Subprograms

Akim Demaille, Etienne Renault, Roland Levillain

April 6, 2020

Table of Contents

Subprograms

- At the origin, snippets copied and pasted from other sources
  - Impact on memory management;
  - Impact on separated compilation;
  - Modular programming: first level of interface/abstraction.
- First impact on Software Engineering: "top-down" conception, by refinements.
- Generalizations: modules and/or objects.

Table of Contents

Procedures vs. Functions

Procedure Subprograms with no return value.
Procedures have side effects

Function Subprograms that return something.
(Pure) Functions do not have side effects

Ada, Pascal, ... have two reserved keywords **procedure** and **function**
BUT function generally describe subprograms with return values,
while procedures do not return values

Distinction sometimes blurred by the language:
(e.g., using `void` ALGOL, C, Tiger...).

Procedures vs. Functions

```
Function Add(A, B : Integer) : Integer;
Begin
 Add := A + B;
End;
```

Functions in **Pascal**

```
Procedure finish(name: String);
Begin
  WriteLn('Goodbye␣', name);
End;
```

Procedures in **Pascal**

Vocabulary

Formal Argument Arguments of a subprogram declaration.

```
let function
    sum (x: int, y: int): int = x + y
```

Vocabulary

Formal Argument Arguments of a subprogram declaration.

```
   let function
       sum (x: int, y: int): int = x + y
```

Effective Argument Arguments of a call to a subprogram.

```
sum (40, 12)
```

Vocabulary

Formal Argument Arguments of a subprogram declaration.

```
let function
    sum (x: int, y: int): int = x + y
```

Effective Argument Arguments of a call to a subprogram.

```
sum (40, 12)
```

Parameter Please reserve it for templates.

Table of Contents

Functions: Side effects

Using functions with side effects is very dangerous. For instance:

```
foo = getc () + getc () * getc ();
```

is undefined ($\neq$ nondeterministic). *On purpose*!

Table of Contents

Default arguments

```
int sum(int a, int b = 21, int c = 42,
        int d = 42){
   return a + b + c + d;
}
```

Default Arguments in C++

- sum(1, 2, 3, 4) is fine
- sum(1, 2) is also fine
- But what if we want to call sum ($b = 1$, $a = 2$) with c's and d's default value?

Named Argument (Some sugar)

In Ada, named arguments and/or default values:

```
put (number   : in float;
     before   : in integer := 2;
     after    : in integer := 2;
     exponent : in integer := 2) ...
```

Some Ada function declaration

```
put (pi, 1, 2, 3);
put (pi, 1);
put (pi, 2, 2, 4);
put (pi, before => 2,
     after => 2, exponent => 4);
put (pi, exponent => 4);
```

Possible invocations

Named Arguments

Named parameters are availables in many languages: Perl, Python, C#, Fortran95, Go, Haskell, Lua, Ocaml, Lisp, Scala, Swift/ObjectiveC (**fixed order of named parameters**!), ...

- No need to remember the order of parameters
- No need to guess specific **default's** values
- More Flexible
- Clarity

Simulate Named Argument

Can we simulate **named arguments** in C++ or Java?

Yes : **Named parameter idiom** uses a proxy object for passing the parameters.

Named Parameter Idiom 1/2

```
class foo_param{
private:
  int a = 0, b = 0;
  foo_param() = default; // make it private
public:
  foo_param& with_a(int provided){
    a = provided; return *this;
  }
  foo_param& with_b(int provided){
    b = provided; return *this;
  }
  static foo_param create(){
    return foo_param();
  }
};
```

Named Parameter Idiom 2/2

```
void foo(foo_param& f)
{
  // ...
}

foo(foo_param::create().with_b(1)
                       .with_a(2));
```

Named Parameter Idiom 2/2

```
void foo(foo_param& f)
{
  // ...
}

foo(foo_param::create().with_b(1)
                       .with_a(2));
```

Works ... but require one specific class per function

For C++, Boost::Parameter library also offer a generic implementation

Table of Contents

Argument passing

From a naive point of view (and for **strict evaluation**), three possible modes: `in`, `out`, `in-out`. But there are different flavors.

