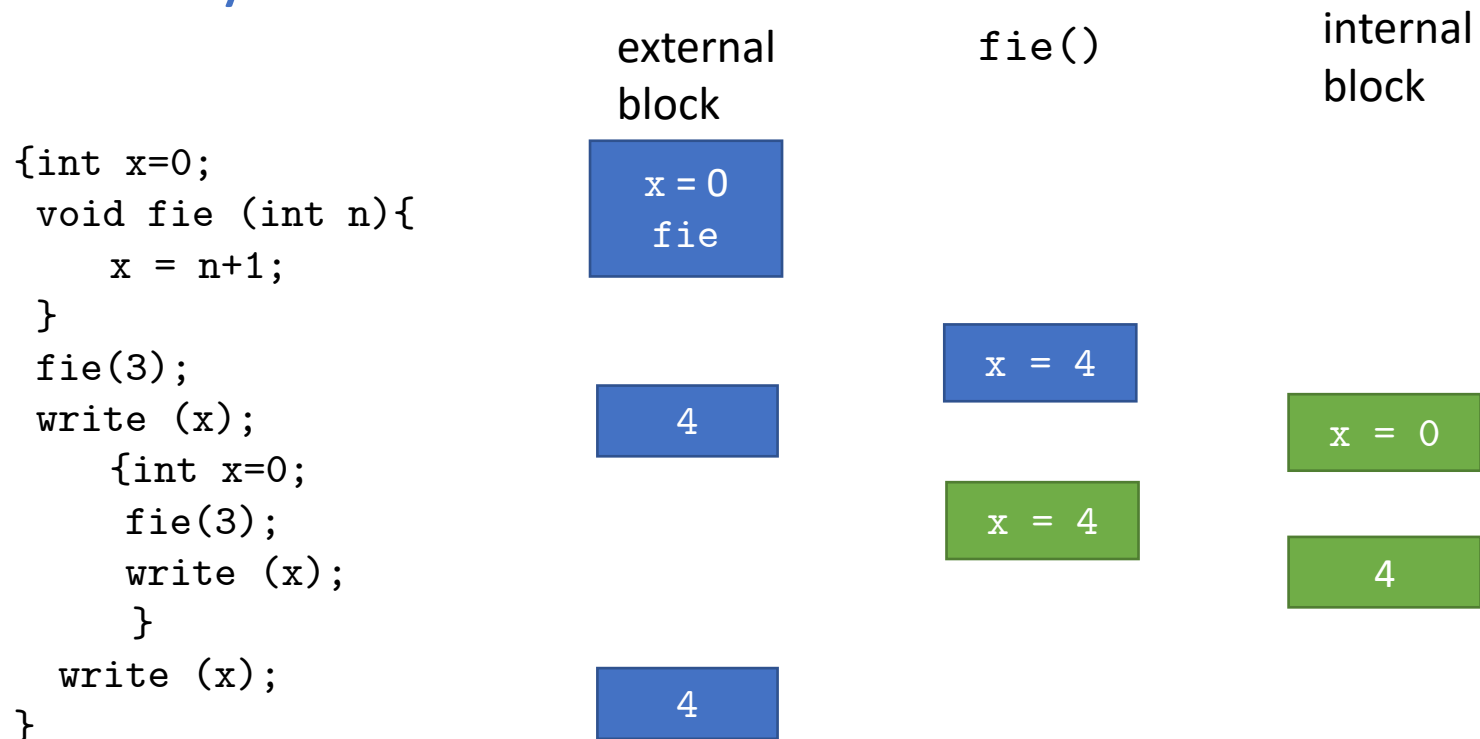
- The **programmer** has a **better understanding** of the program
- The **compiler** can connect the name occurrences with its correct declaration, thus enabling several **correctness tests and code optimisations**
- The compiler has some important information about the storage of the variables making more efficient the execution (w.r.t. the dynamic scoping)
- ALGOL, Pascal, C, C++, Ada, Scheme, ML and Java use some form of static scoping.



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**



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, **ML**
- **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



Environments in ML




Referencing external variables

- What is the result of this code?

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

x	10
addx	Definition of addx
x	3



previous
environment

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

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

What do the following pieces of code produce?

- After the following definitions

```
val a=2;
fun f(b)=a*b;
val b=3;
fun g(a) = a+b;
val a=4;
```
- What does the following produce?

```
> f(4);
val it = 8: int

> f(4)+b;
val it = 11: int

> f(4)+a;
val it = 12: int
```



What do the following pieces of code produce?

