

Abstraction of control and recursion

Programmazione Funzionale

2024/2025

Università di Trento

Chiara Di Francescomarino

Tutoring

- Tomorrow 15:30 – 16:30 in pc A201
- For those of you attending the English course who also wants to attend the tutoring hour, please fill in the form in Moodle or drop me an email every time you need it (the timeslot reserved would be Friday 11:30 – 12:30)

LET'S RECAP...

Recap

What defines the environment?

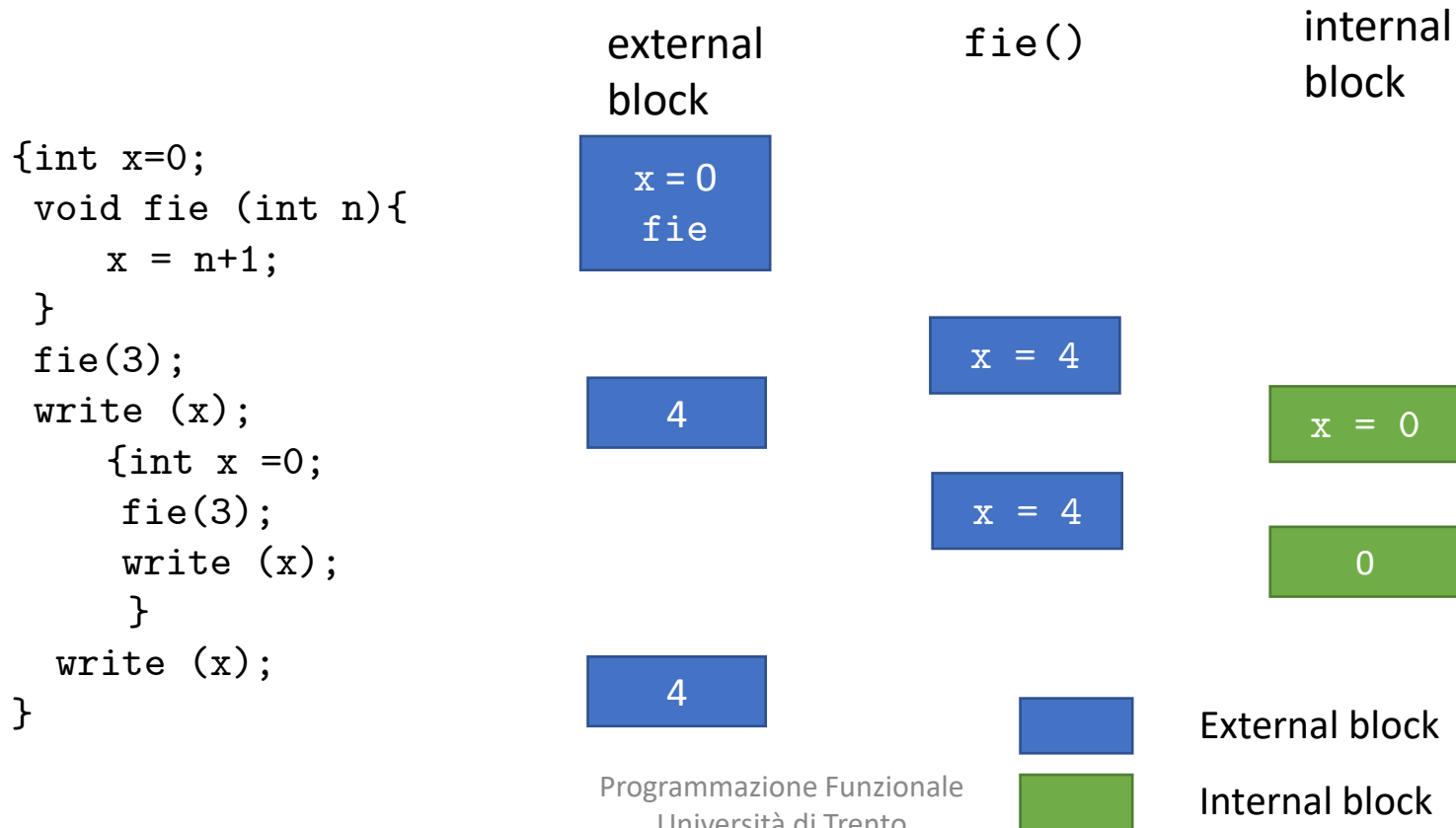
- Visibility rules (based on the block structure)
- Scope rules
- Rules for the parameter passing
- Binding policy