	Val	ValConst	RefConst	Res	Ref	ValRes	Name
ALGOL 60	*						*
Fortran					?	?	
PL/1					?	?	
ALGOL 68		*			*		
Pascal	*				*		
C	*	?			?		
Modula 2	*				?		
Ada (simple types)		*		*		*	
Ada (others)		?	?	?	?	?	
Alphard		*	*		*		

Table of Contents

Call by Value – definition

Passing arguments to a function copies the actual value of an argument into the formal parameter of the function.

In this case, changes made to the parameter inside the function have no effect on the argument.

```python
def foo(val):
    val = 1

i = 12
print (i)
```

Call by value in Python – **output: 12**

Pros & Cons

- **Safer**: variables cannot be accidentally modified
- **Copy**: variables are copied into formal parameter *even for huge data*
- **Evaluation before call**: resolution of formal parameters must be done before a call
  - Left-to-right: Java, Common Lisp, Effeil, C#, Forth
  - Right-to-left: Caml, Pascal
  - Unspecified: C, C++, Delphi, , Ruby

Table of Contents

Call by Reference – definition

Passing arguments to a function copies the actual address of an argument into the formal parameter of the function.

In this case, changes made to the parameter inside the function will have effect on the argument.

```
void swap(int &x, int &y)
     { int aux = x; x = y;  y = aux;  }

int main() {
  int x = 2, y = 3;
  swap(a, b);
  printf("%d,␣%d\n", x, y);
}
```

Call by reference in C++ – **output: 3 2**

Pros & Cons

- **Faster** than call-by-value if data structure have a large size.
- **Readability & Undesirable behavior**: a special attention may be considered when doing operations on multiple references **since they can all refer to the same object**

```
void xor_swap(int &x, int &y) {
  x = x ^ y;
  y = x ^ y;
  x = x ^ y;
}
```

Call by reference in C++ may lead to undesirable behavior when x and y refers the same object (zeroing x and y)

Notes on call-by-reference

- *swap*(*foo*, *foo*) is forbidden in Pascal but what about *swap*(*foo*[*bar*], *foo*[*baz*]) ...

Table of Contents

Call by Value-Result – definition

Passing arguments to a function copies the argument into the formal parameter of the function.
The values are then copied back when exiting the function

In this case, changes made to the parameter inside the function will only reflect on the argument at the end of the function.

Call by Value-Result – Example

```
procedure Tryit is
   procedure swap (i1, i2: in out integer) is
      tmp: integer;
   begin
       tmp := i1;    i1   := i2;    i2   := tmp;
   end swap;

a : integer := 1;    b : integer := 2;
begin
  swap(a, b);
  Put_Line(Integer'Image (a) & "␣" &
           Integer'Image (b)) ;
end Tryit;
```

Call by Value-result in Ada – **output: 2 1**

Notes on call-by-value-result

Pros & Cons

- **Safety** other thread will only see consistent values since changes made will not show up until after the end of the function.
- **Local copies**: but they can be sometimes avoided by the compiler

Notes on call-by-value-result

Pros & Cons

- **Safety** other thread will only see consistent values since changes made will not show up until after the end of the function.
- **Local copies**: but they can be sometimes avoided by the compiler

Remarks:

Notes on call-by-value-result

Pros & Cons

- **Safety** other thread will only see consistent values since changes made will not show up until after the end of the function.
- **Local copies**: but they can be sometimes avoided by the compiler

Remarks:

- Also called: **Call by copy-restore**, **Call by copy-in copy-out**

Notes on call-by-value-result

Pros & Cons

- **Safety** other thread will only see consistent values since changes made will not show up until after the end of the function.
- **Local copies**: but they can be sometimes avoided by the compiler

Remarks:

- Also called: **Call by copy-restore**, **Call by copy-in copy-out**
- If the reference is passed to the callee uninitialized, this evaluation strategy is called **call by result**.

Notes on call-by-value-result

Pros & Cons

- **Safety** other thread will only see consistent values since changes made will not show up until after the end of the function.
- **Local copies**: but they can be sometimes avoided by the compiler

Remarks:

- Also called: **Call by copy-restore**, **Call by copy-in copy-out**
- If the reference is passed to the callee uninitialized, this evaluation strategy is called **call by result**.
- Used in multiprocessing contexts.