- After the following definitions

```
val a=2;  
fun f(b)=a*b;  
val b=3;  
fun g(a) = a+b;  
val a = 4;
```

- What do the following produce?

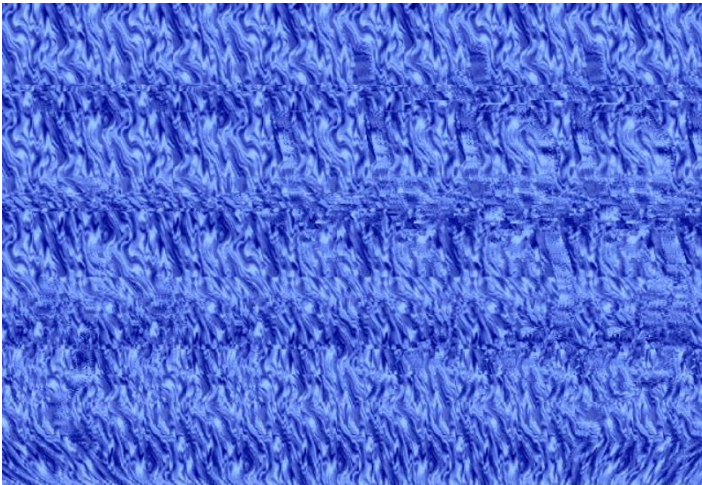
```
> g(5);  
val it = 8: int  
  
> g(5)+a;  
val it = 12: int  
  
> f(g(6));  
val it = 18: int  
  
> g(f(7));  
val it = 17: int
```



What defines the environment?

- Visibility rules (based on the block structure)
- Scope rules
- Rules for the parameter passing
- Binding policy

Abstraction of control



Abstraction of control

- Main mechanism: subprogram/procedure/function
- **Subprogram**: piece of code identified by its name, with a **local environment** and exchanging information with the rest of the code using **parameters**
- Two main constructs

definition

```
int foo (int n, int a) {  
    int tmp=a;  
    if (tmp==0) return n;  
    else return n+1;  
}
```

use

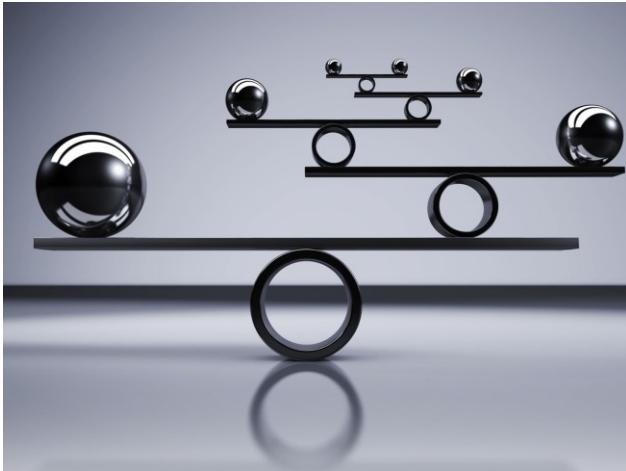
```
...  
int x;  
x = foo(3,0);  
x = foo(x+1,1);
```

Mechanisms for exchanging information with external code

- Parameters
- Return value
- Nonlocal environment

Methods for passing parameters

- Three classes of parameters:
 - Input parameters
 - Output parameters
 - Input/output parameters



Call by value

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!

An example

```
int y = 1;  
void foo (int x) {  
    x = x+1;  
}  
...  
y = 1;  
foo(y+1);
```