Static and dynamic scoping

- **Rules of visibility.** 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
- **Static scoping:** a block A is nested in block B if block B textually includes block A
- **Dynamic scoping:** a block A is nested in block B if block B has been most recently activated and has not yet been deactivated

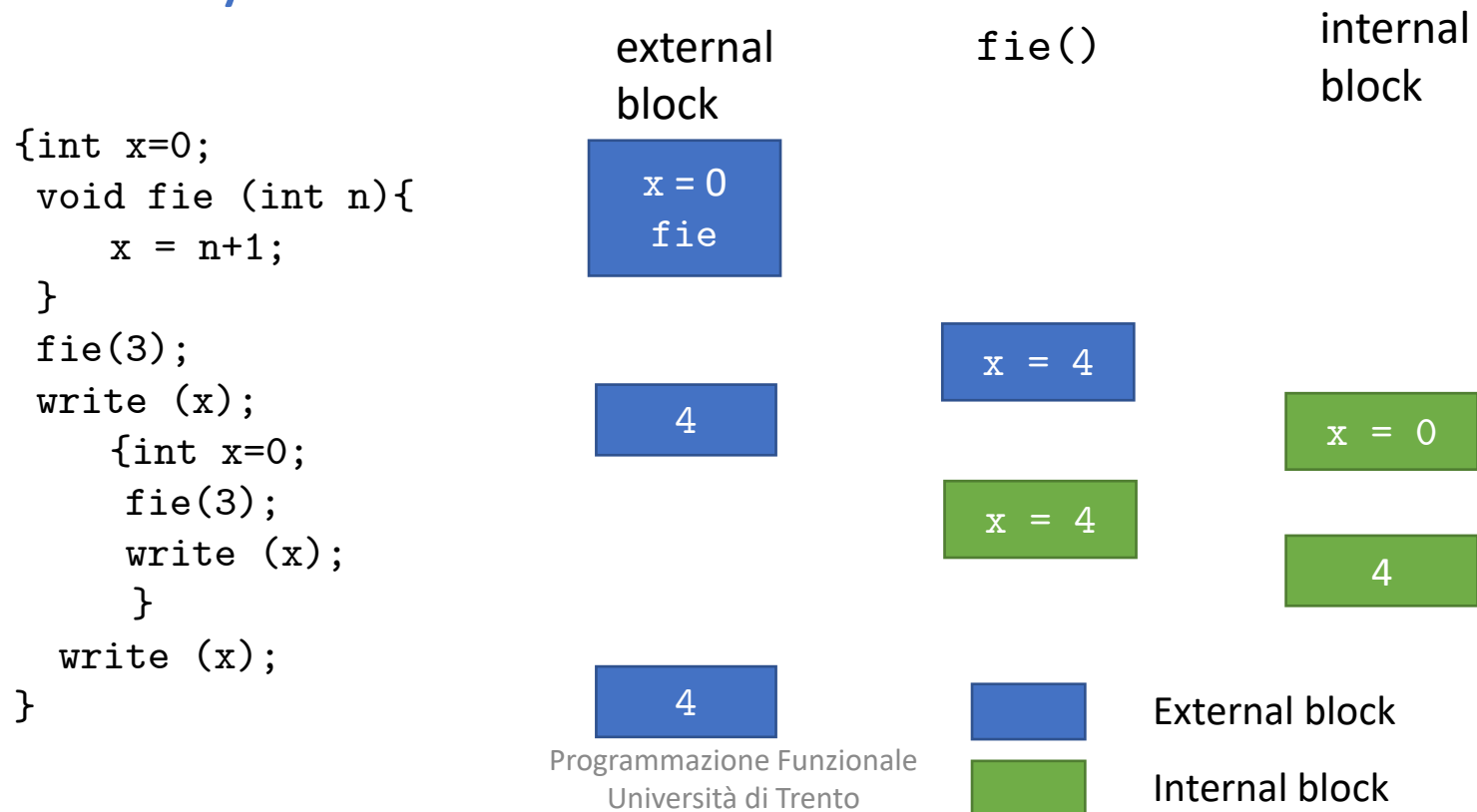
Static scoping

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



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**





Exercise 4.1

- 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);
```




Exercise 4.2

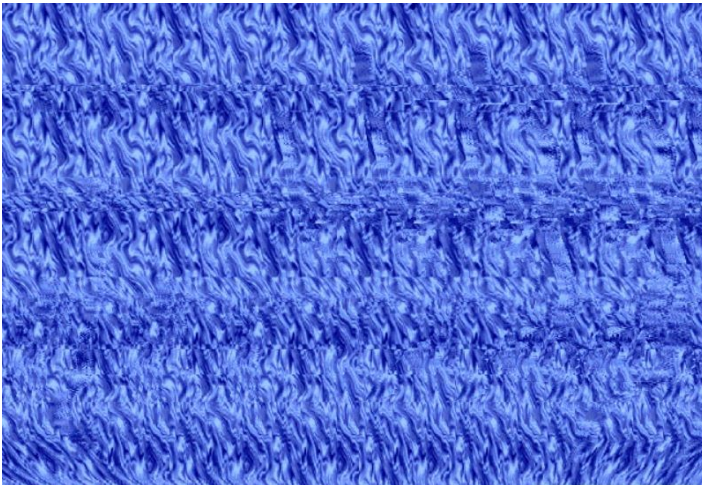
- Consider the following program fragment written in a pseudo-language that uses dynamic scoping. State what the printed value(s) is (are).