Notes on call-by-value-result

Pros & Cons

- **Safety** other thread will only see consistent values since changes made will not show up until after the end of the function.
- **Local copies**: but they can be sometimes avoided by the compiler

Remarks:

- Also called: **Call by copy-restore**, **Call by copy-in copy-out**
- If the reference is passed to the callee uninitialized, this evaluation strategy is called **call by result**.
- Used in multiprocessing contexts.
- Multiple interpretations:
  - Ada: Evaluates arguments once, during function call
  - AlgolW: Evaluates arguments during call **AND** when exiting the function

Table of Contents

An outsider: call by name

(In ALGOL 60) It behaves as a macro would, including with name captures: the argument is evaluated *at each use*.

- Try to write some code which results in a completely different result had SWAP been a function.

```
#define SWAP(Foo, Bar) \
  do {                 \
    int tmp_ = (Foo);  \
    (Foo) = (Bar);     \
    (Bar) = tmp_;      \
  } while (0)
```

- In ALGOL 60, a *compiled* language, "thunks" were introduced: snippets of code that return the l-value when evaluated.

An application of call by name: Jensen's Device

- General computation of a sum of a series $\sum_{k=l}^{u} a_k$:

```
real procedure Sum(k, l, u, ak)
      value l, u;
      integer k, l, u;
      real ak;
      comment 'k' and 'ak' are passed by name;
   begin
      real s;
      s := 0;
      for k := l step 1 until u do
         s := s + ak;
      Sum := s
   end;
```

- Computing the first 100 terms of a real array V[]:

```
Sum(i, 1, 100, V[i])
```

Table of Contents

Call by Need

Call by need is a memoized variant of call by name where, if the function argument is evaluated, that value is stored for subsequent uses.

The argument is then evaluated only once, during its first use.

What if $y = 0$ in the following code?

```
let function loop (z: int):int =
        if z > 0 then z else loop (z)
    function f    (x: int):int =
        if y > 8 then x else -y
in
  /* ≡ ‘if y > 8 then loop (y) else -y’ ? */
  f (loop (y))
end
```

Call by name

Don't pass the evaluation of the expression, but a "thunk" computing it:

```
    let var      a    := 5 + 7 in
          a + 10
    end
==> let function a () := 5 + 7 in
          a () + 10
    end
```

Call by need

The thunk is evaluated once and only once. Add a "memo" field.

Lazy evaluation 1

```
easydiff f x h = (f (x + h) - f (x)) / h

repeat f a = a : repeat f (f a)
halve x = x / 2
differentiate h0 f x = map (easydiff f x) (repeat halve

within eps (a : b : rest)
  | abs (b - a) <= eps = b
  | otherwise          = within eps (b : rest)

relative eps (a : b : rest)) =
  | abs (b - a) <= eps * abs b = b
  | otherwise                  = relative eps (b : rest)
within eps (differentiate h0 f x)
```

Slow convergence... Suppose the existence of an error term:

```
a (i)     = A + B * (2 ** n) * (h ** n)
a (i + 1) = A + B * (h ** n)
```

Table of Contents

Exhibit the differences (Explicit lyrics. . .)

```
var t   : integer
    foo: array [1..2] of integer;

procedure shoot_my (x : Mode integer);
begin
  foo[1] := 6;
  t      := 2;
  x      := x + 3;
end;

begin
  foo[1] := 1;
  foo[2] := 2;
  t      := 1;
  shoot_my (foo[t]);
end.
```

Exhibit the differences (Explicit lyrics. . .)

```
var t  : integer
    foo: array [1..2] of integer;

procedure shoot_my (x : Mode integer);
begin
  foo[1] := 6;
  t      := 2;
  x      := x + 3;
end;

begin
  foo[1] := 1;
  foo[2] := 2;
  t      := 1;
  shoot_my (foo[t]);
end.
```

Mode	foo[1]	foo[2]	t
Val			

Exhibit the differences (Explicit lyrics. . .)

```
var t  : integer
    foo: array [1..2] of integer;

procedure shoot_my (x : Mode integer);
begin
  foo[1]  := 6;
  t       := 2;
  x       := x + 3;
end;

begin
  foo[1]  := 1;
  foo[2]  := 2;
  t       := 1;
  shoot_my (foo[t]);
end.
```

Mode	foo[1]	foo[2]	t
Val	6	2	2
Val-Res (ALGOL W)			