x assumes the initial value 2

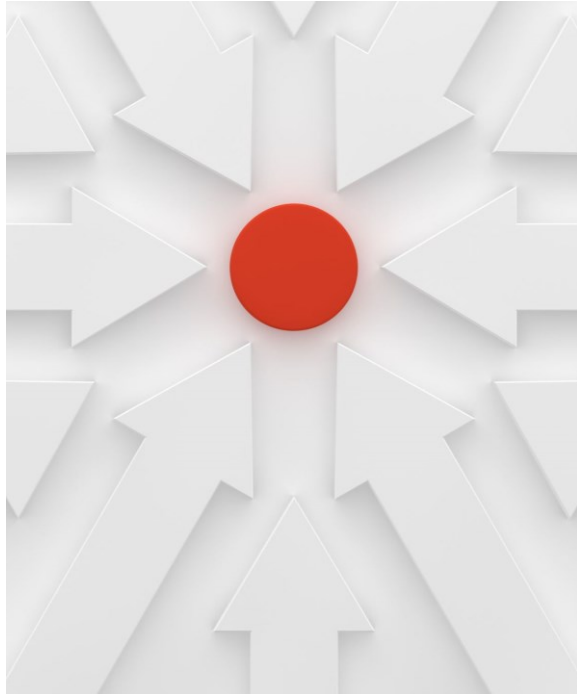
x is incremented to 3

x is destroyed

y+1 is evaluated, and its value assigned to x

y is still 1

- The formal parameter `x` is a local variable
- There is no link between `x` in the body of `foo` and `y` (`y` never changes its value)
- On exit from `foo`, `x` is destroyed
- It is not possible to transmit data from `foo` via the parameter

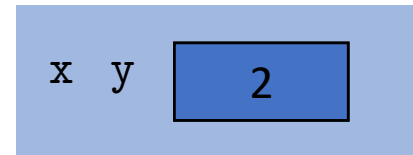


Call by
reference

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

An example



```
int y = 1;
void foo (reference int x) {
    x = x+1;
}
...
y = 1;
foo(y);
```

x is another name for y

x is incremented to 2

x and its link with y are destroyed

a reference is passed

y is 2

- A reference (address, pointer) is passed
- x is an alias of y
- The actual value is an l-value
- On exit from `foo`, the link between x and the address of y is destroyed
- Transmission: Two-way between `foo` and the caller

Call by value vs call by reference

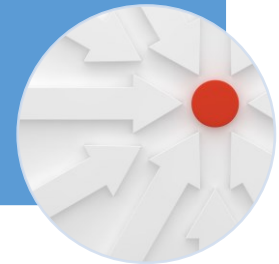
- Simple semantics
- Call could be expensive due to copy operations
- Need for other mechanisms to communicate with the called procedure

Call by
value



- Complicated semantics: aliasing
- Call is efficient
- Reference to formal parameter slightly more expensive

Call by
reference



3.14159265358979323846264338327950288419716939937510582097494459230781640628620898628034825
3421170679821480865132823069479384460995082231725394081284811174502841027019385211055984
46229489549303819644288109756659334461284756482337867831652712019091456485669234603486105453
266482133936072602491412737245870066063155881748815209209282925409171536437892590360011330
53054802466521384146951941511609433052703057599519530218811738193261179310511854807446237
9962749567351885752724891227938183011949129833673362440656643086021394946395247371907021798
6094370277053921717629314875238467481846766940513200056812714526356082778577137257789609173
63717872146844907223583301465498637105079227968925892354201995611212902198090403441815981
46297147713096031407211349999983727804895105973173281698631898050449453469830302642523
082533448850326531188171010003137838752886587533208381426171776691473035825339428755468
7311595626388263787583751657781657780521122806813001927876611185992154201983809525720
10654658632718659361533819277665230018503501859699977362256841389124971772634791315155
7485724245450895950825351108172785588055002915463746493937453300009927701671139009848
82401285031603597076801047101814242953961989467676374484452537977472684710404753464620804
68642590634444311977028891510475916205866240309381501901112533824300355876420474964732
6391419927260426992279678235478163609934172164121024588319030286182974555706749838505494588
58926995909272107975093029552116334443720275980736480965991198818347977535663680742654
257786251818417574672890777738803081478001614554919217214772350144419735685481613
61157352552133475741849468438533239073914333454776241686258983569485562099219222184272550
2542568876717904946016534668048982723271780083793830279977668145410095388378636905068006
4225125051173928489608412948862849404141865285221086115387442786220391843450471207137
8696095636437191728746776465753862413898658326459561339078027590099465764078951269468398
3259857088262620524894077251947826842601476970926401523843745530506820349625245174939
8651431423809198552053722165615107858374105786958912975489530151753924681362869538
6894277415599185592545853998431048972526808458827364469594865383673222626099124608051243
38439045124413654978278079775891435997701296160844184486355844063534202722582848864815
8460285001684273845225746378989259138254995401672828546698116334886230574864880355
93634568174324112515076089419451096596080252288701088314585913686722874884080105103308617
9286892087476091782493835020971480967593261365547818931297482168229999447276080485756401
4270477551523708414815217825438445489847825056413081813474852395231142761021538985
36231442952484937187110145765403590279934037420071057853806218638744780678489883321445713
86875194350643021845319048810053706148604919278701187939952061120574237544464374512371
81021789839101591561946741689123022409071864923196787823002514655202516038913301
4209376213785595663893787053390697920734672218295659966150142150306803844774549022605414
6659252014974428507325156600213243409519071048633173498514539057962685610050810665879699
8163574736384052571459102700704145119771206280439039709546771577004203789936072305587631
7635942473712514772053268174521412586732157919841484820454702695345069722091756711672
2910981690915280173506712748583222871835209353657212083791513698820914442100675103346711
03141267111369908658163983150197016515116851714370576183515565088490998989823873455283316
155078791803589326185499632152930895706420467259079154814165489594613718027081984399
9244889575128289059232326097299712084435732654893823911932597463673058360414481388303203
8249037589524374417029132765618093773444030707469211201913020330380197621101100449293215160
842444896376698386288477317355268211449578672824344189303984624243470772268780287
31891544110146823252716201052652271118603966557309254711057853763468820653109895629186