```
...  
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);
```

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

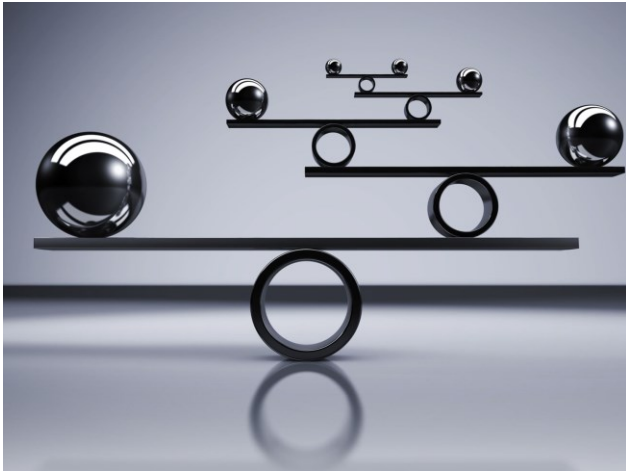
```
...  
int x;  
x = foo(3,0);  
x = foo(x+1,1);
```

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

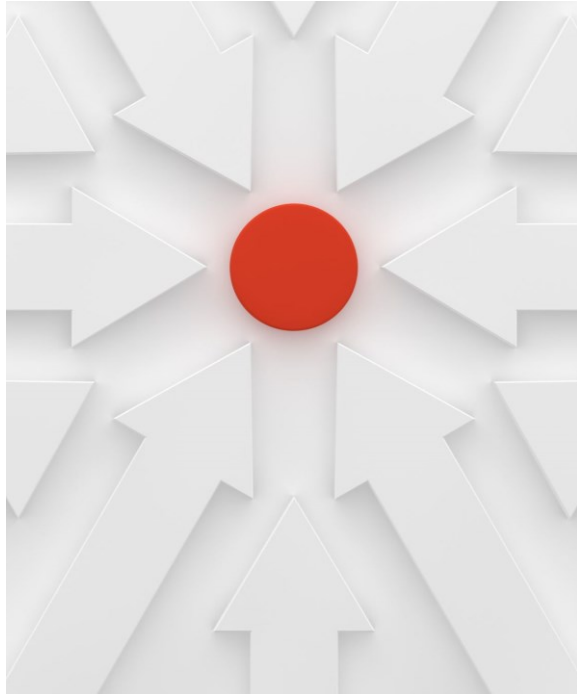
x is incremented to 3

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

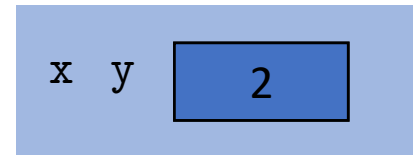


Call by
reference

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

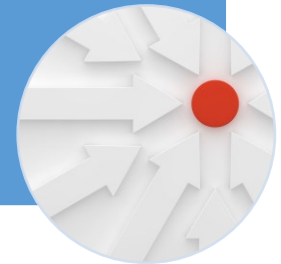
- Simple semantics
- Call could be expensive due to copy operations
- Need for other mechanisms to communicate with the called procedure

Call by
value



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

Call by
reference



3.14159265358979323846264338327950288419716939937510582097494459230781640628620898628034825
3421170679821480865132823069479384460995082231725394081284811174502841027018385211055894
46229489549303819644288109756659334461284756482337867831652712019091456485669234603486105453
2664821339360726024914127372458700660631558817488152092092828925409171536437892590360011330
53054802466521384146951941511609433052703057599519530218811738193261179310511854807446237
9962749567351885752724891227938183011949129833673362440656643086021394946395247371907021798
6094370277053921717629314875238467481846766940513200058127145263560827785771342757789609173
63717872146844901224583430146549863710507922796892589235420199561121290219809403441815981
46297147713096031407211349999983727870489510597317328169863189805044945346983032642523
08253344685032653118817101000313783875288587533208381426171776691473035825339428755468
731159562638826378758375165778165778052112280681300192787611185992154201983809525720
1065465863271865936153381927766523001850350185969897736256841389124971774534791315155
748572424545089595082535110817278558890500029154637464939374813300009927701671139009848
82401285931603597076801047101814242953961989467676374404452537977472684710404753464620804
68642590634444311977028891510475916205866124050938150190111253382430035587640247496732
6391419927260426992279678235478163609934172164121024588319030286182974555706749838505494588
5892699590927210797509302955211633444372027598073648096599119881834797535663680742654
257786251818417574672890777738803081478001614554919217214772350144419735685481613
6115735255213347574184946843853323907391433345476241686258983569485562099219222184272550
254258876717904946016534668048982723271780083793830279977668145410095388378636905068006
4225125051173928489608412948862849404141865285221086115387442786220391843450471207137
8696095636437191728746776465753862413898658326459561339078027590099465764078951269468398