Exhibit the differences (Explicit lyrics. . .)

```
var t  : integer
    foo: array [1..2] of integer;

procedure shoot_my (x : Mode integer);
begin
  foo[1] := 6;
  t      := 2;
  x      := x + 3;
end;

begin
  foo[1] := 1;
  foo[2] := 2;
  t      := 1;
  shoot_my (foo[t]);
end.
```

PARENTAL ADVISORY EXPLICIT LYRICS

Mode	foo[1]	foo[2]	t
Val	6	2	2
Val-Res (ALGOL W)	6	4	2
Val-Res (Ada)			

Exhibit the differences (Explicit lyrics. . .)

```
var t  : integer
    foo: array [1..2] of integer;

procedure shoot_my (x : Mode integer);
begin
  foo[1]  := 6;
  t       := 2;
  x       := x + 3;
end;

begin
  foo[1]  := 1;
  foo[2]  := 2;
  t       := 1;
  shoot_my (foo[t]);
end.
```

Mode	foo[1]	foo[2]	t
Val	6	2	2
Val-Res (ALGOL W)	6	4	2
Val-Res (Ada)	4	2	2
Ref			

Exhibit the differences (Explicit lyrics. . .)

```
var t  : integer
    foo: array [1..2] of integer;

procedure shoot_my (x : Mode integer);
begin
  foo[1]  := 6;
  t       := 2;
  x       := x + 3;
end;

begin
  foo[1]  := 1;
  foo[2]  := 2;
  t       := 1;
  shoot_my (foo[t]);
end.
```

Mode	foo[1]	foo[2]	t
Val	6	2	2
Val-Res (ALGOL W)	6	4	2
Val-Res (Ada)	4	2	2
Ref	9	2	2
Name			

Exhibit the differences (Explicit lyrics. . .)

```
var t  : integer
    foo: array [1..2] of integer;

procedure shoot_my (x : Mode integer);
begin
  foo[1]  := 6;
  t       := 2;
  x       := x + 3;
end;

begin
  foo[1]  := 1;
  foo[2]  := 2;
  t       := 1;
  shoot_my (foo[t]);
end.
```

Mode	foo[1]	foo[2]	t
Val	6	2	2
Val-Res (ALGOL W)	6	4	2
Val-Res (Ada)	4	2	2
Ref	9	2	2
Name	6	5	2

Table of Contents

Call by Sharing – definition

Call by sharing implies that values in the language are based on objects rather than primitive types, i.e. that all values are "boxed"

Differs from both call-by-value and call-by-reference.

```python
def f(list):
    list.append(1)

m = []
f(m)
print(m)
```

Call by sharing in Python – **output: [1]**

```python
def f(list):
    list = [1]

m = []
f(m)
print(m)
```

Call by sharing in Python – **output: []**

Notes on Call-by-sharing

Mutations of arguments performsmall by the called routine will be visible to the caller.

Notes on Call-by-sharing

Mutations of arguments performsmall by the called routine
will be visible to the caller.

Access is not given to the variables of the caller,
but merely to certain objects

Notes on Call-by-sharing

Mutations of arguments performsall by the called routine
will be visible to the caller.

Access is not given to the variables of the caller,
but merely to certain objects

Can be seen as "call by value" in the case where the value is an object reference

Notes on Call-by-sharing

Mutations of arguments performsall by the called routine will be visible to the caller.

Access is not given to the variables of the caller, but merely to certain objects

Can be seen as "call by value" in the case where the value is an object reference

- First introduced by Barbara Liskov for CLU language (1974)
- Widely used by: Python, Java, Ruby, JavaScript, Scheme, OCaml, AppleScript, ...
- **call by sharing** is not in common use; the terminology is inconsistent across different sources.

Table of Contents

Return Statement

What is the purpose of the return statement?

Is there a best way to return something?

Is there a best way to return something?

Table of Contents

Return via a dedicated keyword 1/2

```
int compute(int a, int b) {
  int res = a+b;
  // Some computation
  return res;
}
```

C's return statement uses the **return** keyword

```
int compute(int a, int b) {
  int r_val = a+b;
  // Some computation
  return r_val;
}
```

Java's return statement also uses the **return** keyword

Return via a dedicated keyword 2/2

The **return** statement breaks the current fonction (also for C++, Java, Ada, Modula2).

- Clarity
- Complexify the code
  - No naming convention
  - No homogeneous return inside a given fonction
  - Blur the comprehension via initialisation, intermediate computation, ...