Call by constant

Call by constant/read -only

- Read-only parameter method: ➡
- 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)
- In Java: `final`

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

- In C/C++: `const`



Call by result

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 ➡

An example

```
void foo (result int x) {  
    x = 8;  
}  
...  
y = 1;  
foo(y);
```

x is a local variable

x is 8

the value of x is assigned to the current
l-value of y

x is destroyed

y is 8

- Dual of the call by value
- No link between `x` and `y` in the body of `foo`
- When `foo` ends, the value of `x` is assigned to the location obtained with the l-value of `y`
- It is important when the l-value of `y` is determined (when the function is called or when it terminates)



Call by value
result

Call by value-result

- Bidirectional communication using the formal parameter as a local variable \Leftrightarrow
- The actual parameter must be 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

An example

```
void foo (value-result int x) {  
    x = x+1;  
}  
...  
y = 8;  
foo(y);
```

x is a local variable

the value of y assigned to x

x = 9

the value of x assigned to y

x is destroyed

y is 9

- No link between x and y in the body of foo
- When foo ends, the value of x is assigned to the location obtained with the l-value of y



Call by name

Call by name

- Aim: give a precise semantics to parameter passing
- Copy-rule mechanism of the actual parameter to the formal one
- A call to P is the same as executing the body of P **after substituting the actual parameters for the formal one**
- “Macro expansion”, implemented in a semantically correct way: every time the formal parameter appears we re-evaluate the actual one
- Input and output parameters ⇔
- Appears to be simple but ... it is not that simple: it has to deal with variables with the same name
- No longer used by any imperative language

An example

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

- Blindly applying the copy rule would lead us to a result of $x+x+1=5$
 - Incorrect result as it would depend on the name of the local variable
 - With a body `{int z = 2; return z + y;}` the result would have been $z+x+1=3$
- When the body contains the same name of the actual parameter, we say that it is **captured by the local declaration**
 - In order to avoid substitutions in which the actual parameter is captured by the local declaration, we impose that **the formal parameter** – even after the substitution – **is evaluated in the environment of the caller and not of the callee**
 - Substitute the actual parameter together with its evaluation environment – fixed at the time of the call

Actual parameter evaluation

```
int y;  
void fie (int x){  
    int y;  
    x = x + 1; y = 0;  
}  
...  
y = 1;  
fie(y);
```

x is y (external)

x is 2

y (local) is 0

y is 2

- A pair $\langle \text{exp}, \text{env} \rangle$ is passed (**closure**), where
 - exp is the actual parameter, not evaluated
 - env is the evaluation environment
- Every time the formula is used, exp is evaluated in env

Call by name vs call by value-result

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

x is 1, y is A[1]

call- by value-result

x is 2

y is 1

i is 2, A[1] is 1

```
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]);
```

x is i, y is A[i]

call- by name

x is 2

y is 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 4.1

- 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);
}
```



Exercise 4.2

- 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);
}
```




Exercise 4.3

- 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);
```

Summary

- Type inference in ML
- Environments
- Scoping mechanisms
- Methods for parameter passing

SUMMARY



Readings

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



Next time

A yellow sticky note with a grey tab at the top left, featuring the text "Next Time" in a blue, hand-drawn font.

Next
Time

- Recursion in ML