32598708826262052489407725194782684260147697092640152384374553050682034962545174939
865143142880919855205372216515107858374105786958912975489530151753928481362869538
68942774155991855825458539984310489725268084588273644695948653836736222626099124608051243
384390451244136549782780797758914359977012961606044184486355844063534202722582848864815
84560285001684273845227463788952519822549954015728285466991163348862305774864880355
93634568174324112515076089479451096596090252288701088314585913686722874884080105103308617
9286892087476091782493835020971480967593261365547818931297482168229999447276080485756401
427047756515237084148115017825438445489847892658395231147785025395231147785025395895
36231442952484937187110145765403590279934037420071057853806218638744780678489883321445713
86875194350643021845319048810053706148604919278701187939952061120574237544464374512371
8102178983910159156194671489123022409071864923196787823002514655202516038913301
4209376213785595663893787053390697920734672218295659966150142150306803844774549022605414
66592520149744285073251566002132434009190710486331734596514539057962685610050810665879699
816357473638405251459102700704145119712062804390397095460771577004203789936072305587631
763594247312514772053268174541412586732157919841484820454702695345069722091756711672
29109816909152801735067127485832228718352093536675210835791513698820914442100675103346711
03141267111369908658163983150197016515116851714376576183515565088490998989823873455283316
155078781803883261854996321529308957064204672590791548141456489946137180270881984399
9244889575128289059232326097299712084435732654893823911932597463673058360414481388303203
824903758952437441702913276561809373444030707469211201913020330380197621101100449293215160
84244489637669838628847331355268211449578672824344189303984624243470772268780287
31891544110146823252716201052652271118603966557309254711057853763468820653109896529186

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 \Leftrightarrow
- 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

Actual parameter evaluation

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

x is z

x is 2

y is 0

z is 2

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

- 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

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

$\langle x \text{ is } y \text{ (external), } \{y/1\} \rangle$

$x \text{ is } 2 \text{ and } \{y/2\}$

$y \text{ (local) is } 0$

$y \text{ is } 2$

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 4.3

- 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 4.4

- 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);
}
```

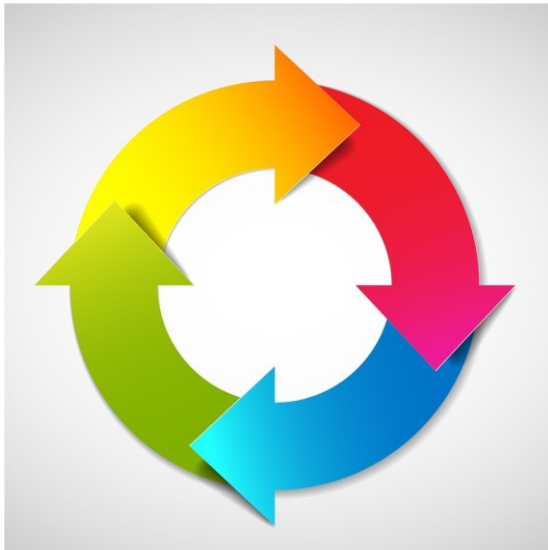
Remember that `x++` means that `x` is increased by 1 but the value of the expression is returned before the increment