Table of Contents

Return via function's name

```
function sum (a, b: integer): integer;
begin
  sum := a + b;
end;
```

Pascal's return statement uses the name of the function

Return via function's name

```
function sum (a, b: integer): integer;
begin
  sum := a + b;
end;
```

Pascal's return statement uses the name of the function

The name of the function is treated as a variable name (also for Fortran, ALGOL, ALGOL68, Simula)

- The "return" may not be the latest statement
- Ambiguous
  - For recursion **sum** denotes a variable **AND** a function
  - Is *somevar* := *sum* legal? (Yes for Pascal, No for Fortan)

Table of Contents

Return via a specific variable (1/2)

```
always_true : BOOLEAN
do
 Result := true
end
```

```
always_one : INTEGER
do
 Result := 1
end
```

```
always_bar : STRING
do
 Result := "bar"
end
```

Effeil's return statement uses the keyword **Result**

Return via a specific variable (2/2)

- The value returned by a function is whatever value is in **Result** when the function ends.
- The return value of a feature is set by assigning it to the **Result** variable (initialised automatically to a default value).
- Unlike other languages, the **return** statement does not exist.

Only in Effeil (to my knowledge)

- Clarity
- Ambiguous if the langage support nested fonctions

Table of Contents

Return the last computed value 1 /2

```
(defun double (x) (* x 2))
```

Lisp's return value is the last computed value

```
fn is_divisible_by(lhs: u32, rhs: u32) -> bool {
    if rhs == 0 {
        return false;
    }
    // The `return` keyword isn't necessary
    lhs % rhs == 0
}
```

For expressions, **Rust's return value** is the last computed value

Return the last computed value 2 /2

In **expression-oriented programming language** (also Lisp, Perl, Javascript and Ruby) the return statement can omitted.

- Instead that the last evaluated expression is the return value.
- A "last expression" is mandatory in Rust
- If no "return" Python returns **None** and Javascript **undefined**

Table of Contents

Named return values and Naked return

```
func make(r int, i int) (re int, im int) {
    re = r
    im = i
    return
}
```

Go combines Named returns values and naked return

Named return values and Naked return

```
func make(r int, i int) (re int, im int) {
    re = r
    im = i
    return
}
```

Go combines Named returns values and naked return

- No declaration/initialisation in the body of the function
- It serves as documentation.
- Functions that return multiple values are hard to name clearly GetUsernameAndPassword
- The signature of the function is slightly more difficult to read

Table of Contents

Subprograms as arguments

```
function diff (f(x: real): real,
               x, h: real) : real;
begin
  if h = 0 then
    slope := 0
  else
    slope := (f (x + h) - f (x)) / h;
  diff := slope
end

begin
  ...
  diff (sin, 1, 0.01);
  ...
end
```

- Typing difficulties ignored in ALGOL 60, Fortran, original Pascal and C: the function-argument was not typed.
- Today function types are available in most languages (except in some OOL).
- Doesn't exist in Ada. Simulated by a function parametrized routine. But you have to instantiate...

Anonymous subprograms

In all the functional languages, but not only (see automake)...

```
use Getopt::Long;
Getopt::Long::config ("bundling", "pass_through");
Getopt::Long::GetOptions
  (
   'version'          => &version,
   'help'             => &usage,
   'libdir:s'         => $libdir,
   'gnu'              => sub { set_strictness ('gnu'); },
   'gnits'            => sub { set_strictness ('gnits'); }
   'cygnus'           => $cygnus_mode,
   'foreign'          => sub { set_strictness ('foreign');
   'include-deps'     => sub { $use_dependencies = 1; },
   'i|ignore-deps'    => sub { $use_dependencies = 0; },
   'no-force'         => sub { $force_generation = 0; },
   'o|output-dir:s'   => $output_directory,
   'v|verbose'        => $verbose,
  )
    or exit 1;
```

Environment capture

Functional languages with block structure.

```
let type intfun = int -> int
    function add (n: int) : intfun =
      let function res (m: int): int = n + m in
    var addFive : intfun := add (5)
    var addTen           := add (10)
    var twenty  := addTen (addFive (5))
in
  twenty = 20
end
```

Create *closures*: a pointer to the (runtime) environment in addition to a pointer to the code. Somewhat hard to implement [Chap. 15](appel.98.modern).