Exercise 4.5

- 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);
```



Recursion in ML

Few suggestions for thinking in a recursive way

1. Carefully read the problem to solve
2. Think about the most trivial case for our function and for which we have an immediate result (e.g., the smallest set, 0, the empty list, the empty string) – without caring about recursive call
3. Think about the recursive case:
 1. Think you are able to solve the problem in the case immediately easier
 2. Build the more complex case

Reversing a list

- Example: `reverse([1,2,3])` is `[3,2,1]`
 - Base case: empty list to empty list
 - Induction: reverse the tail of the list (recursively) and then append the head

```
> fun reverse L =  
    if L = nil then nil  
    else reverse (tl L) @ [hd L];  
val reverse = fn: ''a list -> ''a list  
  
> reverse [1,2,3,4];  
val it = [4, 3, 2, 1]: int list  
> reverse["ab","bc","cd"];  
val it = ["cd", "bc", "ab"]: string list
```

How does the function execution works?

- The arguments are evaluated
- An addition to the environment: **call-by-value**

```
> fun reverse L =  
  if L = nil then nil  
  else reverse (tl L) @ [hd L];  
val reverse = fn: ''a list -> ''a list
```

reverse	Definition of reverse

Environment
Before the call

How does the function execution works?

- The arguments are evaluated
- An addition to the environment: **call-by-value**

```
> fun reverse L =
  if L = nil then nil
  else reverse (tl L) @ [hd L];
val reverse = fn: 'a list -> 'a list

> reverse [1,2,3];
```

L	[1,2,3]
reverse	Definition of reverse

Added in call to
reverse ([1,2,3])

Environment
Before the call

How does the function execution works?

- The arguments are evaluated
- An addition to the environment: **call-by-value**

```
> fun reverse L =
  if L = nil then nil
  else reverse (tl L) @ [hd L];
val reverse = fn: 'a list -> 'a list

> reverse [1,2,3];
```

L	[2,3]
L	[1,2,3]
reverse	Definition of reverse

Added in call to
reverse ([2,3])

Added in call to
reverse ([1,2,3])

Environment
Before the call

How does the function execution works?

- The arguments are evaluated
- An addition to the environment: **call-by-value**

```
> fun reverse L =
  if L = nil then nil
  else reverse (tl L) @ [hd L];
val reverse = fn: ''a list -> ''a list

> reverse [1,2,3];
```

L	[3]
L	[2,3]
L	[1,2,3]
reverse	Definition of reverse

Added in call to
reverse ([3])
Added in call to
reverse ([2,3])
Added in call to
reverse ([1,2,3])

Environment
Before the call

How does the function execution works?

- The arguments are evaluated
- An addition to the environment: **call-by-value**

```
> fun reverse L =
  if L = nil then nil
  else reverse (tl L) @ [hd L];
val reverse = fn: 'a list -> 'a list

> reverse [1,2,3];
```

L	nil
L	[3]
L	[2,3]
L	[1,2,3]
reverse	Definition of reverse

Added in call to
reverse (nil)
Added in call to
reverse ([3])
Added in call to
reverse ([2,3])
Added in call to
reverse ([1,2,3])

Environment
Before the call

How does the function execution works?

- The arguments are evaluated
- An addition to the environment: **call-by-value**

```
> fun reverse L =
  if L = nil then nil
  else reverse (tl L) @ [hd L];
val reverse = fn: 'a list -> 'a list
```

```
> reverse [1,2,3];
```

L	[3]
L	[2,3]
L	[1,2,3]
reverse	Definition of reverse

Added in call to
reverse ([3])
Added in call to
reverse ([2,3])
Added in call to
reverse ([1,2,3])

Environment
Before the call

`nil@[3] = [3]`

How does the function execution works?

- The arguments are evaluated
- An addition to the environment: **call-by-value**

```
> fun reverse L =
  if L = nil then nil
  else reverse (tl L) @ [hd L];
val reverse = fn: 'a list -> 'a list

> reverse [1,2,3];
```

L	[2,3]
L	[1,2,3]
reverse	Definition of reverse

Added in call to
reverse ([2,3])

Added in call to
reverse ([1,2,3])

Environment
Before the call

$[3] @ [2] = [3, 2]$

How does the function execution works?

- The arguments are evaluated
- An addition to the environment: **call-by-value**

```
> fun reverse L =
  if L = nil then nil
  else reverse (tl L) @ [hd L];
val reverse = fn: 'a list -> 'a list

> reverse [1,2,3];
val it = [3, 2, 1]: int list
```

L	[1,2,3]
reverse	Definition of reverse

Added in call to
reverse ([2,3])
Added in call to
reverse ([1,2,3])

Environment
Before the call

$[3,2]@[1] = [3,2,1]$

Different ways for writing functions

- Syntax **fn** (corresponds with λ in the λ -calculus, that we will see later)

```
fn <param> => <expression>;
```

- We can also directly apply the function to the parameter (**anonymous function**)

```
(fn n => n+1) 5;
```

- We can associate the functions to a name, just like values

```
> val increment = fn n => n+1;
```

```
val increment = fn: int -> int
```

- In case the function is recursive use **rec**

```
> val rec fact n = fn 0 => 1
```

```
  |n => n*fact(n-1);
```

- Syntactic sugar notation for functions with names (no need to specify **rec**)

```
> fun increment n = n+1;
```

```
val increment = fn: int -> int
```

Nonlinear recursion

- A function can call itself recursively multiple times
- Example: Number of combinations of k things out of n
 - Written $\binom{n}{k}$
 - Can be shown to be equal to $\frac{n!}{(n-k)!k!}$
 - We can also use the following recursive definition

Combinations

- **Base case.** If $k = 0$ the number of ways to pick 0 out of n is 1. Similarly, if $k = n$, there is exactly one way to pick n out of n .

$$\binom{n}{0} = \binom{n}{n} = 1$$

- **Induction.** If $0 < k < n$ to select k out of n we can
 - reject the first thing and select k out of the remaining $n - 1$
 - pick the first thing, and pick $k - 1$ out of the remaining $n - 1$
 - formally

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

- We assume that $0 \leq k \leq n$
- We use this to write a ML program

Combinations

```
> fun comb(n,k) = (* assumes 0 <= k <= n *)  
    if k=0 orelse k=n then 1  
    else comb(n-1,k) + comb(n-1,k-1);  
val comb = fn: int * int -> int
```

- Without using orelse:

```
> fun comb (n,k) =  
    if k=0 then 1  
    else  
        if k=n then 1 else comb(n-1,k)+comb(n-1,k-1);  
val comb = fn: int * int -> int
```

```
> comb (5,0);  
val it = 1: int  
> comb (5,5);  
val it = 1: int  
> comb (5,2);  
val it = 10: int
```

Why the type is `fn: int * int -> int`?

```
> fun comb(n,m) = (* assumes 0 <= m <= n *)  
    if m=0 orelse m=n then 1  
    else comb(n-1,m) + comb(n-1,m-1);
```

```
val comb = fn: int * int -> int
```

- Result of `if` clause is an integer
- Therefore `comb(n-1,m)+comb(n-1,m-1)` is an integer
- Therefore, so are `comb(n-1,m)` and `comb(n-1,m-1)`
- Therefore `comb` returns an integer
- The expression `n-1` and `m-1` must be integers (because of `-1`)
- `comb` maps pairs of integers to an integer

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
```

$\wedge T$ 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
```

Mutual recursion

- Two functions can call one another recursively
- Example. A function that takes a list L and produces a list with the first, third, fifth etc. elements of L
- Two functions:
 - `take(L)` takes the first element of L and then alternates
 - `skip(L)` skips the first element and then calls `take`

First attempt

```
> fun take(L) =  
  if L = nil then nil  
  else hd(L) :: skip(tl(L));  
> fun skip(L) =  
  if L = nil then nil  
  else take(tl(L));
```



```
> fun take(L) =  
  if L = nil then nil  
  else hd(L) :: skip(tl(L));
```

poly: : error: Value or constructor (skip) has not been declared

Found near if L = nil then nil else hd (L) :: skip (tl (L))

Solution: keyword `and`

```
fun
    <definition of first function>
and
    <definition of second function>
and ...
```

```
fun
take(L) =
    if L = nil then nil
    else hd(L) :: skip(tl(L))
and
skip(L) =
    if L = nil then nil
    else take(tl(L));
val skip = fn: 'a list -> 'a list
val take = fn: 'a list -> 'a list
```

```
> take ([1,2,3,4,5]);
val it = [1, 3, 5]: int list
```

Summary

- Scoping mechanisms
- Abstraction of control and methods for parameter passing
- Recursion in ML

SUMMARY



Readings

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

